Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Creación de un motor de videojuegos 2D multiplataforma para sistemas empotrados

Estudante: Nicolás Vázquez Cancela

Dirección: Emilio José Padrón González

A Coruña, octubre de 2023.

Para todos aquellos niños con dificultades para el aprendizaje, quedando demostrado que con perseverancia cualquier cosa es posible.

Agradecimientos

Dedico este logro a todos mis seres queridos, quienes me brindaron un apoyo inquebrantable a lo largo de mis estudios. Esto incluye a aquellos que, aunque no pudieron estar presentes en este momento, siempre estarán en mi corazón.

A todos los profesores que alguna vez me ayudaron para poder llegar hasta aquí y que supieron encaminar mi futuro profesional a través de la pasión por sus materias que compartían desinteresádamente.

Y finalmente, a los compañeros y amigos que me acompañaron con paciencia y comprensión en este camino.

Resumen

En este proyecto se describe el proceso completo de desarrollo de un motor gráfico destinado a la creación de videojuegos 2D, diseñado específicamente para sistemas empotrados y compatible con diversas plataformas. La implementación se realiza utilizando el lenguaje de programación C++ y se aprovecha la funcionalidad proporcionada por la popular biblioteca multimedia SDL (Simple DirectMedia Layer) para el manejo de eventos, renderizado, audio, entrada y salida.

El motor gráfico se compone de clases abstractas, siguiendo un patrón de plantilla. Estas clases deben ser implementadas para representar objetos del videojuego, otorgándoles diversas funcionalidades. Además, se incluye una API para la gestión de recursos, como ficheros, escenas, *sprite sheets*, animaciones, sonidos y música. Con el motor se incluye un editor de escenas en línea de comandos, que facilita la creación y modificación de escenas y pantallas, junto con sus objetos. Finalmente, el motor culmina en la generación del ejecutable del juego, que implementa su bucle principal.

El proyecto incluye, además, un pequeño juego de demostración, realizado íntegramente con el motor propuesto, probado tanto en una Raspberry Pi 1 como en un PC.

Abstract

This project outlines the complete process of creating a game engine designed for the development of 2D video games, specifically tailored for embedded systems and compatible across various platforms. The implementation was done using the C++ programming language, harnessing the capabilities provided by the widely-used multimedia library SDL(Simple DirectMedia Layer) for event handling, rendering, audio, input and output.

The graphics engine consists of abstract classes following a template pattern. These classes need to be implemented to represent game objects, giving to them diverse functionalities. Furthermore, an API is included for resource management, like files, scenes, sprite sheets, animations, sounds, and music. Has a scene editor to allow the creation and modification of scenes, along with their respective objects. Ultimately, the engine culminates in generating the game's executable, which implements its main loop.

Palabras clave:

Motor gráfico

Multiplataforma

Sistema empotrado

Videojuego 2D

Objeto del juego

Escena del juego

Editor de escenas

Sprite

Tile

Keywords:

Game engine

Cross-platform

Embedded system

2D video game

Game object

Game Scene

Scene editor

Sprite

Tile

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Un videojuego es un producto software de gran complejidad, en los que hay que lidiar con distintos subsistemas y sus diferentes problemáticas (gráficos, mecánicas, sonido, IA, etc.) a la vez que se garantiza una experiencia interactiva de calidad. Para permitir a los equipos de desarrollo centrarse en los aspectos más relacionados con el diseño de un videojuego, sin tener que preocuparse en exceso de la parte más técnica, es habitual disponer de un motor de videojuegos que esconda gran parte de la complejidad técnica del sistema, al menos hasta los ajustes finales.

En este proyecto proponemos el desarrollo de un motor de videojuegos 2D orientado a sistemas empotrados de propósito general de bajo coste (y bajas prestaciones), como pueda ser una Raspberry Pi. En este tipo de sistemas no disponemos de los mismos recursos que en un PC o una videoconsola, por lo que cobra especial importancia disponer de un motor ligero que haga una gestión eficiente de los mismos.

1.1 Contexto y motivación

El concepto de motor gráfico se refiere a un conjunto de herramientas y tecnologías utilizadas para crear y desarrollar gráficos, físicas y mecánicas en los videojuegos, principalmente. A medida que los videojuegos evolucionaron desde sus inicios simples en las décadas de 1970 y 1980, se volvió evidente que había una necesidad de estandarizar ciertas funciones para facilitar el desarrollo de juegos más avanzados, sobre todo según fueron creciendo en complejidad.

En la década de 1970 [1], los primeros videojuegos eran simples y se ejecutaban en hardware dedicado. Juegos como "Pong" se basaban en sistemas empotrados diseñados específicamente para un solo propósito: jugar a ese juego en particular. Estos sistemas a menudo consistían en circuitos personalizados.

La década de 1980 [2, 3, 4] vio el surgimiento de consolas de videojuegos como la Atari 2600

y la Nintendo Entertainment System (NES). Estas consolas incorporaban sistemas empotrados que permitían a los jugadores disfrutar de una variedad de juegos en un solo dispositivo. Esto marcó el inicio de la popularización de los videojuegos en los hogares. Al mismo tiempo [5], los primeros ordenadores personales como el Commodore 64 y el ZX Spectrum, entre otros, también se convirtieron en plataformas de juego populares, con procesadores de 8-bits que permitían a los usuarios crear y jugar a una amplia variedad de títulos, fomentando la creatividad de los desarrolladores y la diversidad de juegos a la vez que se aprendía a programar con BASIC.

En las décadas de 1990 y 2000 [6], surgieron motores gráficos notables como el motor *id Tech* de id Software, que impulsó juegos como "Doom" y "Quake". Posteriormente, otros motores como *Unreal Engine* de Epic Games y *Unity* ganaron importancia, brindando a los desarrolladores una base sólida para crear rápidamente prototipos de videojuegos. En la década de 2010, aparecen los primeros teléfonos móviles inteligentes con pantallas táctiles considerados inicialmente como sistemas empotrados. Estos dispositivos abren nuevas posibilidades de interacción en los videojuegos por sus características técnicas y permiten a más gente probarlos al ser más baratos y cómodos que los ordenadores y consolas.

Durante todo este tiempo, la industria de los videojuegos ha experimentado un imparable crecimiento, pasando de ser un nicho de entretenimiento a una forma dominante de entretenimiento global, rivalizando con la industria del cine en términos de ingresos generados [1]. A pesar de toda esta evolución, el desarrollo de videojuegos sigue siendo un proceso complejo y costoso. El precio de las licencias de los motores gráficos y programas de modelaje 3D, diseño gráfico, y edición de sonido, aumenta significativamente los costos de producción, sin contar el coste del gran equipo multidisciplinar de artistas, diseñadores, programadores y escritores. Además, la curva de aprendizaje para dominar estas herramientas y tecnologías necesarias puede ser empinada, convirtiéndose en un obstáculo para toda persona con interés en la industria.

Estas son algunas de las razones porque los desarrolladores de videojuegos independientes [2] optan por crear desde cero videojuegos 2D, ya que son intrínsecamente menos complejos que el 3D, requiriendo una menor inversión de recursos económicos y humanos, y permitiendo centrarse más en la jugabilidad, la narrativa y la creatividad en lugar de lidiar con aspectos técnicos y de diseño más complejos.

Aprovechando los menores requisitos técnicos que requieren este tipo de videojuegos, resulta interesante hacerlos capaces de ejecutarse en plataformas muy limitadas computacionalmente como son los sistemas empotrados. Aprendiendo programación de bajo nivel en un entorno mucho más simple, técnicamente hablando, similar al estilo de desarrollo que se

[1] Más información sobre los datos de la industria de los videojuegos: https://en.wikipedia.org/wiki/Video_game_industry.

[2] Información sobre los videojuegos independientes: https://en.wikipedia.org/wiki/Indie_game.

usaba clásicamente para implementar los videojuegos [5]. Además, los sistemas empotrados ofrecen un menor consumo y tamaño, posibilitando desarrollar y probar los videojuegos en cualquier lugar.

1.2 Objetivos

El motor gráfico propuesto en este proyecto aspira a solucionar los inconvenientes previamente mencionados, aportando una base técnica simple y de código libre para que los desarrolladores de videojuegos 2D independientes tengan dónde empezar a crear sus proyectos sin necesidad de comenzar de cero, manteniendo todo el control sobre los recursos técnicos, y permitiendo la extensión del motor si se precisa, siguiendo el principio de Abierto Cerrado [7].

El motor gráfico también busca ser versátil y adaptable, permitiendo a los desarrolladores crear videojuegos de diversos géneros y estilos, desde plataformas y rompecabezas hasta juegos de rol y aventuras gráficas.

Además, deberá permitir a los desarrolladores crear los juegos y ejecutarlos eficientemente en distintas plataformas, como PC, consolas, y dispositivos móviles, especialmente en sistemas empotrados como la Raspberry Pi, sin tener que preocuparse por la complejidad específica de la implementación en cada una de ellas, abstrayendo los detalles técnicos y específicos de cada arquitectura.

El motor consta de una biblioteca C++ que, junto a algunas herramientas adicionales, facilitará la creación de videojuegos 2D multiplataforma, llevando a cabo las siguientes funciones básicas que requiere un videojuego 2D:

- Carga y descarga: cómo se debe realizar la inicialización y finalización de cada objeto para incluirlo dentro del videojuego.
- Interacción del jugador: haciendo que capturen eventos producidos por distintos periféricos.
- Actualización de objetos: permitiendo darles vida y que reaccionen a ciertas acciones.
- Dibujado, manejo de diferentes sistemas de coordenadas, imágenes, animaciones, y capas de profundidad en las se dibujan.
- Sonido, reproducción de efectos de sonidos y música.
- Persistencia, serialización en disco de la información necesaria para volver a cargar un objeto en un formato portátil.

Para desarrollar un videojuego 2D usando el motor propuesto, solo se deberá incluir la biblioteca principal y definir sus objetos siguiendo una estructura estándar y empleando las

funciones aportadas por esta. A mayores, la biblioteca ya implementa el bucle principal del juego y algunos objetos de utilidad para la gestión de eventos, escenas, etc.

Finalmente, y considerando esencial llevar a cabo la implementación de una demostración técnica efectiva del motor, se incorpora una pequeña demostración del mismo. Esta demostración no solo sirve como un testimonio tangible de las capacidades del motor, sino que también cumple un papel crucial al proporcionar un ejemplo concreto de cómo los desarrolladores pueden utilizarlo de manera práctica.

1.3 Soluciones existentes

En la actualidad, hay varias soluciones en el mercado para el desarrollo de videojuegos, como Game Maker Studio 2, Godot, Unity y Construct. Cada una de estas herramientas tiene sus propias ventajas y desventajas en términos de facilidad de uso, flexibilidad, escalabilidad y características específicas para el desarrollo de videojuegos 2D.

Los tres primeros podrían ser buenos candidatos para satisfacer las necesidades mencionadas en la sección de objetivos 1.2 ya que proporcional herramientas fáciles de usar y aprender para desarrollar videojuegos 2D, con grades comunidades, y soporte de la mayoría de plataformas actuales e incluso de la Raspberry Pi 3 y 4.

Sin embargo, estas dos últimas versiones de los sistemas empotrados Raspberry Pi están muy cotizadas y resulta casi imposible conseguir una en la actualidad. Además, es imposible desarrollar los videojuegos en las Raspberry Pi y salvo Godot, ningún motor es de código abierto y requieren pagar una licencia por usar su versión completa.

Capítulo 2

Metodología y planificación

En este capítulo se hablará de la metodología seleccionada para llevar acabo el proyecto y se justificará por qué se cree que es la más adecuada para cumplir los objetivos mencionados en la Sección 1.2, teniendo en cuenta sus características. Luego, se explicará e ilustrará cómo esta metodología afectará a planificación de las distintas tareas del proyecto, estimando la duración y recursos humanos, materiales y maquinarias que requiere cada una para poder ser completada. Finalmente, se analizarán los riesgos considerados.

2.1 Metodología

Escoger una metodología adecuada para la gestión de un proyecto software, que se ajuste a sus características particulares y a los recursos disponibles para llevarlo a cabo, desempeña un papel fundamental en el éxito y la eficiencia del desarrollo. Esta decisión no solo afecta a la estructura y organización del equipo de desarrollo, sino que también influye en la planificación de las tareas, la asignación de recursos y la calidad del producto final.

Cuando se trata de motores gráficos interactivos, la complejidad inherente a la creación de un sistema que maneje gráficos, audio, físicas y otros aspectos técnicos requiere una planificación sólida y una metodología que garantice la eficiencia en cada etapa del desarrollo.

En este caso se decidió usar una metodología ágil e iterativa basada en el modelo en espiral de Barry Boehm [1] basándose en los consejos del libro *The Art of Game Design* [8].

Este modelo nos sugiere seguir los siguientes pasos a lo largo de todo el proyecto, ilustrados en la imagen de la Figura 2.1:

1. Comenzar con un diseño básico.
2. Averiguar los mayores riesgos del diseño.
3. Construir prototipos que reduzcan esos riesgos.

[1] Información sobre el modelo en espiral: https://en.wikipedia.org/wiki/Spiral_model.

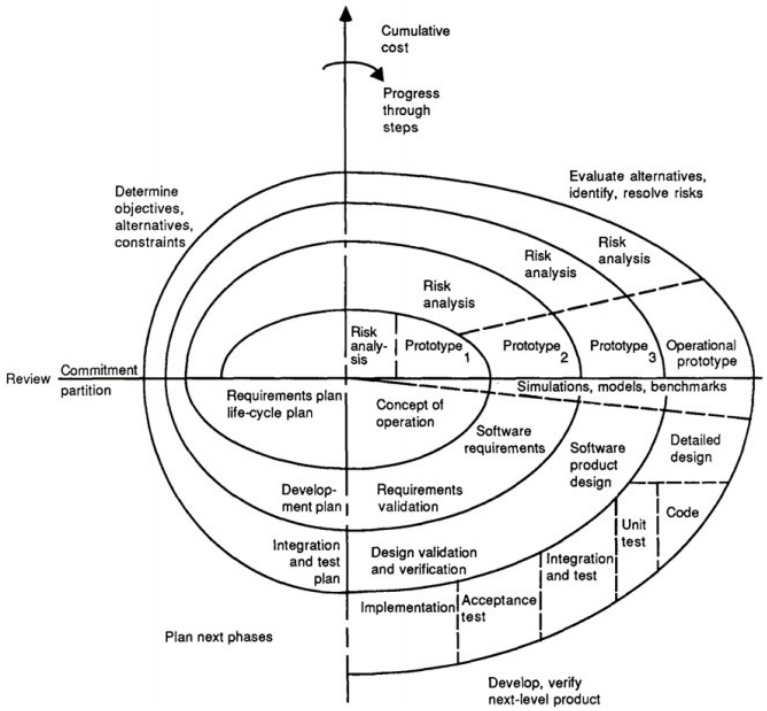

Figura 2.1: Desarrollo software basado en el modelo en espiral de Barry Boehm

4. Probar los prototipos.
5. Elaborar un diseño más elaborado y detallado basado en lo aprendido.
6. Volver al segundo paso.

Cuantas más veces se repitan estos pasos, más completo y complejo se volverá el proyecto, pero nos permite disponer en todo momento de un prototipo funcional que evidencie el progreso realizado hasta la fecha, algo muy importante cuando se esta debajo de la supervisión de un cliente o profesor en este caso.

Entre las ventajas que en [8] se mencionan de usar el enfoque en espiral para desarrollar videojuegos y motores gráficos, frente a una metodología más formal como un desarrollo en cascada, son las siguientes:

1. Permite abordar estas complejidades de manera gradual y sistemática.
2. Se adapta bien a situaciones en las que los cambios son inevitables, ya que permite la revisión y adaptación periódica de los objetivos y entregables.
3. Posibilita iterar y experimentar en las fases de desarrollo.
4. Hace más fácil identificar y abordar los riesgos de manera proactiva.

Estas ventajas se vuelven aún más convenientes cuando no se tiene experiencia previa desarrollando un proyecto de esta escala, y con una duración muy limitada y unos plazos muy ajustados, como es el caso.

2.2 Recursos

Durante las tareas del proyecto serán necesarios una serie de recursos humanos, materiales y del tipo maquinaria que harán posible alcanzar las metas de estas tareas. A continuación se lista cuales fueron los recursos escogidos para cada tipo con sus costos correspondientes:

2.2.1 Recursos humanos

A continuación, en el Cuadro 2.1, se presentan los miembros del proyecto indicando el puesto y tareas que desempeñaran junto al salario que cobraran por ello. Para este último dato, se baso en las estadísticas proporcionadas por el estudio salarial realizado por la consultoría *Vitae* [9] en el sector de las Tecnologías de la Información de la Galicia situada entre los años 2015 y 2016. Debido a la antigüedad de estas estadísticas y a la gran inflación de la moneda europea en Galicia, se realizó una corrección de los salarios en base al porcentaje de inflación ocurrido entre estos años, obtenido mediante la herramienta de cálculo del indice de inflación que proporciona el portal del Instituto Nacional de Estadística [2].

Puesto	Tareas asignadas	Salario (€/h)
Jefe de proyecto software	Análisis de requisitos y riesgos del prototipo	26,56
Analista programador C/C++	Análisis de requisitos, diseño, implementación y prueba del prototipo	10,12

Cuadro 2.1: Recursos humanos del proyecto con su puesto, asignación y salario por hora.

2.2.2 Recursos de tipo maquinaria

Para llevar a cabo las tareas relacionadas con el desarrollo del motor gráfico y la prueba de su demostración técnica, fueron necesario las siguientes herramientas presentadas en el Cuadro 2.2.

[2] https://www.ine.es/varipc

2.2.3 Recursos materiales

Durante el transcurso proyecto, se necesitaron varios útiles para poder emplear los recursos de tipo maquinaria mencionados en la anterior sección. Las unidades y coste de estes útiles se puede apreciar en el Cuadro 2.3.

2.3 Planificación

Siguiendo las directrices proporcionadas por el modelo en espiral, se ha planificado la ejecución del proyecto de manera que las funcionalidades del motor gráfico se integren gradualmente y de manera estructurada. Estas funcionalidades se diseñaran, analizaran, implementaran, y finalmente probarán en cada iteración a través de la construcción de demostraciones técnicas, lo que permitirá un crecimiento simultáneo tanto del motor gráfico como de la demostración técnica.

Con el fin de facilitar el seguimiento del proyecto de una manera visual, se realizo el diagrama de Gantt [3], visible en la Figura 2.2, donde las etapas del proyecto se descomponen en tareas y se establece su duración y las relaciones entre ellas.

El diagrama de Gantt también no sirve para poder percibir como de bien se cumplieron las estimaciones sobre la duración de cada tarea, y como, ante el retraso de alguna de ellas, repercute en la fecha de finalización del proyecto. En el caso de este proyecto, debido a la inesperada y reducida disponibilidad de los recursos humanos en dos primeras tareas, riesgo contemplado en la sección 2.4, resulto en un grave retraso de todas las tareas del proyecto, teniendo que aplazar la entrega final del mismo. Esto queda reflejado en el diagrama de Gantt de la Figura 2.3, realizado tras la finalización del proyecto.

2.3.1 Fase inicial

Para poder iniciar el desarrollo del proyecto, es fundamental primero hacer un análisis detallado y establecer los requisitos mínimos del motor gráfico, los cuales deben satisfacer los objetivos definidos en la sección 1.2.

Al mismo tiempo, se identificarán las herramientas y tecnologías adecuadas a las características del proyecto y que permitan cumplir los requisitos previamente establecidos, como el lenguaje de programación, bibliotecas y entorno de desarrollo, que nos faciliten la implementación y su posterior mantenimiento.

[3] https://es.wikipedia.org/wiki/Diagrama_de_Gantt

Herramienta	Unidades	Precio (€/ud.)	Total (€)
Portátil personal	1	200 (aprox.)	200
Sistema empotrado Raspberry Pi 1 Model B	2	26 (segunda mano)	52
Pantalla HD	1	100 (aprox.)	100
		Total	352

Cuadro 2.2: Recursos de tipo maquinaria del proyecto con su número de unidades, precio por unidad, y coste total.

Útil	Unidades	Precio (€/ud.)	Total (€)
Cargador Raspberry Pi	2	10	20
Cable RJ45 UTP 2 metros	2	5	10
Cable HDMI 2 metros	2	5	10
Carcasa Raspberry Pi 1 Model B (Impresa en 3D)	1	2	2
Tarjeta microSD con adaptador SD de 8 GB	2	5	10
Teclado	2	5	10
Ratón	2	5	10
Mando de consola	1	25	25
		Total	97

Cuadro 2.3: Recursos materiales del proyecto con su número de unidades, precio por unidad, y coste total.

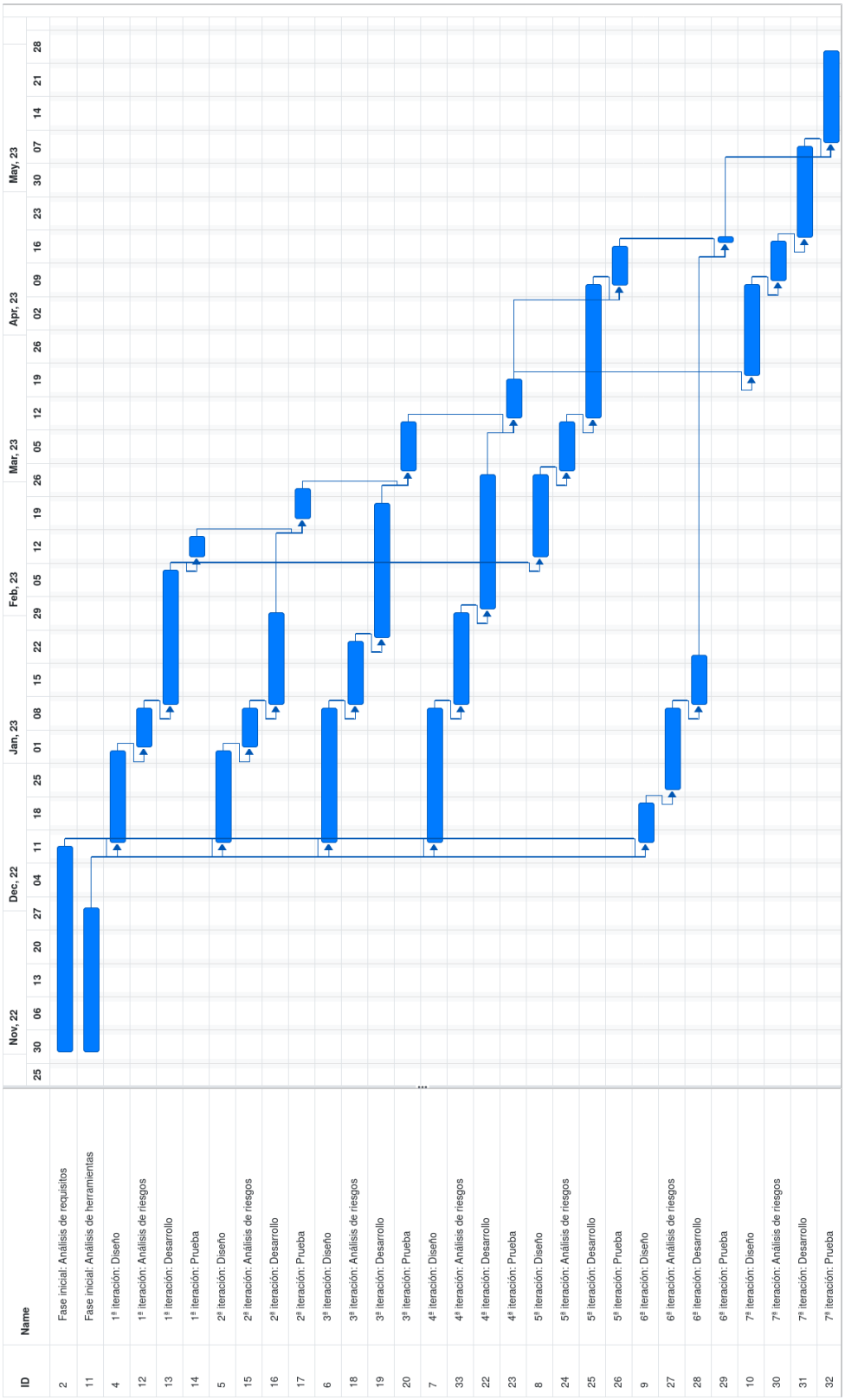

Figura 2.2: Diagrama de Gantt para la planificación inicial.

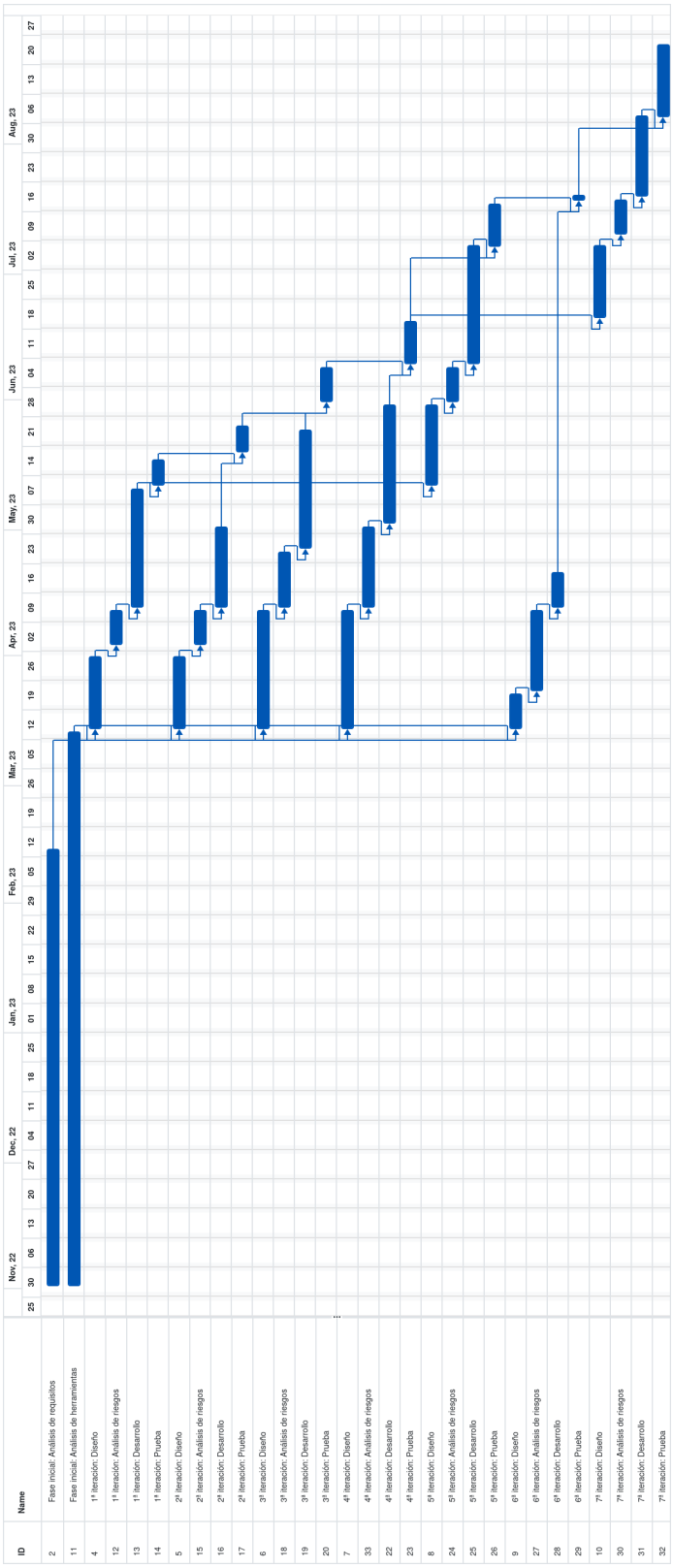

Figura 2.3: Diagrama de Gantt para la ejecución de la planificación.

2.3.2 Primera iteración: Hola mundo

Esta primera iteración se enfoca en conseguir el objetivo básico de crear la ventana principal del juego y renderizar formas básicas como puntos y rectángulos, lo que proporciona una base para la representación visual de los elementos de un videojuego.

Esta tarea puede parecer simple en una primera instancia, pero dependiendo del lenguaje de programación, librería gráfica, y de la plataforma, dibujar un simple triangulo puede requerir cientos de lineas de código y horas para configurar el dispositivo.

A mayores, será necesario implementar el dibujado de elementos relativos a una cámara virtual, lo que facilita el desplazamiento y la visualización de áreas más grandes del mundo del juego.

2.3.3 Segunda iteración: Interacción con el jugador/a

El siguiente paso lógico en el proceso de desarrollo del motor gráfico sería la incorporación de la gestión de eventos del jugador/a, que abarca la interacción con varios periféricos como el teclado, el ratón y joysticks. Esta etapa es de vital importancia, ya que permite que los jugadores interactúen con los elementos del videojuegos.

Integrar esta gestión de eventos de manera eficiente y efectiva en el motor gráfico es fundamental para proporcionar una experiencia de juego fluida y receptiva. Además, permite la implementación de controles personalizables, lo que aumenta la accesibilidad y la adaptabilidad del juego a diferentes preferencias de los jugadores.

2.3.4 Tercera iteración: Actualización los objetos del juego

Otro aspecto clave a implementar de los juegos es la interacción de los objetos del juego con el mundo virtual. Para ello, es necesario actualizar sus estados internos de forma independiente en cada ciclo del bucle principal. Lo que permitirá que reaccionen a ciertos eventos y dotarlos de inteligencia artificial.

También, sería conveniente para el usuario desarrollar un motor de físicas simple para detectar colisiones y aplicar leyes físicas básicas de forma eficiente sobre los objetos del juego que se desee.

2.3.5 Cuarta iteración: Creación de escenas

Las escenas de un motor gráfico son un elemento crucial en los juegos para dividir su lógica en unidades más manejable; agrupando los objetos del juego que pertenecen al mismo contexto para gestionar sus recursos, como gráficos o sonidos. Esto asegura que los recursos se carguen y descarguen de manera eficiente según la escena en la que se encuentre el jugador/a, lo que ahorra memoria y recursos del sistema. Cada escena puede representar una pantalla,

etapa, nivel o situación dentro del juego, y la lógica relacionada se concentra en esa escena. El uso de este recurso facilita la gestión del código y la depuración, ya que cada escena se puede desarrollar y probar por separado.

En esta fase, se llevará a cabo el diseño e implementación de un sistema para administrar escenas y sus objetos en el motor gráfico. Este sistema permitirá crear y eliminar escenas, reservando y liberando los recursos necesarios para crear los objetos del juego que contienen. También deberá poder guardar los objetos de la escena en un fichero con un formato bien estructurado, que requiera el menor espacio posible y permita su posterior carga de forma rápida y eficiente. Finalmente, el sistema gestionará la activación, y desactivación, de las escenas y sus objetos para ahorrar el procesamiento asociado a estos elementos cuando no es necesario, pero manteniendo sus recursos cargados para reactivarlos más tarde.

Aprovechando este sistema, lo siguiente será desarrollar un programa para la edición de escenas. En este programa se podrá crear y guardar nuevas escenas, y añadir, borrar, listar, visualizar y modificar los atributos de los objetos del juego que el usuario haya implementado.

Una vez establecido este sistema, se procederá al desarrollo de una herramienta simple de edición de escenas. Esta aplicación proporcionará la posibilidad de crear una escena vacía o cargarla a partir de un fichero previamente creado. Entre las funcionalidades destacadas se incluyen la adición, eliminación, guardado, listado de objetos; y la visualización y modificación de sus atributos. Este primer editor de escenas se usará mediante una interfaz de línea de comandos para ejecutar los casos de uso mencionados. Eso sí, todos los cambios en la escena que se van realizando mediante las distintas acciones en el editor son actualizados en tiempo real, pudiendo visualizar su nuevo estado en una ventana auxiliar.

2.3.6 Quinta iteración: Gráficos y sonidos

Algo fundamental para mejorar la experiencia de juego es ofrecer la posibilidad en el motor de renderizar texturas y de reproducir efectos de sonido y música. Además, el motor debe incorporar mecanismos para que la introdución de estos elementos más artísticos en el desarrollo de un videojuego permita a las personas involucradas en el mismo expresar su creatividad, con un balance entre flexibilidad y sencillez.

Con ese objetivo se desarrollarán dos interfaces sencillas pero completas para el manejo de gráficos y sonidos. La primera deberá ofrecer las tres transformaciones geométricas básicas de transposición, rotación, y escalado con texturas. La segunda podrá reproducir una canción en bucle y varios sonidos, cambiar el volumen de la música y sonidos independientemente para ajustarse a las preferencias del jugador/a, y pausar y resumir la canción actual.

Además, se incorporará al motor gráfico la gestión eficiente de ficheros de imágenes y sonidos para ser usados por estas dos interfaces.

2.3.7 Sexta iteración: Guardar y cargar partida

Cuando una partida a un videojuego puede tener una duración excesiva, es necesario ofrecer la posibilidad de pausar la partida para retomarla en otro momento. Esto implica para el videojuego tener que guardar el estado actual de ciertos objetos en algún tipo de memoria persistente, como un fichero o una base de datos.

Para el tipo de videojuegos que pretende cubrir el motor gráfico propuesto será suficiente con proveer al desarrollador/a unas funciones que permitan escribir variables de distintos tipos en un fichero, de forma compacta e independiente de la plataforma, que garantice poder continuar el juego incluso en otro dispositivo.

2.3.8 Séptima iteración: Transición y múltiples escenas

Ya contando con el concepto de escenas bien implementado, en esta iteración nos proponemos ofrecer al desarrollador/a un mecanismo para transicionar entre ellas, permitiendo la creación de juegos que no solo consistan en una pantalla: desde juegos con varios niveles distintos hasta extensos mundos abiertos cargados progresivamente.

Para ello se desarrollará un gestor de escenas que abstraiga al desarrollador/a de las operaciones mencionadas en la Sección 2.3.5 para realizar las siguientes operaciones:

1. Cambiar a una nueva escena. Esto implica cargar una nueva escena y descargar las escenas previamente cargadas.

2. Añadir una nueva escena. Idéntica a la anterior operación, salvo que se mantienen las escenas ya cargadas, permitiendo múltiples escenas cargadas al mismo tiempo.

3. Ocultar una escena. Sirve para desactivar la escena y sus objetos, pudiendo reactivarla más tarde si se requiere.

4. Cerrar una escena. Asegura que se descargue por completo la escena seleccionada, pudiendo estar incluso ya desactivada.

5. Seleccionar la escena activa. Esta será la escena donde se guarden los nuevo objetos creados.

6. Obtener la escena activa. Entrega información sobre ella.

Disponiendo del gestor de escenas, otra tarea podría ser extender el editor para crear y abrir múltiples escenas a la vez. Esto puede ser muy útil para visualizar el contexto que generan varias escenas que tienen que coexistir y cómo interactuarían sus objetos, lo que sería muy incómodo de hacer por separado. Además, será necesario dar alguna forma de reconocer y seleccionar en el editor qué escena de las cargadas es la que se esta modificando.

2.4 Riesgos considerados

En esta sección se describen y valoran los problemas que podrían surgir en el proyecto antes de empezarlo. No obstante, debido a la metodología iterativa a la que el proyecto se rige, esta no será la única vez que se analicen los riesgos que puedan acontecer.

1. **No disponer de los recursos en todo momento:** Debido a la incierta y limitada disposición de los miembros de este proyecto, mencionados en la sección 2.2, debido a las complicaciones que puedan darse, ni disponer del 100% de su tiempo laboral, pueden ocurrir retrasos en los entregables de algunas tareas, repercutiendo drásticamente en la fecha limite del proyecto si esta se encuentra en su camino crítico. Además, puede haber también dificultades para conseguir el sistema empotrado de los recursos materiales a causa del encarecimiento global de semiconductores que esta ocurriendo, retrasando la ejecución de las pruebas en esta plataforma hasta disponer del sistema empotrado.

2. **Errores de diseño e implementación:** Debido a la gran complejidad técnica inherente de este proyecto y a la falta de experiencia de sus miembros desarrollando motores gráficos, es posible que se cometan fallos en los diseños realizados para satisfacer los requisitos del proyecto y en la implementación de estos, haciendo que dichas tareas requirieran de más tiempo para la refactorización de estos diseños y sus códigos.

3. **Complicaciones en la preparación del sistema empotrado:** Dada a la falta de información sobre el sistema empotrado y a la poca experiencia manejando esta clase de dispositivos, es posible necesitar más tiempo para preparar el sistema empotrado para su uso en pruebas de la demostración técnica del motor gráfico, o incluso requerir cambiar el sistema empotrado por otro distinto, con todos los costes que conlleva.

4. **Fallo hardware/software del ordenador y sistema empotrado:** Al manipular y usar estos dispositivos electrónicos es posible que se dallen por errores ajenos al proyecto, pudiendo necesitar un remplazo si se da el caso, lo que acarrearía más costes económicos y tiempo al proyecto.

5. **Problemas con las pruebas en el sistema empotrado:** Durante las pruebas de la demostración técnica del motor gráfico, pueden darse fallos técnicos que no se produjeran al hacerlas en el propio ordenador de desarrollo, complicando y ralentizando el proceso de depuración al tenerse que hacer directamente en el sistema empotrado con mucha menor potencia. Estos fallos normalmente se deben a las diferencias al implementar el compilador de un lenguaje de programación para cada arquitectura de procesador, y a los fallos cometidos durante él desarrollo de nuestro propio programa, que pueden pasar por alto en un compilador y en otro no.

Para valorar las consecuencias que tendrían los anteriores riesgos explicados, es necesario clasificar el grado de impacto que tendían sobre el proyecto, que tan expuesto este encuentra a los riesgos, y la probabilidad de cada uno. En el Cuadro 2.4 se puede observar la clasificación de los riesgos.

Riesgo	Probabilidad	Impacto	Exposición
No disponer de los recursos en todo momento	Media	Alto	Alto
Errores de diseño e implementación	Baja	Medio	Bajo
Complicaciones en la preparación del sistema empotrado	Media	Medio	Bajo
Fallo hardware/software del ordenador y sistema empotrado	Baja	Medio	Alta
Problemas con las pruebas en el sistema empotrado	Alto	Baja	Media

Cuadro 2.4: Probabilidad, impacto y exposición de los riesgos considerados inicialmente en el proyecto.

Capítulo 3

Solución

En este capítulo se describirá detalladamente todo el trabajo realizado a lo largo del proyecto, pasando por el análisis de requisitos, diseño, gestión de riesgos, desarrollo, y prueba de la solución, todos procesos necesarios para la gestión del proyecto.

Debido a la naturaleza iterativa del modelo en espiral (metodología a la que se ciñó el desarrollo del proyecto, descrita en el Capitulo 2), estos procesos se repitieron por cada etapa de la planificación. Por eso, este capitulo se dividió por secciones para cada una, y subsecciones para sus procesos.

3.1 Fase inicial: Análisis de requisitos del motor y selección de herramientas

En esta fase inicial del proyecto, que sigue rigurosamente la planificación previamente establecida, el enfoque principal será llevar a cabo un análisis exhaustivo de los requisitos que serán solicitados al motor gráfico. Este análisis abarcará no solo los aspectos técnicos, sino también los requisitos no funcionales, que desempeñan un papel igualmente crítico en el éxito y eficiencia del motor.

Luego, se buscarán las tecnologías y herramientas que se usarán para implementar el motor gráfico. Entre ellas, deberá encontrarse el ordenador de moderadas prestaciones donde se desarrollará, el sistema empotrado para probarlo (también se probará el motor en el ordenador para asegurar portabilidad), sus sistemas operativos, y el lenguaje de programación principal de su implementación junto a su correspondiente cadena de herramientas para cada plataforma. Este cadena de herramientas consistirá en un editor de texto con el que editar cómodamente el código fuente, un compilador y enlazador para transformar el código fuente en un programa ejecutable o interprete para analizar y ejecutar programas, gestor de dependencias para construir el ejecutable, bibliotecas para proveer una interfaz al sistema operativo, un depurador para solucionar errores, y un sistema de control de versiones con el que registrar

los cambios del proyecto.

3.1.1 Análisis de requisitos del motor gráfico

Antes de empezar el proyecto, es necesario establecer unos requisitos funcionales y no funcionales mínimos que la solución deberá cumplir para poder ser usada. Estos serán mínimos debido a que, como cualquier otro proyecto, a medida que este se desarrolle podrán ir añadiéndose nuevos requisitos que cubran aspectos que se ignoraban al principio.

Requisitos funcionales

1. **Creación del contexto de renderizado:** Que permita abrir una ventana con un nombre y tamaño ajustable en la que renderizar.
2. **Control de la tasa de refresco del juego:** Para dar la mejor experiencia posible según la potencia de la plataforma.
3. **Renderizado 2D eficiente:** Que permita renderizar figuras básicas e imágenes *bitmap* de forma rápida y eficiente.
4. **Manejo de Sprites y Animaciones:** Debe proporcionar herramientas para gestionar y animar *sprites*, lo que es fundamental para juegos 2D.
5. **Gestión de eventos:** Debe ser capaz de gestionar la interacción del jugador/a de manera sencilla, incluyendo teclado, ratón o mando de consola.
6. **Programación de objetos basada en scripts:** Que permita definir el comportamiento y las interacciones de un elemento específico dentro del juego.
7. **Gestión de Sonido y Música:** Permitirá la reproducción de efectos de sonido y música en el juego.
8. **Comprobación de Colisiones:** Proporcionará mecanismos para detectar colisiones entre objetos en el juego.
9. **Soporte para Escenas y Capas:** Debe ofrece herramientas para la creación y gestión de escenas y capas para organizar los elementos visuales.
10. **Multiplataforma:** Que tanto las herramientas del motor y los juegos desarrollados con estas sean compatibles con múltiples plataformas, especialmente con sistemas empotrados.

Requisitos no funcionales

1. **Código abierto:** Permitiendo a cualquier interesado tanto desarrollar juegos con el motor o participar en su desarrollo mientras aprenden su funcionamiento.
2. **Documentación Completa y Clara:** Necesita contar con documentación detallada y fácil de entender para facilitar su uso y desarrollo.
3. **Sencillo y fácil usar:** Con una breve formación inicial, posibilitará al desarrollador/a la creación rápida de prototipos de sus juegos usando el motor.
4. **Flexibilidad y Escalabilidad:** Permitirá la expansión y personalización del motor para adaptarse a diferentes tipos de juegos y requisitos específicos del proyecto.
5. **Compatibilidad con Herramientas de Desarrollo:** Debe ser compatible con las herramientas y entornos de desarrollo comunes para facilitar la creación y depuración de juegos.
6. **Cumplimiento de Normas y Estándares:** Debería seguir las mejores prácticas de desarrollo y cumplir con estándares de la industria para garantizar la calidad y la interoperabilidad.

3.1.2 Selección de herramientas

A continuación, se analizarán los requisitos, las opciones, los riesgos, para finalmente justificar la elección de cada herramienta necesaria mencionada.

Sistema empotrado para las pruebas

Requisitos:

Funcionales:

1. Gran nivel de control. Al igual que el ordenador, deberá ofrecer fácilmente todo su potencial técnico.
2. CPU con al menos un núcleo de 700 MHz de 32 bits(aritmética en punto flotante opcional) y una arquitectura (típicamente basada en ARM) que soporte el sistema operativo y cadena de herramientas escogidos.
3. GPU integrada que soporte mínimo OpenGL ES 2.0.
4. Memoria RAM y VRAM que sumen un mínimo 512Mb.

5. Memoria flash preferente extraible con al menos 8 GB.
6. Pantalla integrada o alternativamente una salida de vídeo de HDMI o VGA con salida de audio jack para visualizar en una pantalla o proyector auxiliar las pruebas del motor gráfico.
7. Un puerto serie UART, un puerto RJ-45 Ethernet, o un módulo WIFI, para facilitar la actualización de su firmware y software y poder iniciar una sesión remota por terminal desde el ordenador con el sistema empotrado, amenizando la interacción y permitiendo la depuración de las pruebas en este.
8. Al menos un puerto USB que soporte un teclado, ratón y mando de consola, con los que poder interactuar con las pruebas del motor gráfico que se ejecuten en el sistema empotrado.

No funcionales:

1. Alto *stock*. Para asegurar un remplazo en caso de fallo irreversible.
2. Bajo coste. Que facilite su adquisición y minimice la inversión económica del proyecto.
3. Poco consumo. Que permita alimentarlo con un típico cargador de móvil de 5 voltios y 2.0 amperios o una batería.
4. Documentación y buen soporte al cliente. A poder ser en línea, facilitando la resolución de problemas relacionados con el hardware o software de la plataforma.

Riesgos:

1. Fallo hardware/software que impida usarlo. La sustitución podría consumir mucho tiempo del proyecto.
2. Dificultades para conseguir un ejemplar del sistema embebido escogido en caso de no disponer de uno.
3. Retrasos en el envío de ser necesario.
4. Problemas para instalar las herramientas necesarias en el sistema empotrado.
5. Rendimiento inferior a lo esperado al ejecutar las pruebas del motor gráfico en sistema empotrado.

Opciones:

1. *Dragon board 810 Development Kit*[10]

Aspectos positivos:

(a) Cumple más que de sobra todos los requisitos técnicos explicados, e incluso dispone de una pantalla integrada.

(b) Disponibilidad inmediata de varios ejemplares listos para usar sin coste alguno.

Aspectos negativos:

(a) Inexistente documentación y soporte al cliente debido a que el fabricante discontinuó la fabricación de este sistema embebido.

(b) Tiene como sistema operativo Android 6, lo que en un principio puede no parecer un inconveniente, limita enormemente el control sobre el sistema y sus recursos por cuestiones de seguridad e impone su propia cadena de herramientas con la que desarrollar para la plataforma.

2. *Raspberry Pi 1 model B*[11]

Aspectos positivos:

(a) Gran cantidad de documentación y buen soporte al cliente. Debido a la gran popularidad de su serie de sistemas empotrados, existen cientos de recursos bibliográficos, tutoriales y discusiones en la web que ayudan enormemente.

(b) Libertad para instalar el sistema operativo que se desee (y que funcione) y configurarlo para disponer de casi el cien por cien de su potencial computacional.

(c) Bajo precio.

(d) El director del proyecto ya dispone de un ejemplar, por lo que solo sería necesario adquirir otra para el alumno.

Aspectos negativos:

(a) Recursos técnicos algo limitados.

(b) Poca disponibilidad del producto.

3. *Raspberry Pi 3 model B*[12]

Aspectos positivos:

(a) Al igual que su predecesor, existe bastante información en la web sobre este sistema empotrado y ofrece gran libertad para configuarlo.

(b) Mayor margen de recursos técnicos.

Aspectos negativos:

(a) Muy escasa disponibilidad del producto.

(b) Coste económico desorbitado debido a su enorme demanda y a las pocas unidades disponibles en el mercado.

(c) Es necesario conseguir al menos dos ejemplares para el director y alumno.

4. *Rock Pi 4*[13]

Aspectos positivos:

(a) Mayor margen de recursos técnicos.

(b) Buena disponibilidad del producto.

(c) Reducido precio.

Aspectos negativos:

(a) Es necesario conseguir al menos dos ejemplares para el director y alumno.

(b) No existe mucha documentación sobre el sistema empotrado y el soporte al cliente es deficiente.

(c) Bastantes fallos hardware y software.

Elección: En un principio, debido a que la universidad ya disponía de dos sistemas *Dragon board 810 Development Kit* para el directo y alumno, se optó por usarlas como sistema empotrado para ejecutar las pruebas del motor gráfico y hacer profiling de este a pesar de sus inconvenientes, mientras no se disponía de otra opción.

Más tarde, debido a complicaciones técnicas que surgieron durante la instalación del sistema operativo escogido, fueron descartadas y se pasó a buscar de forma más activa unas *Raspberry Pi* por sus puntos positivos y el soporte garantizado del sistema operativo y las herramientas cadena de herramientas. También se pensó en la *Rock Pi 4*, pero los inconvenientes pesaron más que su buen precio. Poco después, se consiguió obtener de segunda mano una *Raspberry Pi 1 model B* aprovechando que el director ya tiene la suya, por lo que esta se convirtió en el nuevo candidato de sistema empotrado a usar, asumiendo un mayor riesgo de fallo al desconocer sus usos previos a la adquisición. Finalmente, aparecieron varias *Raspberry Pi 3 model B* sin usar de la universidad, pero debido al coste en tiempo de instalar todas las herramientas en ellas, se decidió seguir usando la primera, y si sobrase algo de tiempo al terminar el motor gráfico y la demostración técnica de este, podrían servir para probar estos en una plataforma más.

Lenguaje de programación

Requisitos:

1. Muy rápido y eficiente.
2. Máximo control de los recursos computacionales.
3. Con las bibliotecas librerías que permitan desarrollar motores gráficos.
4. Que permita generar código fácilmente escalable.
5. Con compilador y enlazador o interprete, y depurador para el sistema embebido y muchas otras plataformas.

Riesgos:

1. Falta de tiempo para aprender a programar en el lenguaje.
2. No ofrezca el rendimiento esperado.
3. Ofusque y complique en exceso la implementación del motor.
4. Problemas para generar el programa en alguna de las plataformas deseadas.
5. Se encuentren fallos que no se lleguen a comprender.

Opciones:

1. El lenguaje C

   Aspectos positivos:

   (a) Conocido por su excelente rendimiento.
   (b) El más usado en sistemas empotrados y tiempo real.
   (c) Gestión explicita de la memoria.
   (d) Permite deducir e incluso controlar qué producirá el compilador en base a un mismo código.
   (e) Acceso a recursos de muy bajo nivel como registros.
   (f) Tiene bibliotecas estandarizadas para cada subsistema del motor gráfico (OpenGL para los gráficos, OpenAL para le audio, ...).
   (g) Es portable siempre que tenga un compilador para la plataforma y las bibliotecas necesarias.

Aspectos negativos:

(a) Requiere de un gran nivel de dominio del lenguaje para hacer un software tan complejo como el de este proyecto.

(b) Necesita conocimiento avanzados en todos los componentes del sistema operativo.

(c) Produce código potencialmente complejo y difícil de escalabilizar.

(d) Propenso a fallos, a veces muy costosos de depurar.

(e) Lenta creación de prototipos.

(f) Comparte las bibliotecas de C y agrega otras más recientes como Vulkan.

2. El lenguaje C++

Aspectos positivos:

(a) Todos los de C ya que lo soporta natívamente.

(b) Multiparadigma, entre ellos, el orientado objetos que permite un código más escalable y aplicando sus patrones.

(c) La inclusión de plantillas para definir automáticamente funciones y estructuras de datos de forma genérica.

(d) El más usado en el desarrollo de videojuegos.

(e) Es portable siempre que tenga un compilador para la plataforma y las bibliotecas necesarias.

Aspectos negativos:

(a) Al igual que C, es necesario conocer a fondo todas las funcionalidades del lenguaje antes de empezar a desarrollar el motor. Pero en el caso de C++, estas son muchas más que las de C, por lo que formarse en él requiere más tiempo y es sustancialmente más difícil.

(b) También hay que tener una fuerte base de conocimientos sobre los sistemas operativos para usarlo.

(c) Propenso a fallos, a veces muy costosos de depurar.

(d) En ciertos aspectos no es tan eficiente como C, pero por lo general hace código más fácil de comprender.

(e) Lenta creación de prototipos.

3. El lenguaje Python

Aspectos positivos:

(a) Es uno de los lenguajes más usados en general.

(b) Promovido por Raspberry Pi para usarse en sus sistemas.

(c) Facilidad de Aprendizaje. Python es conocido por su sintaxis legible y fácil de aprender.

(d) Es un lenguaje de alto nivel que permite a los desarrolladores escribir código más rápidamente que los lenguajes de bajo nivel como C++ o C.

(e) tiene una gran comunidad de desarrolladores y una amplia variedad de bibliotecas y herramientas disponibles como Pygame.

(f) Es un lenguaje interpretado, por lo que es sencillo de depurar.

(g) Provee unas pautas para hacer que el código se entienda mientras se lee de seguido.

(h) Existen varias implementaciones del interprete para múltiples plataformas.

(i) Multiparadigma, con Duck typing.

Aspectos negativos:

(a) Al ser un lenguaje interpretado, resulta en un rendimiento inferior en comparación con lenguajes compilados.

(b) Tiende a utilizar bastante memoria.

(c) No ofrece el mismo nivel de control de hardware.

(d) No se suele usar al desarrollar videojuegos en plataformas de recursos limitados.

Elección: Inicialmente se pensaba usar únicamente el lenguaje C para el desarrollo do proyecto. Tanto por las ventajas ya mencionadas, como también por la mayor experiencia registrada con este lenguaje por parte del alumno, tanto en ordenadores como sistemas empotrados.

Más tarde, al empezar a diseñar el primer prototipo con este lenguaje en mente, se observó que se estaban aplicando técnicas de diseño que encajaban muy bien con los principales paradigmas de la orientación de objetos, principalmente el polimorfismo de las clases. Esto, sumado a los consejos del director y la necesidad de una solución que facilite hacer extensiones, hizo que finalmente se optara por usar, principalmente, el lenguaje C++ para el desarrollo, pero de una forma particular y clásica, tomando bastantes restricciones que simplifiquen su uso a la vez que posibiliten la compatibilidad con más plataformas. Estas restricciones consisten en usar C++ de una forma muy similar al C estándar, sin sobrecarga de operadores (resulta confuso viniendo de C), sin espacios de nombres (por consistencia con las partes del motor

en C), sin excepciones (cuando se capturan, afecta notablemente al rendimiento, además, no siempre están soportadas por la plataforma), sin RTTI (su principal utilidad es permitir hacer Down Casting, un proceso lento e indicador de malas prácticas, y existen plataformas que no lo disponen), y, más importante, sin incluir ni enlazar la biblioteca estándar de C++, esto último puede parecer desorbitado ya que ofrece una aproximación orientada a objetos de la librería estándar de C y muchas más funcionalidades, pero entender su funcionamiento costaría demasiado tiempo, además de suponer una dependencia menos para el motor gráfico, algo muy importante para facilitar su portabilidad.

También se pensó en la posibilidad de usar el lenguaje Python, pero debido a su intrínseco inferior rendimiento, y que ya fue usado anteriormente por el alumno en la asignatura *Contornos Inmersivos, Interactivos y de Entretenimiento* para hacer un videojuego usando un prototipo de este motor gráfico, sería interesante probar a implementarlo con otro lenguaje en esta ocasión.

Bibliotecas para la abstracción del acceso al hardware

Requisitos:

1. Sencilla pero potente.
2. Renderizado 2D de imagenes *bitmap*.
3. Recepción de eventos de varios periféricos como teclado, ratón, y mando de consola.
4. Gestión de la reproducción de audio.
5. Manejo de la entrada y salida de ficheros.
6. Creación de nuevos hilos de procesamiento.
7. Control del tiempo durante la ejecución.
8. Abstracción e independencia de la plataforma.
9. Disponibilidad en distintas plataformas.
10. Compatible con el lenguaje escogido.

Riesgos:

1. Largo periodo de formación sobre las utilidades y usos de su API.
2. Dificultades técnicas durante su uso.

3. Resulte muy limitante o no satisfaga completamente todas las necesidades.
4. Errores en su diseño e implementación.
5. No funcione correctamente en alguna de las plataformas.

Opciones:

1. *SDL(Simple DirectMedia Layer).*

   Aspectos positivos:

   (a) Código abierto.

   (b) Larga trayectoria de desarrollo (desde 1998[14]).

   (c) Ampliamente utilizado en la industria, con más de 700 juegos, 180 aplicaciones y 120 demos[14]. Entre ellos se encuentran reconocidos títulos como el videojuego *Deponia*[15], el *port* de Super Mario 64 a PC[16], y el motor gráfico de Half Life 1[17].

   (d) Facilita la creación de aplicaciones que se ejecuten en una variedad de sistemas operativos y plataformas.

   (e) Proporciona una abstracción de bajo nivel para el manejo de gráficos, sonido y entrada de eventos.

   (f) Gran comunidad activa de desarrolladores y una amplia documentación.

   (g) Tiene sub-bibliotecas que extienden aspectos como la apertura de más formatos de imágenes, mejor manejo del audio, comunicación TCP y UDP, y renderizado de fuentes.

   (h) Permite el uso complementario de otras bibliotecas gráficas como OpenGL, Vulkan, Metal, y Direct3D si se desea generar gráficos más complejos.

   Aspectos negativos:

   (a) Su API es enorme, y para sacarle el mayor partido es necesario estar familiarizado con la biblioteca.

   (b) Resulta muy lento y complejo hacer un primer prototipo.

   (c) Aunque SDL es eficiente en términos de abstracción y portabilidad, no lo es tanto en términos de rendimiento bruto como algunas otras bibliotecas o enfoques más específicos.

(d) Al simplificar muchas operaciones comunes de gráficos y entrada de eventos, se pierde control del bajo nivel.

2. *SFML(Simple Fast Multimedia Library)*.

Aspectos positivos:

(a) Código abierto.

(b) API orientada a objetos, más intuitiva y fácilmente escalable.

(c) En comparación con bibliotecas más antiguas, como SDL y Allegro, la base de usuarios de SFML es relativamente pequeña, pero va en aumento.

(d) Tiene varios módulos para el uso de vectores, cadenas de caracteres de Unicode, creación de hilos, gestión del tiempo, creación de ventanas, manejo de eventos, renderizado de gráficos 2D y texto, reproducción de audio, y conexión TCP y UDP.

(e) También permite usarse junto a OpenGL.

(f) Compatible con una variedad de sistemas operativos y plataformas.

Aspectos negativos:

(a) Para proyectos más complejos, la configuración inicial de SFML puede ser un poco complicada.

(b) A pesar de ser compatible con multitud de plataformas, no tiene soporte directo con ciertos sistemas embebidos como las Raspberry Pi.

(c) No ofrece todo el control de bajo nivel que se pueda necesitar en búsqueda de abstraer la plataforma al desarrollador/a.

3. *Allegro(Atari Low Level Game Routines)*.

Aspectos positivos:

(a) Código abierto.

(b) La biblioteca más antigua de todas.

(c) Concebida para plataformas con notables limitaciones técnicas.

(d) Ofrece una abstracción de la gestión de gráficos, entrada de eventos, sonido y otros recursos multimedia.

(e) A pesar de originalmente diseñarse para sistemas Windows y DOS, ahora es multiplataforma.

Aspectos negativos:

(a) A pesar de ser una biblioteca más longeva, Allegro no es tan ampliamente utilizada ni conocida como las otras dos ya comentadas.

(b) Su documentación es incompleta o desactualizada.

(c) Aunque Allegro es multiplataforma, algunas funciones pueden no estar disponibles o pueden comportarse de manera diferente dependiendo del sistema.

Elección: Se escogió definitivamente usar la biblioteca SDL por las siguientes razones:

1. Es la que ofrece un mayor número de plataformas en las que se encuentra disponible de forma probada, entre ellas, el ordenador y sistema empotrado usados.
2. La que más funcionalidades tiene de forma abstracta.
3. La que menos dependencias con otras bibliotecas tiene y más opciones tiene a falta de una.
4. Permite crear un motor que prácticamente solo dependa de esta biblioteca, al implementar la mayoría de la biblioteca estándar de C en caso de no existir una nativa, automatizando casi por completo el proceso de hacer port de los videojuegos desarrollados con el motor gráfico.
5. Se estudió su API durante más de cuatro años haciendo pequeños prototipos, siendo la biblioteca con mayor dominio por parte del estudiante.

A mayores, se usó de forma auxiliar la biblioteca SGLIB[18] para usar sus listas enlazadas genéricas, y *Simple SDL2 Audio*[19] para simplificar la gestión del audio de SDL.

3.2 Primera iteración: Apertura y dibujado de la ventana

En esta iteración se describe el trabajo hecho para construir un prototipo en el que se abra una nueva ventana y varios objetos dibujen en ella figuras geométricas.

3.2.1 Diseño

El primer paso fue diseñar la clase *ESGE_Display*, cuya representación UML se muestra en la Figura 3.1. Esta clase, que será el *display* o pantalla en el motor gráfico, representa la ventana, con su renderizador al que dibujar, y la cámara virtual, que se podrá mover.

Cada instancia crea una nueva ventana, por lo que es necesario indicar el nombre y resolución de la misma. Esto permite el uso de múltiples ventanas para el mismo videojuego,

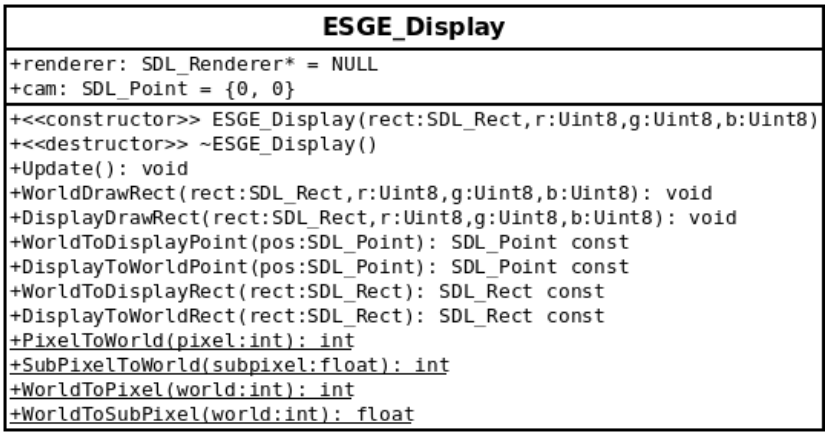

Figura 3.1: Diagrama UML de la clase ESGE_Display.

posibilitando presentar distintos puntos de vista de una escena, por ejemplo. Cada ventana permite dibujar sobre ella rectángulos, relativos o no a la posición de la cámara virtual, mediante un par de métodos. Finalmente, para presentar todos estos cambios dibujados en la pantalla, y limpiarla para volver a poder ser dibujada, se añadió un método de actualizado, que debe ser llamado en cada iteración del bucle principal.

Además, también se añadieron unos métodos para trasladar un punto del sistema de coordenadas de la pantalla al del mundo virtual y viceversa. Para concluir esta clase, a mayores se pensó que podría ser útil hacer que el sistema de coordenadas tenga una mayor escala que el de la ventana para producir movimientos de objetos más fluidos y calcular sus colisiones de forma más precisa. Para esto, se añadieron varios métodos que permiten calcular la conversión del valor de la distancia en píxeles y sub-píxeles a la correspondiente del mundo y a la inversa.

Una vez ya diseñada la pantalla, es necesario crear un mecanismo para que los objetos puedan dibujar en ella las figuras de una forma sencilla y rápida. Adicionalmente, es imprescindible dar alguna forma de organizar el orden en el que se presentan sus figuras en la pantalla. Para ello se definió la clase abstracta *ESGE_ObjDraw* (Figura 3.2) siguiendo el patrón plantilla, el patrón observador, y el patrón del método de actualizado [20] basado en la solución de Game Maker Studio [21]. Esta clase contiene una lista con todas las instancias que la deriven y desean ser notificadas, la siguiente instancia en la lista, y la capa en la que se dibujará (Más alto el valor, más por encima estará). La capa debe ser especificada desde la clase derivada. Además, como el nombre del patrón indica, la clase servirá de plantilla para todos aquellos objetos del juego que requieran ser dibujados en pantalla, heredándola y definiendo un nuevo comportamiento en su método abstracto de dibujado, al que se notificará según su capa cuando se necesite dibujar todas las instancias a través de su método estático. También es posible activar y desactivar la notificación del método de dibujado y preguntar sobre cuál

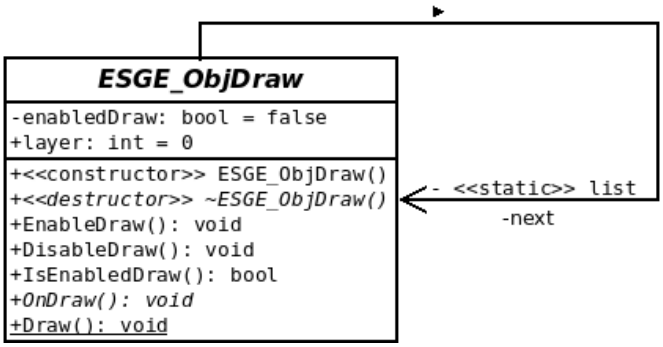

Figura 3.2: Diagrama UML de la clase ESGE_ObjDraw.

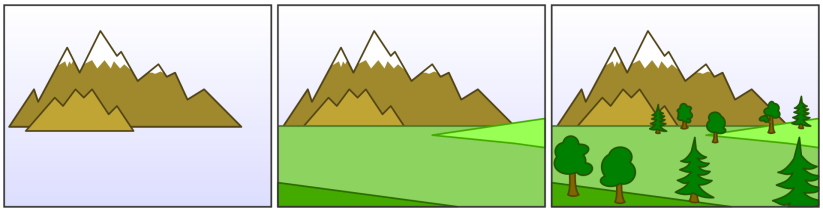

Figura 3.3: Ejemplo del proceso de dibujado usando el algoritmo del pintor.

es su estado actual mediante los tres métodos correspondientes.

Esta aproximación de dibujado por capas está de nuevo basada en la que usa Game Maker Studio [21], que hace uso del algoritmo del pintor [22], una simple pero efectiva (en general) técnica de computación gráfica para la determinación de superficie visible a renderizar. La técnica consiste, básicamente, en ordenar las figuras geométricas según su distancia respecto a la cámara, para dibujarlas en secuencia, comenzando por las más lejanas y terminando por las más próximas. De esta forma, como se puede ver en la imagen de la Figura 3.3, la figura que se encuentre más cerca se superpondrá ante las más distante, garantizando una escena con profundidad consistente (salvo en algunos casos problemáticos).

Sus ventajas son las siguientes:

1. **Implementación sencilla:** Solo se requiere de una lista, donde almacenar las figuras a dibujar, y de un algoritmo de ordenación rápido para ordenarla según su profundidad.

2. **Eficiencia en espacio:** Al no ser necesario guardar ninguna información sobre la profundidad de las figuras (como en técnicas basadas en un búfer de profundidad [23]), basta con el framebuffer para representar la imagen final.

Sus desventajas son las siguientes:

1. **Solapamiento cíclico:** Como se puede ver en la Figura 3.4, los polígonos A, B y C se superponen de tal manera que es imposible determinar cuál de ellos está encima de los

demás. En este caso, los polígonos problemáticos deben ser recortados para permitir la ordenación.

2. **Polígonos atravesadores:** Cuando un polígono asoma a través de la superficie de otro, el primero quedaría oculto por el segundo a pesar de intersecarlo, por lo que es necesario recortarlos para resolver este caso.

3. **Eficiencia en tiempo:** Al dibujar figuras que van a quedar ocultas por otras más cercanas, se esta desperdiciando una gran cantidad de tiempo en algo que no se va a representar en la imagen final.

Sin embargo, dado al contexto de videojuegos 2D que concierne al motor gráfico desarrollado, las dos primeras desventajas no le afectarían en absoluto, ya que se trabaja exclusivamente con figuras e imágenes 2D sin ningún tipo de profundidad geométrica. La única forma de representar esta profundidad es dibujando antes o después la figura en la pantalla, justamente el principio de funcionamiento de esta técnica.

La última desventaja será un coste que se deberá asumir a favor de la memoria, ya que suele ser este último el recurso más limitante en los sistemas empotrados.

3.2.2 Análisis de riesgos

Algunos de los riesgos que conllevan las decisiones de diseño que se tomaron respecto a la apertura y dibujado de la ventana se presentan a continuación:

1. Depositar la creación y gestión de ventanas al desarrollador/a puede resultar complejo y causarle problemas.

2. Manejar varias ventanas dificulta la futura gestión de eventos y el renderizado de los objetos.

3. No se ofrezca las suficientes figuras para dibujar que le puedan ser necesarias al desarrollador/a.

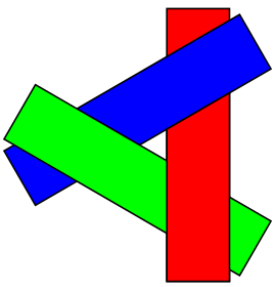

Figura 3.4: Caso problemático para el algoritmo del pintor.

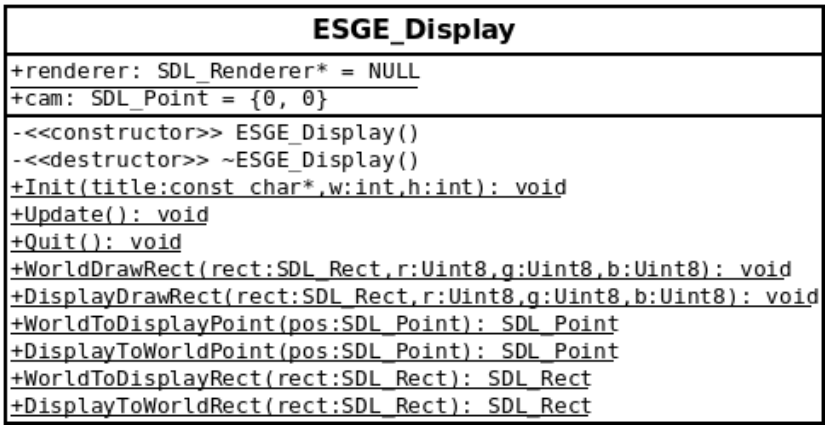

Figura 3.5: Diagrama UML de la clase ESGE_Display versión 2.

4. Escalar las coordenadas del mundo virtual respecto a las de la pantalla puede ser confuso para el desarrollador/a y complicar la comprobación de colisión.

5. El uso de listas enlazadas y métodos virtuales puede no ser la solución más rápida y eficiente.

3.2.3 Re-diseño y desarrollo

Para mitigar los riesgos encontrados en el diseño previo de la pantalla se pensó en realizar los siguientes cambios en la clase *ESGE_Display*, mostrados en la Figura 3.5:

1. Mantener una única ventana e instancia siguiendo el patrón *singleton* que facilite el acceso de sus métodos de dibujo y atributos.

2. Al limitarse a una única ventana por juego, puede ser el propio motor el encargado de abrirla, actualizarla y cerrarla.

3. Respecto a las figuras disponibles para dibujarse, la clase podría extenderse en un futuro añadiendo más métodos para dibujar más tipos de figuras si es necesario.

4. Eliminar el escalado de las coordenadas del mundo virtual respecto a las de la pantalla, facilitando la localización de las figuras que se dibujen y el futuro motor de físicas.

Respecto a la clase *ESGE_ObjDraw*, no se aplicaron cambios por el momento, pero se plateó usar una gestión de la memoria basada en regiones [24] para reducir los fallos cache que conllevan las listas enlazadas.

Después, se comenzó a implementar el prototipo en base al nuevo diseño aprovechando las estructuras y funciones relacionadas con el renderizado proporcionadas la biblioteca SDL

junto con SGLIB para manejar la lista enlazada ordenada. Este prototipo, consistirá en la implementación del bucle principal del juego en el que se dibujarán los objetos y se actualizará la pantalla, las clases *ESGE_Display* y *ESGE_ObjDraw*, y algunas derivadas de la segunda para probar el dibujado sobre la pantalla de los objetos que representan.

Para implementar la inicialización de la pantalla, se uso la nueva API de reder que permite aceleración por hardware de los gráficos y guardar en la memoria de vídeo texturas(necesario para más adelante), acelerando el dibujado de objetos y ahorrando memoria de la principal. Además, esta API también permite renderizar a una resolución virtual independiente a la del dispositivo de destino (se escala la imagen del juego al tamaño de la ventana).

Luego, para gestionar la activación y desactivación de los objetos dibujables (heredan de *ESGE_ObjDraw*), se implemento añadiendo y retirando la instancia de la lista ordenada por sus capas para reducir accesos a memoria innecesarios durante su recorrido en el dibujado de todos las instancias.

Para terminar, después de implementar todas las clases, al final del bucle principal se añadió una medición del tiempo transcurrido durante el dibujado para controlar la tasa de fotogramas por segundo a la que se iterará y ejecutará el juego, esto no siempre es necesario, pero al decidir no usar números flotantes, resulta determinante tener una tasa estable de fotogramas para un funcionamiento consistente de los subsitemas del motor dependientes del tiempo como pueden ser las físicas y animaciones.

3.2.4 Prueba del prototipo

El prototipo demostró funcionar bien en la instalación del ordenador de desarrollo, siendo el siguiente paso comprobar el mismo resultado en el sistema embebido debido al objetivo del motor gráfico de dar soporte a estos aparte de las plataformas habituales.

En este caso, al disponer de una *Raspberry Pi 1 model B* como sistema embebido, se procedió a la instalación de la biblioteca SDL siendo la única dependencia del motor gráfico y su demo. Gracias a disponer de una guía con los pasos a seguir en la instalación de la biblioteca hecha por los propios desarrolladores [25], esta tarea fue un poco más sencilla. Además, en esta se afirma que el juego podría abrir una ventana sin un entorno de escritorio ni gestor de ventanas mediante los subsistemas KMS (Kernel Mode Setting [26]) o alternativamente DRM (Direct Rendering Manager [27]), permitiendo usar una instalación *headless* del sistema operativo Raspbian [28] en la Raspberry Pi usada. De esta manera, esta ofrecerá su máximo potencia técnico para el motor gráfico al no tener que lidiar con un entorno de escritorio. Esta configuración es muy frecuente en las consolas antiguas, ya que su único propósito era el de leer una ROM en donde se encontraba el programa del juego y demás recursos para proceder a la ejecución de este. La pila de gráfica que forma el motor gráfico en un sistema con Kernel de Linux se muestra en la Figura 3.6

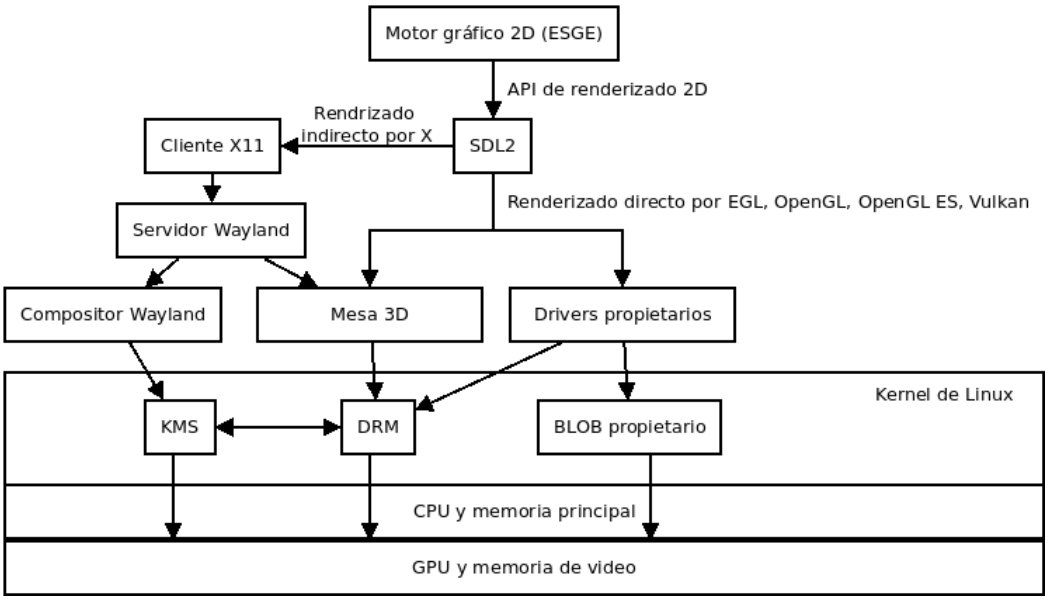

Figura 3.6: Pila gráfica de Linux usando la biblioteca SDL2.

Finalmente, tras un largo proceso de instalación del sistema operativo Raspbian y la biblioteca SDL en la Raspberry Pi, y la depuración de algunos fallos cometidos solo perceptibles en dicha plataforma, se consiguió que el prototipo se ejecutara correctamente sin entorno de escritorio.

3.3 Segunda iteración: Gestión de eventos

El objetivo de esta iteración es diseñar e implementar un mecanismo para notificar a los objetos del juego de los eventos registrados que necesiten en cada iteración del bucle principal.

3.3.1 Diseño

Para conseguir el anterior objetivo, se diseñó la clase abstracta *ESGE_ObjKeyEvent*, representada en UML en la Figura 3.7. Se siguieron los mismos patrones software que para la clase *ESGE_ObjDraw* (Figura 3.2), pero, en este caso, las instancias de las clases derivadas solo serán notificadas cuando se pulse o suelte una tecla en sus correspondientes métodos, justo antes de dibujase para hacer los cambios hechos por estos métodos visibles en el mismo fotograma. Por lo que solo se deberá llamar a sus métodos estáticos cuando se detecten estos eventos para notificar a todas las instancias activas interesadas en recibirlos.

3.3.2 Análisis de riesgos

Al anterior diseño se le encontraron los siguientes problemas:

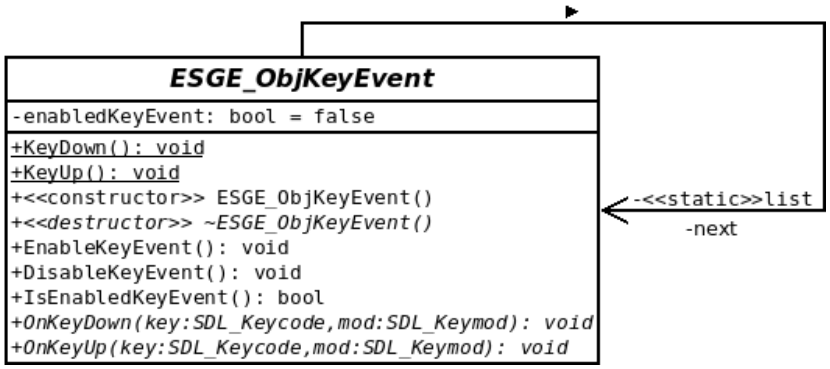

Figura 3.7: Diagrama UML de la clase ESGE_ObjKeyEvent.

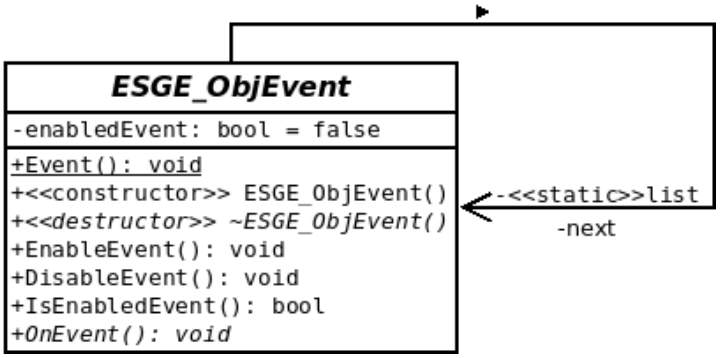

Figura 3.8: Diagrama UML de la clase ESGE_ObjEvent.

1. No se capturan todos los tipos de eventos necesarios para un videojuego como el ratón y mando de consola.

2. Delegar al desarrollador/a la captura de eventos y llamar al correspondiente método estático le requiere de conocimientos innecesarios sobre todos los eventos posibles y puede causar problemas.

3.3.3 Re-diseño y desarrollo

Para solucionar el primer problema de forma temporal se diseño a mayores otra clase abstracta *ESGE_ObjEvent* (Figura 3.8) que contiene un método abstracto al que se notificará cuando se detecte cualquier tipo de evento, será el deber del desarrollador/a filtrar el que le interese y procesarlo. En el futuro se diseñaran más clases como *ESGE_ObjKeyEvent* para procesar eventos de ratón y mando de consola.

El segundo problema se arreglará implementando una única función basada en el patrón del bucle de eventos [29]. Esta se encargará de solicitar los eventos registrados, llamar al método estático de *ESGE_ObjEvent* por cada uno para notificar a las instancias de sus derivadas,

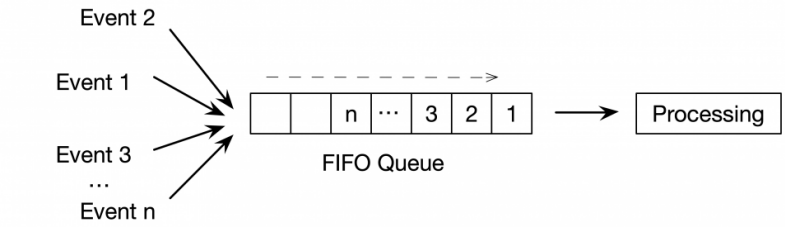

Figura 3.9: Sistema de procesado de eventos en SDL.

y filtrarlos y procesarlos para llamar a los correspondientes métodos estáticos de la clase *ESGE_ObjKeyEvent*.

En la implementación, fue necesario usar la API de eventos de SDL, la cual consiste en una unión enorme de estructuras que albergan información de cada evento y su tipo, y unas funciones para solicitar los eventos ocurridos. Estas funciones siguen el paradigma de la cola de mensajes [30], ejemplificada en la Figura 3.9, debido a que la captura de eventos del sistema operativo se realiza por SDL en un hilo distinto al del bucle principal. Por lo que implementar la función del bucle de eventos haciendo uso de estas resulta tan sencillo como el siguiente pseudo-código:

```
message := poll_next_message()
while message != quit
    message := poll_next_message()
    process_message(message)
end while
```

Finalmente fue necesario añadir dos variables globales, una para almacenar el evento de SDL actual que será actualizada por la función con el bucle de eventos y leída por el método abstracto de la clase *ESGE_ObjEvent* y los métodos estáticos de *ESGE_ObjKeyEvent*. La otra será un *flag* para indicar la solicitación de finalizar el juego mediante el evento de cierre de su ventana. El estado de este deberá ser comprobado por el bucle principal para detener su iteración.

3.3.4 Prueba del prototipo

Con el objetivo de probar las funcionalidades añadidas al motor, se implemento en la demo una clase que hereda de *ESGE_ObjDraw*, *ESGE_ObjKeyEvent* y *ESGE_ObjEvent* e implementa sus métodos abstractos para dibujar un rectángulo en un punto determinado por las teclas que se estén pulsando y el lugar donde se encuentre el cursor del ratón. Luego se incorporó al bucle principal la función del bucle de eventos y la comprobación de cierre justo antes de

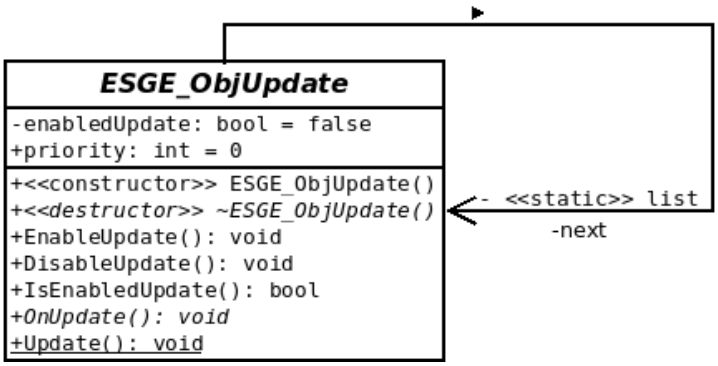

Figura 3.10: Diagrama UML de la clase ESGE_ObjUpdate.

dibujar los objetos del juego, con el fin de recibir los eventos deseados y dibujar en base a ellos.

Al probar el prototipo tanto en el ordenador como en el sistema empotrado, se consiguieron los resultados esperados, e incluso en el segundo apareció el cursor en la pantalla a pesar de no tener el entorno de escritorio.

3.4 Tercera iteración: Actualizado y físicas de objetos

En esta iteración se incorporará al prototipo de motor gráfico el sistema que actualizará el estado del estado de los objetos y otro para sus físicas. A continuación se presentan las distintas etapas de su construcción.

3.4.1 Diseño

Para actualizar los objetos se siguió el mismo diseño que el explicado para la clase *ESGE_ObjDraw*, creando *ESGE_ObjUpdate* (Figura 3.10), clase para la que sus derivadas deben proporcionar una definición concreta del método abstracto para que sus instancias activas sean notificadas por el método estático cuando se requiera actualizar sus estados. La única diferencia es que en este caso, el orden de la lista de instancias activas a actualizar determinará qué objeto se procesa se procesa antes, la que tenga el menor valor del atributo de prioridad será la primera.

Esta prioridad de actualizado de objetos activos da posibilidad al escenario de la Figura 3.11, donde los objetos que dependen del siguiente estado de otro tienen una prioridad inferior de actualizarse respecto a él, compartiendo lugar incluso con otros tipos de objetos. Esto permite agruparlos en grupos de objetos sin interdependencia, pudiéndolos actualizar en un orden arbitrario dentro de estos e incluso hacerlo en paralelo con distintos hilos de procesamiento, aprovechando el potencial de las arquitecturas de procesadores multi-núcleo

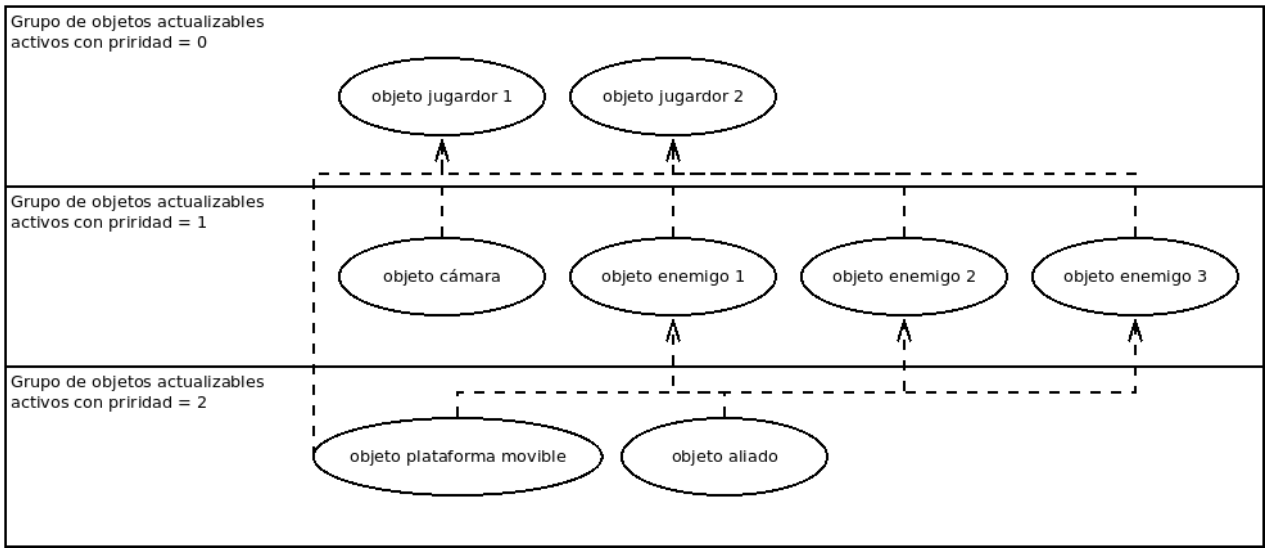

Figura 3.11: Ejemplo de agrupamiento de objetos actualizables activos según su prioridad.

y reduciendo fallos caché.

Más tarde se diseñaron todos los componentes necesarios para el motor de físicas visibles en la Figura 3.12. Estos componentes son estructuras y clases abstractas siguiendo el patrón plantilla que nos permite incorporar funcionalidades automáticamente a las derivadas. Estas se describirán a continuación:

1. ***ESGE_ObjPoint*:** Representa un objeto situado en un espacio 2D(p.e. el mundo virtual).

2. ***ESGE_ObjCell*:** Similar a la anterior pero restringiendo la posición del objeto a una cuadricula.

3. ***ESGE_ObjBox*:** Representa un objeto que tiene un área rectangular al rededor de su punto cuyas aristas están alineadas con los ejes del sistema de coordenadas 2D, también abreviado con el acrónimo AABB(Axis Aligned Bounding Box). Su tamaño y distancia respecto al punto vienen dados pos su atributo.

4. ***ESGE_ObjCollider*:** Representa un objeto que puede colisionar en todas direcciones con otra instancia de una clase derivada, la colisión ocurrirá entre dos objetos cuando sus áreas se solapen, y cada uno deberá llamar al método correspondiente del otro para notificarle de esta.

5. ***ESGE_ObjPhysic*:** Representa un objeto (sin colisión) que puede moverse definiendo su método abstracto que será notificado por el estático cuando se requiera actualizar las físicas.

6. ***ESGE_ObjStatic*:** Representa un objeto estático que no se puede mover (sin físicas) con posición y tamaño sujeto a una cuadrícula, el cual únicamente podrá chocar contra un objeto que se mueva(con físicas) y tenga colisión. Contiene una matriz 2D de listas a la que se añadirán las instancias activas de las clases derivadas para acelerar el proceso de detección colisión usando los métodos estáticos indicados.

7. ***ESGE_ListV*:** Representa una columna en la matriz de objetos estáticos.

8. ***ESGE_ListH*:** Representa una fila en la matriz de objetos estáticos.

9. ***ESGE_ObjDynamic*:** Representa un objeto dinámico que se puede mover (con físicas) y con colisión, sin embargo este solo podrá chocar contra objetos estáticos debido a que el caso reciproco es bastante complejo de simular y supera los requisitos establecidos. La corrección de la colisión tendrá lugar en la implementación del método abstracto de la clase *ESGE_ObjPhysic*, tomando la resta de la posición previa guardada en su atributo con la actual como la velocidad actual de la colisión. Un ejemplo de las distintas aproximaciones posibles para hacer esta corrección se muestra en la Figura 3.13.

3.4.2 Análisis de riesgos

Respecto al diseño de los objetos actualizables, se concluyo que podría resultar muy complejo al desarrollador/a del juego manejar explícitamente el orden de actualizado de los objetos en base a sus dependencias. Además, también sería bastante difícil de implementar el sistema para controlar los distintos hilos de procesamiento de grupos con objetos independientes entre sí, aparte de poder ser potencialmente ineficiente en sistemas empotrados con procesadores de un solo núcleo como es el caso de la *Raspberry Pi 1 Model B*.

Sobre el motor de físicas, podría se ineficiente en temas de fragmentación de la memoria y fallos caché el uso de una matriz 2D basada en listas enlazadas, pero, de nuevo es un problema que se podría solucionar haciendo uso de una gestión de la memoria basada en regiones [24].

3.4.3 Re-diseño y desarrollo

Antes de empezar a implementar las clases y estructuras diseñadas, se retiró el atributo de prioridad de *ESGE_ObjUpdate* (resultando la nueva versión de la clase mostrada en la Figura 3.14) con el fin de solucionar la posible complejidad que conllevaría tanto implementarlo como usarlo.

Algo que destacar del desarrollo fue la costosa implementación de la matriz 2D de objetos estáticos manejando listas enlazadas, y es que gracias a ellas esta matriz no tiene limite de tamaño e irá creciendo cuando se activen los objetos que contendrá, pudiendo un mismo

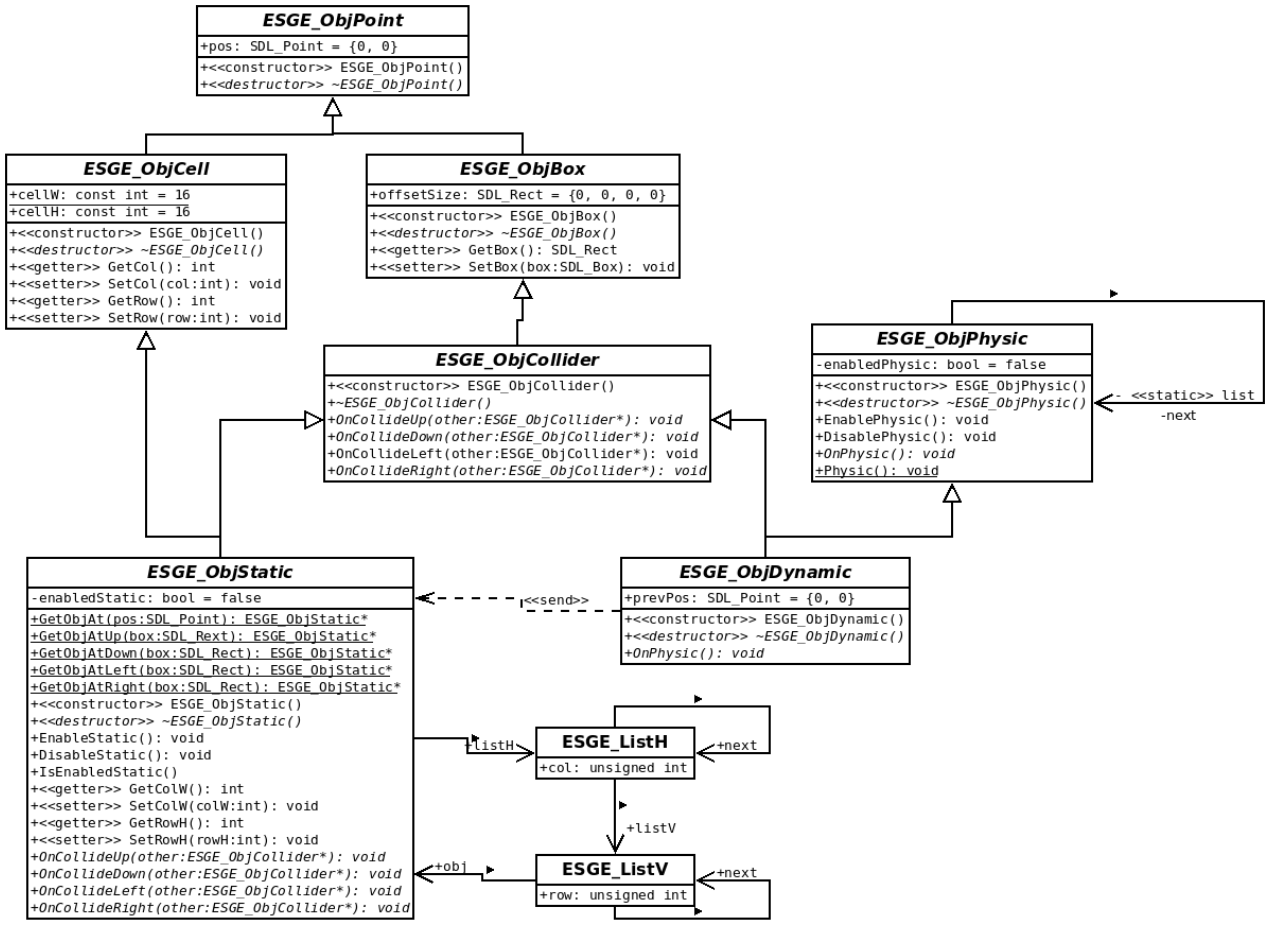

Figura 3.12: Diagrama UML de las clases que componen el motor de físicas.

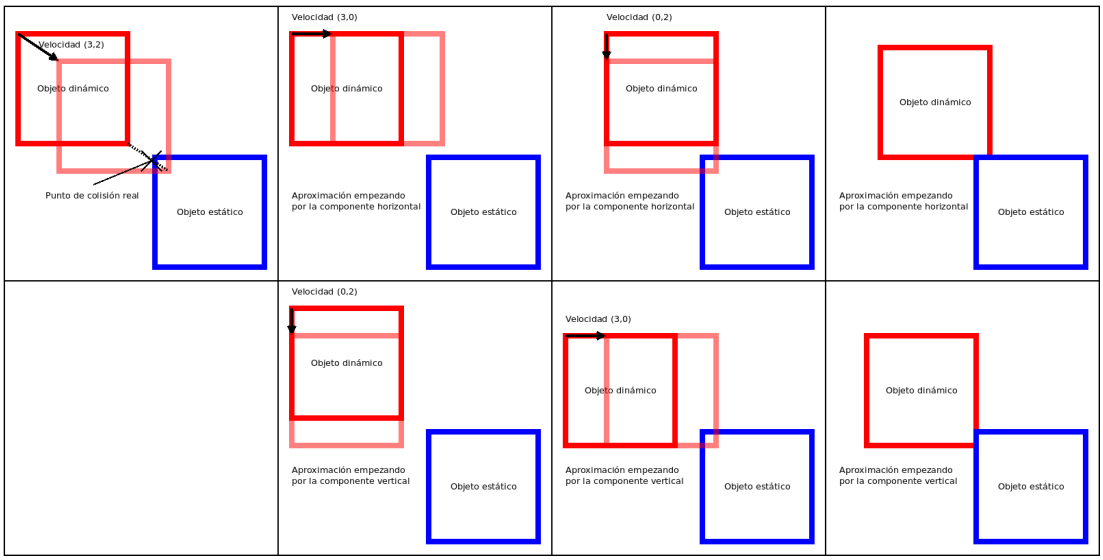

Figura 3.13: Distintas aproximaciones usadas para comprobación de colisiones usando AABB.

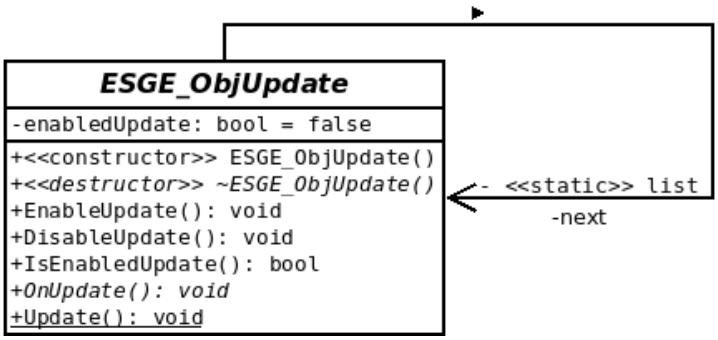

Figura 3.14: Diagrama UML de la clase ESGE_ObjUpdate versión 2.

objeto cubrir múltiples celdas adyacentes. Además, para comprobar la colisión de un objeto con esta matriz, será necesario seguir el proceso descrito mediante la Figura 3.15.

3.4.4 Prueba del prototipo

Para probar el sistema de actualización de objetos y el motor de físicas se implemento en la demo técnica dos nuevas clases. Una que representará el objeto protagonista de un posible juego que controlará el jugador/a, este deberá heredar de *ESGE_ObjKeyEvent* para detectar la pulsación de las teclas que controlarán su movimiento, de *ESGE_ObjUpdate* para efectuar el movimiento y darle gravedad, de *ESGE_ObjDynamic* para la comprobación de colisiones con objetos estáticos, y de *ESGE_ObjDraw* para dibujar el rectángulo de colisión. La otra representará una superficie colisionable en la que el objeto protagonista chocará, por eso esta hereda de *ESGE_ObjStatic* además de *ESGE_ObjDraw* para hacerla visible a través del rectángulo de colisión.

Finalmente, se añadió al bucle principal la actualización del estado y físicas de los objetos activos. Un pseudo-código que muestra de forma abstracta la secuencia actual de los procesos aplicados sobre los objetos del juego correspondientes puede observarse a continuación.

```
1 while not quit_game
2   event_loop()
3   update_objs_state()
4   update_objs_physics()
5   draw_objs()
6 end while
```

3.5 Cuarta iteración: Creación de escenas y su editor

En esta iteración se diseñará, implementará y probará, la clase que permitirá cargar y activar escenas, y el sistema de persistencia y activación de sus objetos para ser cargados y

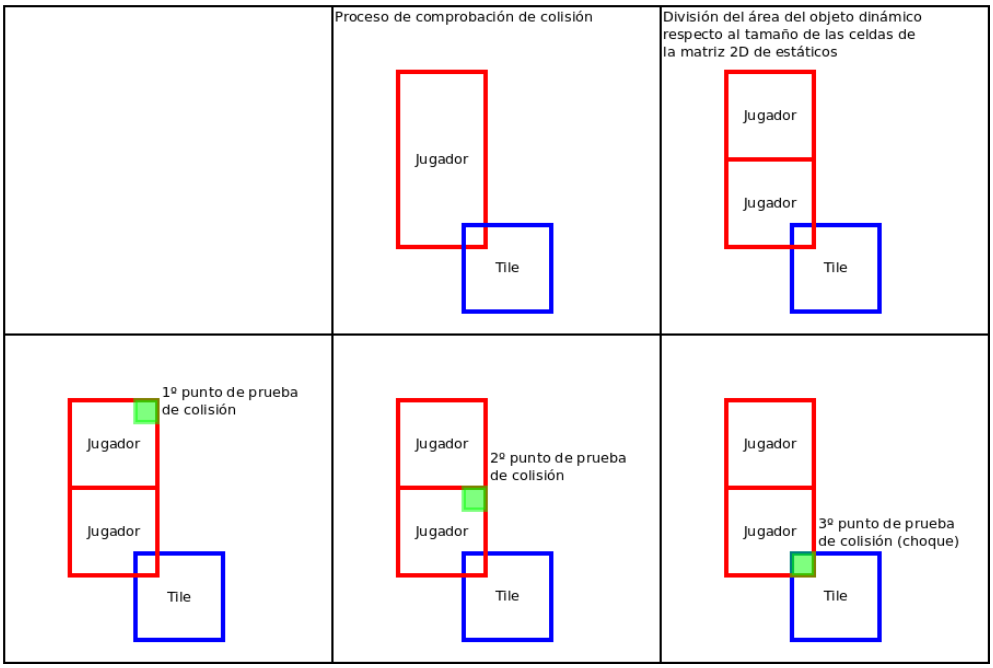

Figura 3.15: Proceso de comprobación de colisión del objeto dinámico con el estático usando la matriz 2D en la que el segundo se encuentra.

activados por estas.

3.5.1 Diseño

Se empezó por diseñar el sistema ilustrado en la Figura 3.16 que guardará los atributos persistentes de los objetos en la escena con el fin de poder recuperar sus valores cuando esta se cargue. Este sistema es formado por los siguientes componentes:

1. ***ESGE_FieldValue*:** Plantilla para almacenar las dos funciones para obtener y editar el valor de un atributo persistente según su tipo, pudiendo ser un entero de 8, 16, 32, 64 bits, número flotante, o una cadena de caracteres.

2. ***ESGE_Field*:** Estructura que almacena el nombre y las dos funciones para obtener y editar el valor de un atributo persistente de un objeto.

3. ***ESGE_Type*:** Clase abstracta donde las instancias de sus derivadas representan un tipo de objeto (clase) que puede contener atributos persistentes. Además, tiene un método abstracto que al sobrescribirlo permite crear una nueva instancia dinámicamente (en tiempo de ejecución) de una clase desconocida en tiempo de compilación (estáticamente). Esto será crucial para la creación del editor de escenas ya que sería muy tedioso que este conociera todos los tipos posibles del juego.

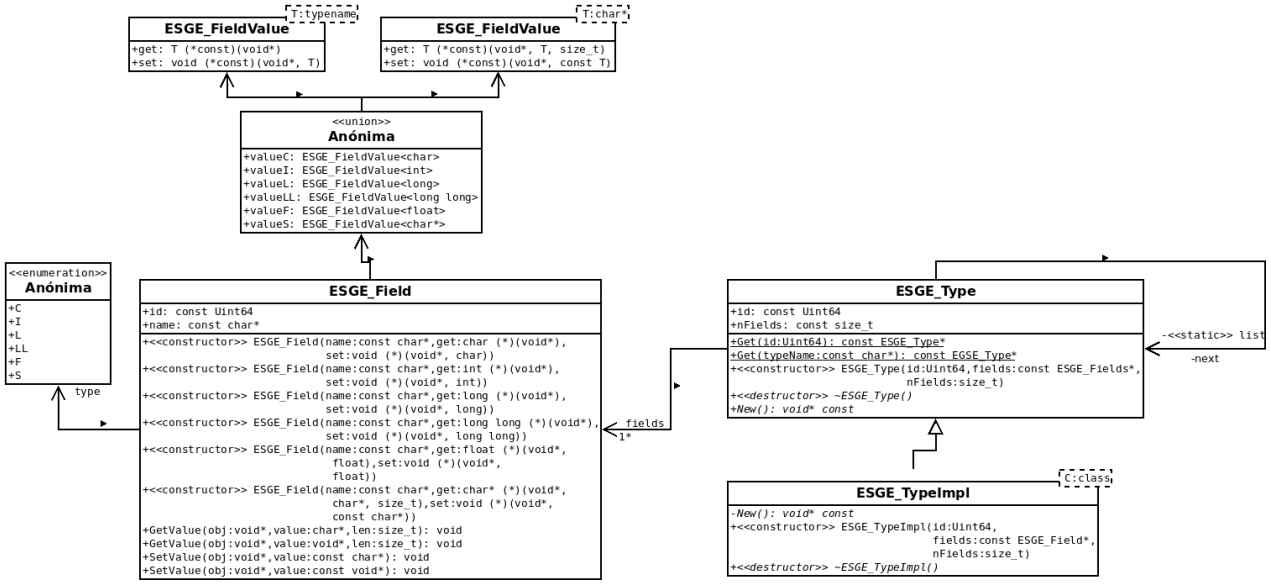

Figura 3.16: Diagrama UML de las clases *ESGE_Field* y *ESGE_Type*, y plantillas *ESGE_FieldValue* y *ESGE_TypeImpl*.

4. ***ESGE_TypeImpl*:** Plantilla para definir la clase derivada de *ESGE_Type* donde su única instancia (*singleton*) representa el tipo del objeto del juego definido por el desarrollador/a.

Una vez ya diseñado el sistema de persistencia de objetos, el siguiente paso es diseñar la clase abstracta *ESGE_ObjScene* de la Figura 3.17 que represente a todos los objetos que se encuentren dentro de una escena y por lo tanto dentro también del juego. Estos podrán ser instanciados dinámicamente y añadidos a la escena actual mediante el método estático *Create* al que se le debe entregar el identificador del tipo de objeto, destruidos y eliminados de esta con *Destroy*, activados y desactivados con los métodos abstractos correspondientes de la clase abstracta *ESGE_ObjActive*, y cargar y guardar su atributos persistentes mediante los métodos correspondientes de la clase abstracta *ESGE_ObjSerial*. Adicionalmente, *ESGE_ObjScene* proporciona los siguientes métodos abstractos para definir por el desarrollador/a en subclases:

1. ***OnInit*:** Llamado después de la carga del objeto por la escena.
2. ***OnQuit*:** Llamado justo antes de ser eliminado por la escena.
3. ***OnStart*:** Llamado después de activar el objeto por la escena.
4. ***OnEditorEnable*:** Llamado al activar un objeto por la escena cuando esta se encuentra en el editor.

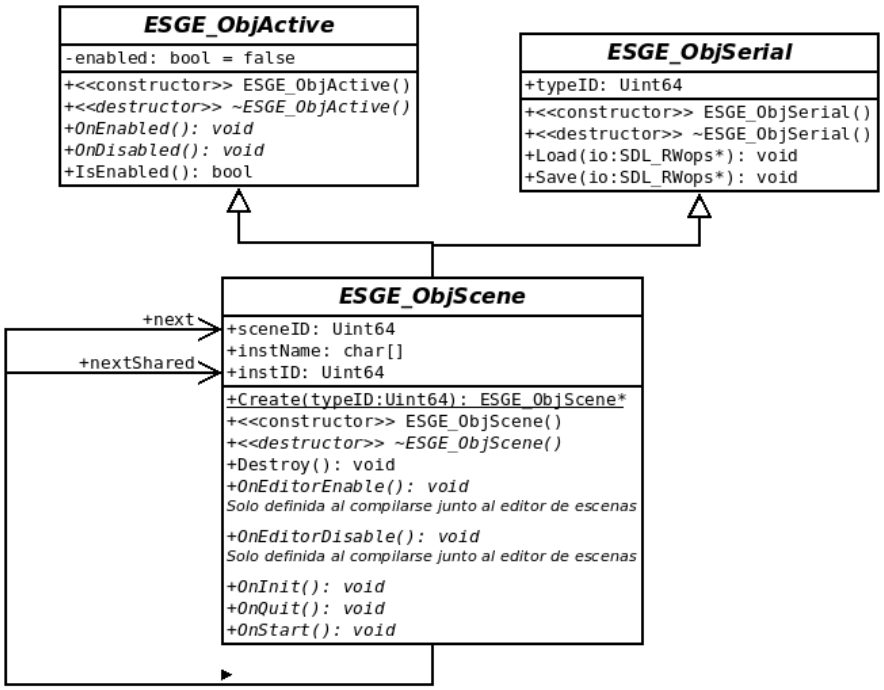

Figura 3.17: Diagrama UML de las clases ESGE_ObjActive, ESGE_ObjSerial y ESGE_ObjScene.

5. ***OnEditorDisable*:** Llamado al desactivar un objeto por la escena cuando esta se encuentra en el editor.

Ya disponiendo del diseño de los objetos en la escena, es momento de pensar cómo va a ser esta, llegando a la conclusión de la clase *ESGE_Scene* ilustrada en la Figura 3.18.

Esta clase será un contenedor de los objetos que se encuentren en ella para tomar sobre ellos las acciones correspondientes de sus métodos. A mayores esta dispone de dos métodos para crear, agregar, destruir, eliminar un objeto (usados por los métodos *Create* y *Destroy* junto al editor de escenas).

3.5.2 Análisis de riesgos

El diseño de las clases *ESGE_ObjSerial* y *ESGE_ObjScene* podría ser problemático para el desarrollador/a al tener que manejar explícitamente los identificadores numéricos de tipo e instancia del objeto.

También, referenciándose en el libro [20], se encontró un potencial problema en la creación, destrucción, activación, y desativación inmediata de los objetos de la escena. Este consiste en que, al realizar alguna de las anteriores acciones durante la actualización de los objetos, modifica la propia secuencia de procesado, pudiendo dejar alguno de estos sin actualizar o incluso salirse de la lista de objetos como se pude ver en el siguiente código C y en la Figu-

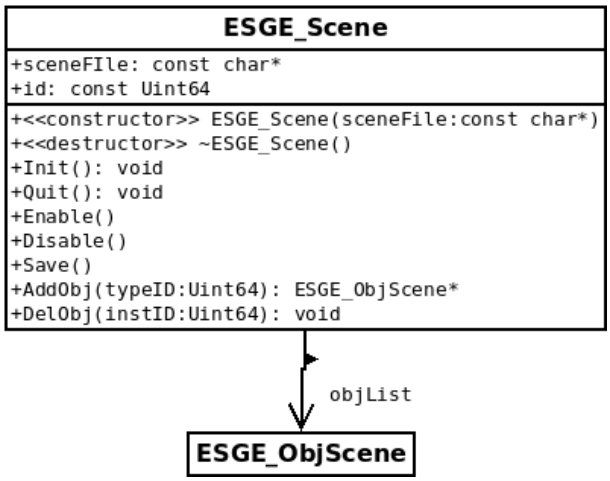

Figura 3.18: Diagrama UML de la clase ESGE_Scene.

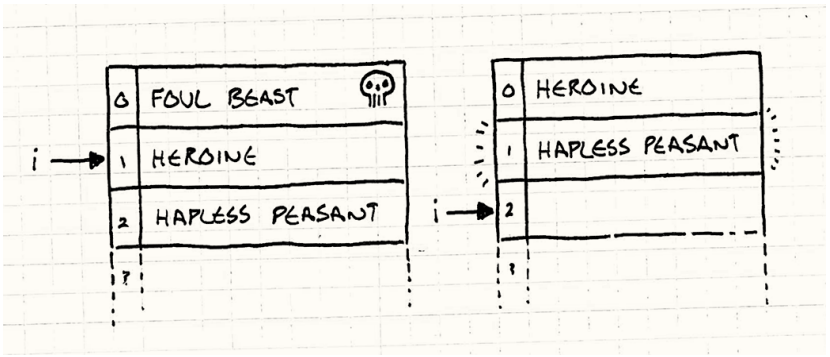

Figura 3.19: Ejemplo problemático de eliminación inmediata de un objeto del juego.

ra 3.19.

```
1  for (int i = 0; i < numObjects_; i++)
2  {
3    objects_[i]->update();
4  }
```

3.5.3 Re-diseño y desarrollo

Para solucionar el primer problema, se decidió sustituir el manejo explicito de identificadores numéricos por los producidos por una función hash a través de una cadena de caracteres más fácil de reconocer. El cambio queda reflejado en Figura 3.20.

Luego se paso a solucionar el otro problema sobre la creación, destrucción, activación, y desativación inmediata de los objetos de la escena y la secuencia de actualizado. Para ello, se modificó el diseño de la clase *ESGE_ObjScene* y *ESGE_ObjScene* de forma que estas acciones

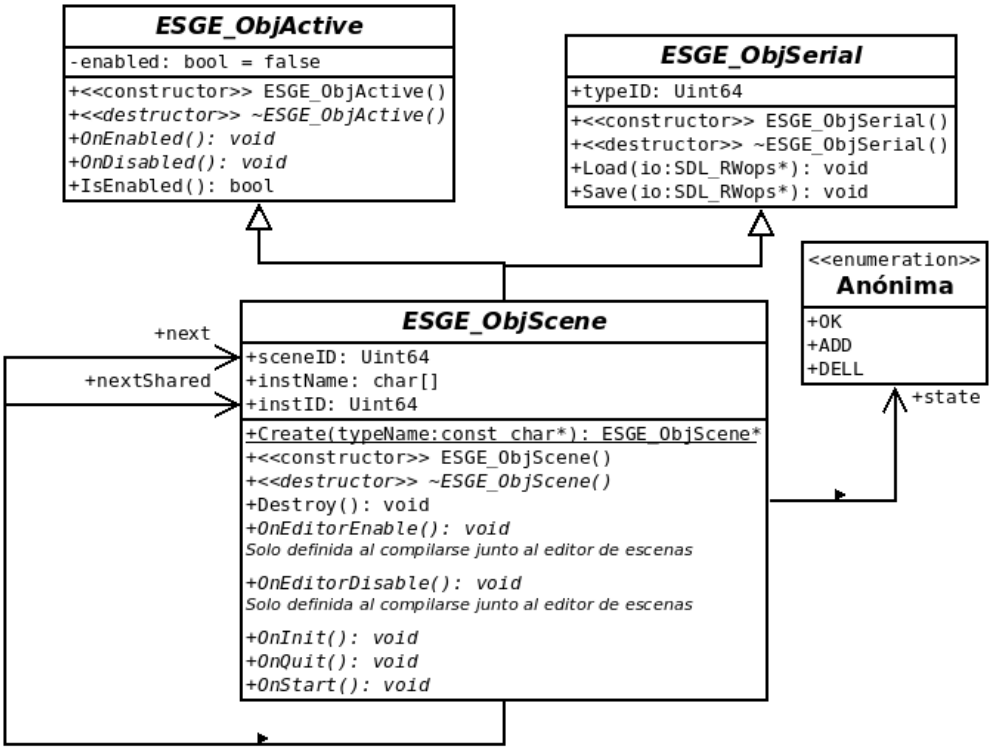

Figura 3.20: Diagrama UML de las clases ESGE_ObjActive, ESGE_ObjSerial y ESGE_ObjScene versión 2.

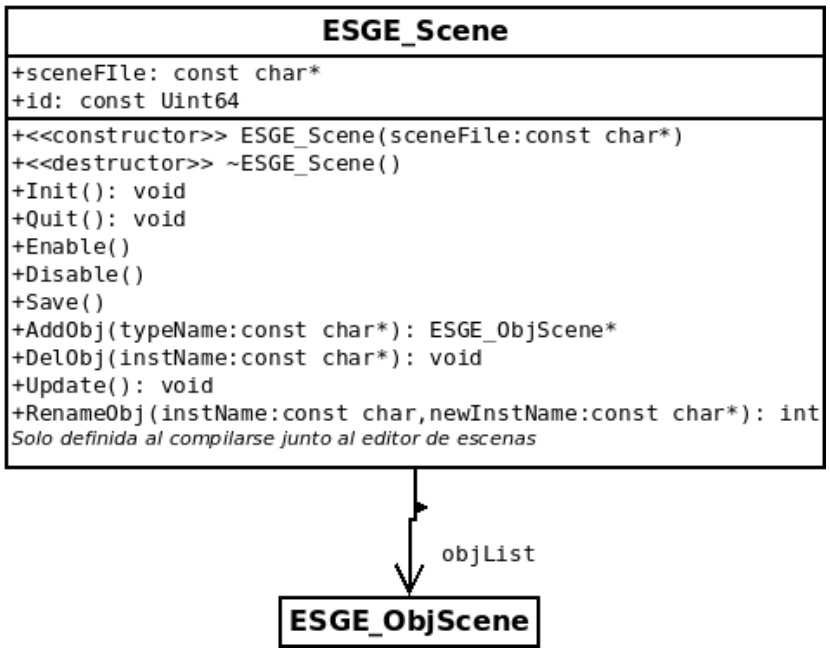

Figura 3.21: Diagrama UML de la clase ESGE_Scene versión 2.

se pospongan hasta actualizar la escena usando el método apropiado que deberá llamarse después de actualizar el estado y físicas de todos los objetos en el bucle principal debido a la posibilidad de que estas acciones se soliciten en medio de este proceso. Esto se consiguió mediante un atributo en los objetos que registra la última acción solicitada. Los efectos de estas modificaciones se muestran en la Figura 3.20 y la Figura 3.21.

Para implementar el sistema que permite la instanciación dinámica y persistencia de los objetos de escenas, se aprovechó la posibilidad del lenguaje C++ de definir variables de clases en un contexto global, requiriendo ser inicializadas con su constructor antes de comenzar el programa y finalizadas con su destructor al terminar. Esto permite ejecutar código que haga cambios con otras variables globales sin necesidad de modificar ningún otro fichero de código fuente. En este caso, esta característica es aprovechada para agregar la única instancia de las clases derivadas de *ESGE_Type* (que representa el tipo de un objeto) en su lista estática de tipos. Luego, durante la ejecución del programa, podrá explorarse esta lista para crear un nuevo objeto y modificar sus atributos de forma que el compilador no puede predecir su tipo. Esta técnica se extrajo del libro [7] para inicializar y finalizar módulos.

También hay que destacar el sistema implementado mediante el uso macros y sobrecarga de funciones para automatizar la definición de la variable global de *ESGE_TypeImpl* que corresponde al nuevo tipo de objeto definido por el desarrollador/a y otra de un vector de *ESGE_Field* para almacenar los nombres y funciones para leer y escribir el valor de cada uno de sus atributos persistentes.

Finalmente, para facilitar la creación y modificación de escenas e independizar de forma estructural el motor gráfico de los juegos desarrollados con este, se desarrollaron dos ejecutables distintos. Uno de ellos será el editor de escenas y el otro el propio juego desarrollado.

El editor permite crear o abrir una escena para añadir, modificar, eliminar, listar objetos en ella mediante sus comandos recibidos por la terminal a la vez que se visualiza la escena y sus cambios en tiempo real por una ventana abierta al comienzo del programa.

El juego permite cargar una escena escogida y ejecutar el bucle principal para comenzar el juego desarrollado.

Para generar el ejecutable del editor, el desarrollador/a solo tendrá que enlazar la biblioteca del motor gráfico para el editor junto a los códigos objeto de sus ficheros fuente con la opción del preprocesador de C/C++ para definir la macro *ESGE_EDITOR*. Para el ejecutable de juego tendrá que hacer lo mismo pero con la biblioteca del motor gráfico para el juego y sin definir la macro.

3.5.4 Prueba del prototipo

Para probar el nuevo prototipo de motor gráfico y sus nuevas incorporaciones funcionales, se adaptaron las clases del objeto protagonista y superficie colisionable de la demo técnica (creadas en la anterior iteración) para heredar de *ESGE_ObjScene* y así poder añadirlos a una escena con el editor para luego probarla en el juego. Luego, se definieron las funciones para leer y escribir el valor de sus atributos persistentes y sus respectivas instancias de la clase *ESGE_TypeImpl* con el fin de poder crear y modificar estos objetos en el editor. Finalmente, se definieron los métodos *OnEnable* y *OnDisable* en ambos objetos para activar y desactivar la detección de pulsación de teclas, el actualizado del estado y físicas, y dibujado del objeto protagonista, y la colisión estática y el dibujado en el caso del objeto superficie colisionable, cuando estes se activen y desactiven en el juego, y *OnEditorEnable*, y *OnEditorDisable* para activar solo el dibujado de ambos objetos cuando se encuentren el editor, al no ser procesador los demás subsistemas por este.

3.6 Quinta iteración: *sprites*, animaciones y audio

En esta iteración se diseñaran e implementarán los componentes necesarios para el dibujado de *sprites*, y reproducción de animaciones, efectos de sonido y música.

3.6.1 Diseño

Para renderizar *sprites* y animaciones, antes de nada es necesario incorporar varios casos de uso a la clase *ESGE_Display* visibles en la Figura 3.22 que representa la pantalla del juego. Primero se diseño un método estático que permitirá abrir una imagen en formato BMP para

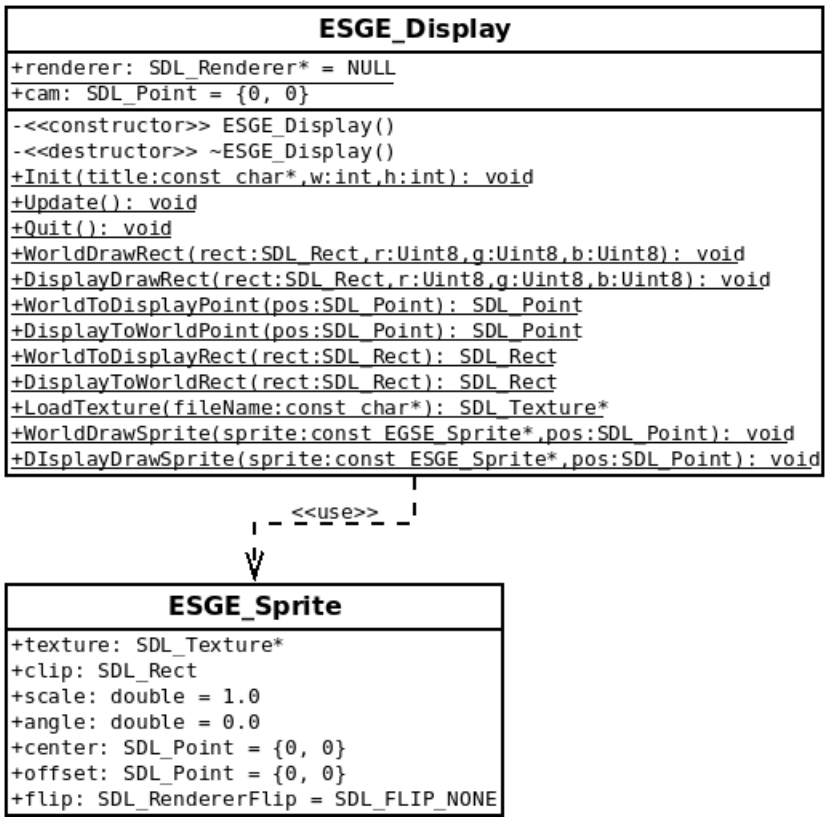

Figura 3.22: Diagrama UML de la clase ESGE_Display versión 3.

obtener su respectiva textura. Esta textura contendrá los *sprites* y permitirá renderizarlos a través de los dos nuevos métodos estáticos para hacerlo en relación a la cámara virtual o no (p.e. interfaz de usuario). Para simplificar y optimizar el uso de estos métodos estáticos, se diseño la estructura *ESGE_Sprite* que contendrá la textura donde se encuentra, la porción de esta que ocupa, el escalado, rotación, efecto espejo que se le aplicará en la pantalla, su pivote de rotación, y la distancia desde el punto en que se dibujará.

Para simplificar la definición de objetos que dibujen *sprites*, haciendo uso del patrón plantilla, se diseño la clase abstracta *ESGE_ObjDrawSprite* de la Figura 3.23 que contiene el *sprite* que se va a renderizar en método sobrescrito de dibujado de la clase *ESGE_ObjDraw* de la que hereda.

Para facilitar la apertura de texturas que representan matrices de *sprites* (*sprite sheets*) y la extracción de sus *sprites*, se diseño la clase *ESGE_Spritesheet* de la Figura 3.24.

Aprovechando la clase *ESGE_Spritesheet*, se diseño *ESGE_AnimPlayer* de la Figura 3.25 donde sus instancias representan un reproductor de una animación que se puede modificar se velocidad, repetirse infinitamente o no y terminar en el segundo caso, además de acceder al *sprite* actual y actualizarse con el tiempo transcurrido mediante sus métodos. Las animaciones

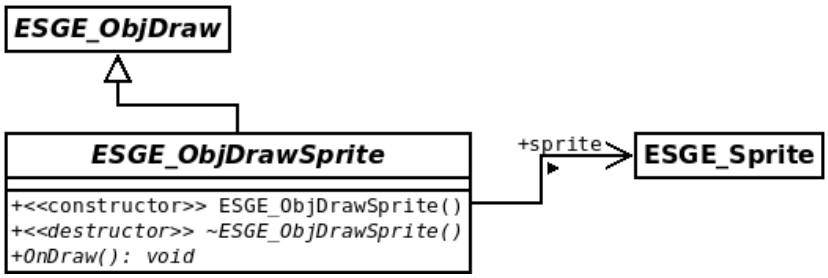

Figura 3.23: Diagrama UML de la clase ESGE_ObjDrawSprite.

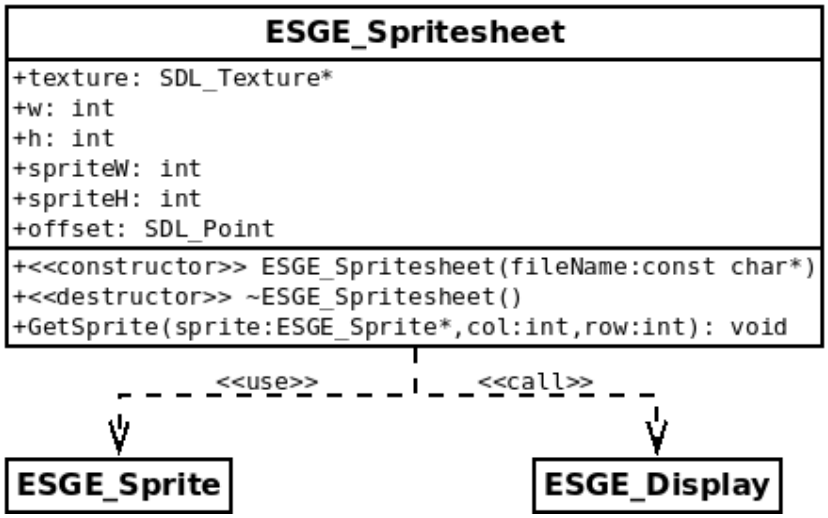

Figura 3.24: Diagrama UML de la clase ESGE_Spritesheet.

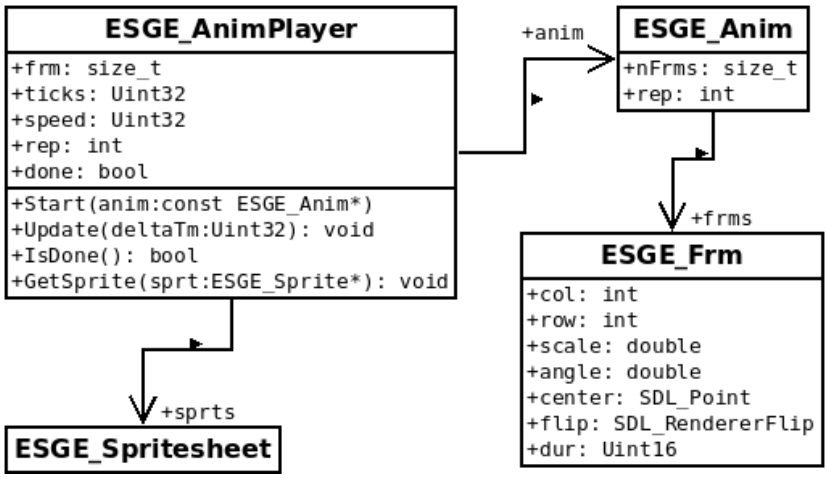

Figura 3.25: Diagrama UML de la clase ESGE_AnimPlayer.

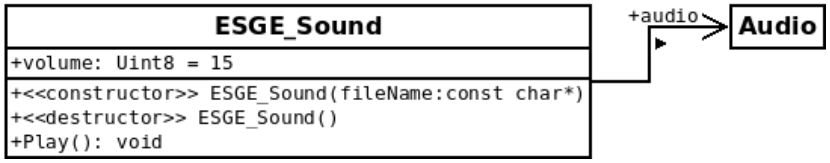

Figura 3.26: Diagrama UML de la clase ESGE_Sound.

no son más que una estructura que contiene cuantas veces se repetirá y un vector con los *sprites* correspondientes a cada fotograma.

Una vez completado todo el apartado gráfico del motor, se paso a diseñar el sonoro con las clases *ESGE_Sound* de la Figura 3.26, y *ESGE_Music* de la Figura 3.27 para encargarse abrir los ficheros de audio en formato WAV y reproducirlos una sola vez en el caso de la primera, y en bucle por la segunda. Además, la primera podrá reproducir varios efectos de sonido al mismo tiempo, sin embargo la segunda solo podrá reproducir una. También se dispondrá de unas funciones para pausar y continuar la reproducción de audio.

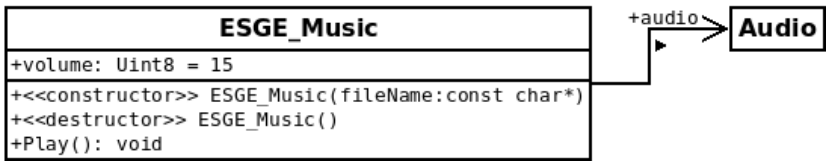

Figura 3.27: Diagrama UML de la clase ESGE_Music.

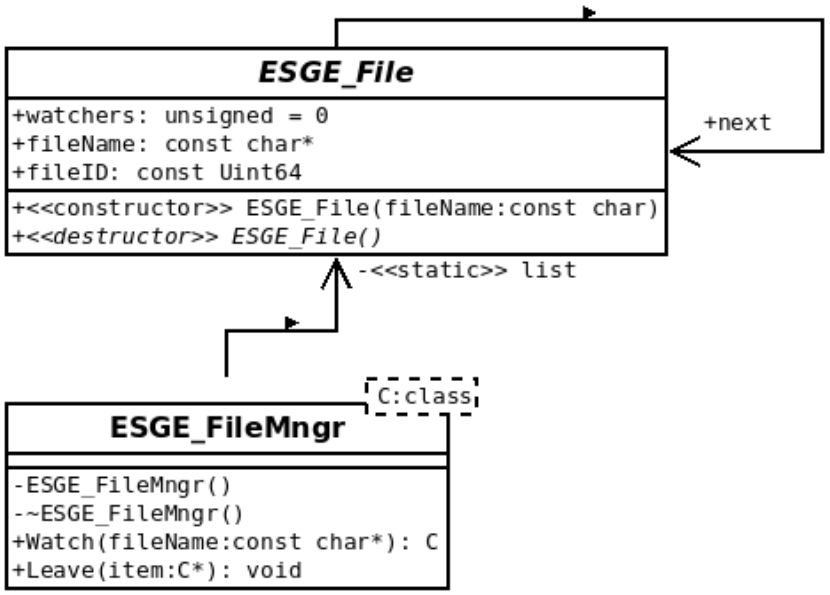

Figura 3.28: Diagrama UML de la clase ESGE_FileMngr.

3.6.2 Análisis de riesgos

Un problema de diseño sobre las clases *ESGE_Spritesheet*, *ESGE_Sound* y *ESGE_Music* es múltiples instancias podrían almacenar el mismo fichero, desperdiciando enormes cantidades de memoria y haciendo lecturas de ficheros innecesarias.

3.6.3 Re-diseño y desarrollo

Para lidiar con el anterior problema, basándose en la solución del libro [7] sobre el conteo de referencias, se diseñaron las clases *ESGE_File* y *ESGE_FileMngr* de la Figura 3.28. La primera es abstracta y sirve como base para que las clases que dependen de ficheros la hereden, llevando cuenta de las referencias a una misma instancia que tiene el fichero, y la segunda es una plantilla singleton para cada clase que derive de *ESGE_File*, esta servirá de intermediario para saber si es necesario crear una instancia y leer el fichero si aun no se hizo cuando se observa a través de su método estático, y destruirla si no tiene ninguna cuando se libera a través de su otro método estático. Asegurando de esta manera solo tener un único fichero en memoria con la dirección entregada.

Después, se adaptaron las clases *ESGE_Spritesheet*, *ESGE_Sound* y *ESGE_Music* para heredar de *ESGE_File* y así poder obtener la instancia correspondiente al fichero por medio de su clase plantilla *ESGE_FileMngr*. Los cambios pueden ser observados en la Figura 3.29, en la Figura 3.30, y en la Figura 3.31.

Algo que destacar de la implementación es la reparación que fue necesaria hacer en la

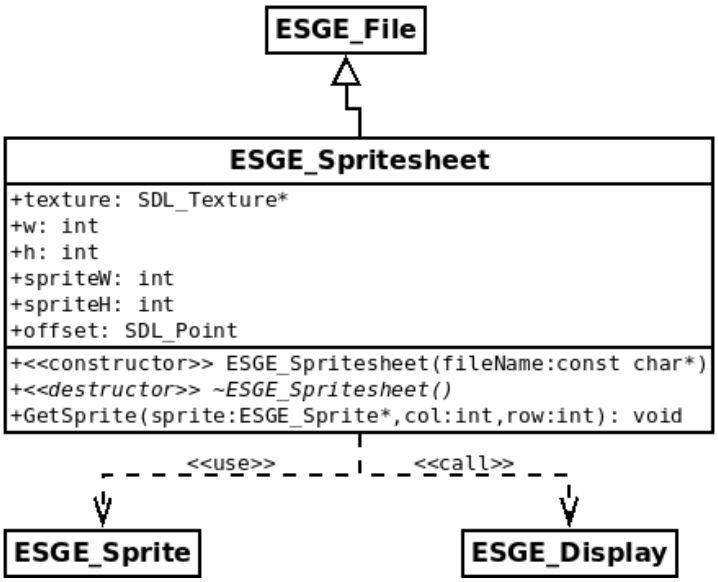

Figura 3.29: Diagrama UML de la clase ESGE_Spritesheet versión 2.

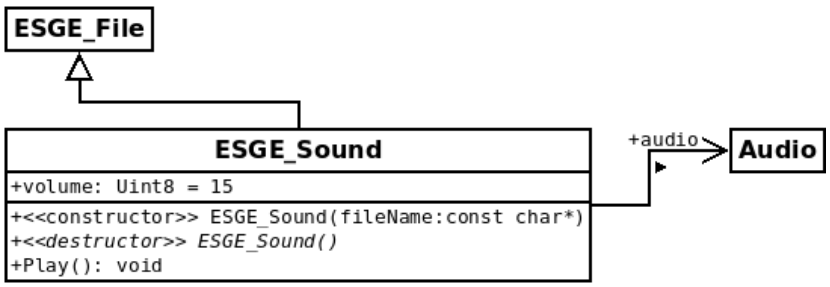

Figura 3.30: Diagrama UML de la clase ESGE_Sound versión 2.

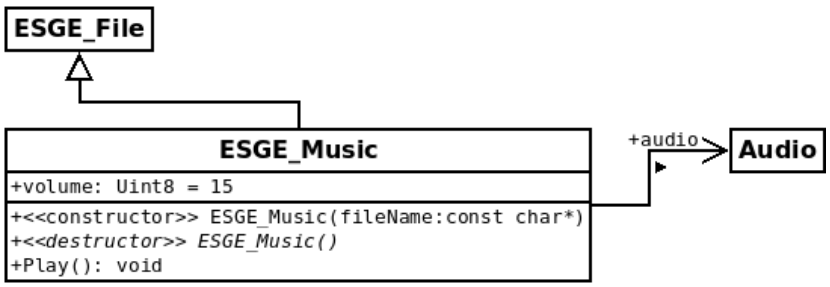

Figura 3.31: Diagrama UML de la clase ESGE_Music versión 2.

biblioteca *Simple SDL2 Audio* [19] usada para simplificar la gestión de la reproducción de sonido con SDL, ya que la forma en que liberaba los recursos de sonido era inmediata y podría producir que el hilo que esta reproduciedo leyera un sonido que ha sido liberado, provocando un fallo de segmentación. Para solucionarlo fue necesario entender el funcionamiento de la biblioteca y hacer que la liberación de sonidos sea pospuesta hasta que el hilo termine de reproducirlo.

3.6.4 Prueba del prototipo

Con el objetivo de probar las animaciones, efectos de sonido y música, se modificó la demo técnica y su clase del objeto protagonista para que herede de *ESGE_ObjDrawSprite* en vez de *ESGE_ObjDraw* directamente, permitiendo que este dibuje su *sprite* a partir de una instancia de *ESGE_Spritesheet* que contiene en un atributo. Además, se definieron varias animaciones que podrá reproducir mediante su instancia de *ESGE_AnimPlayer*. También se le añadieron unos atributos *ESGE_Sound* para reproducir efectos especiales en sus acciones.

También, se definieron un par de nuevas clases, una para representar un tile visible con un *sprite* que servirá de decoración para la demo técnica. Esta clase hereda de *ESGE_ObjCell* y *ESGE_ObjDrawSprite* para organizar la colocación y permitir que dibuje su *sprite*. La otra será un objeto que tendrá un atributo *ESGE_Music* para reproducir una canción nada mas activarse.

3.7 Sexta iteración: Persistencia del juego

En esta iteración se diseñará e implementará el sistema para que el desarrollador/a pueda guardar el estado del juego para luego cargarlo y retomar la partida de una forma persistente y portable entre distintas plataformas.

3.7.1 Diseño

Este sistema constará en una serie de funciones para leer y escribir en binario sobre un *stream* de un fichero valores de tipo entero con un tamaño de 8, 16, 32, 64 bits, número flotante, o cadena de caracteres, haciendo de estos dos procesos más rápido que otros métodos de persistencia basados en caracteres.

3.7.2 Análisis de riesgos

Un posible problema es que las función de escritura de cadenas de caracteres escriba en búfer entregado pueda ser escrito fuera de rango si no se tiene en cuenta previamente su tamaño.

Otro problema podría darse si el fichero en el que se guardan los datos pueda ser leído en una plataforma con una arquitectura de CPU de diferente *endianidad*, provocando que sus bytes sean interpretados al revés.

También podría resultar complicado modificar de forma externa algún valor contenido en el fichero, requiriendo de herramientas de lectura de ficheros en hexadecimal.

Además, esta forma de guardar la información es inflexible, no se puede modificar la estructura de valores dentro del fichero o estos serán corruptos. Se requeriría una herramienta externa para reestructurarlos.

3.7.3 Re-diseño y desarrollo

Para cubrir el primer problema, la función de lectura de cadenas de caracteres requerirá entregarle. aparte del búfer, el tamaño de este para detener su escritura antes de superar su rango de memoria e ignorando los siguientes caracteres no nulos.

Sobre la *endianidad* de los valores guardados en el fichero, se solucionó asegurando que los bytes sean escritos y leídos en el mismo orden sin importar la arquitectura de la plataforma.

Los dos últimos problemas de interpretar y modificar la estructura del fichero de forma externa son un coste a pagar por la mayor eficiencia en la lectura de los valores almacenados en un fichero.

En la implementación, se aprovecho la API de la biblioteca SDL para manejar *streams* de ficheros que abstrae la biblioteca usada para la misma tarea en cada plataforma, garantizando una mayor portabilidad que de usarlas directamente. Además, SDL también proporciona funciones para la manipulación individual de bytes y detección de *endianidad* muy útiles para solucionar el segundo problema mencionado.

3.7.4 Prueba del prototipo

Para probar estas funciones, se le añadió al objeto protagonista de la demo técnica la captura de dos nuevas teclas con el fin de guardar y cargar su posición actual siempre que se quiera incluso después de cerrar el juego. Lo cual funcionó bastante bien en el ordenador de desarrollo pero no tanto en el sistema empotrado. Obteniendo un fallo de segmentación nada más ejecutarlo cuando se compilaba sin optimizaciones, y al cambiar de escena con ellas. Lo cual resultó bastante desconcertante en un principio pero se resolvió depurando paso a paso el código en el sistema empotrado y observando que, a pesar de tener en cuenta del tamaño del búfer en la función de lectura de cadenas de caracteres, se estaba escribiendo fuera de rango por un fallo en la implementación de la función, provocando que una variable local adyacente en la pila de la memoria del programa cambiara de valor con consecuencias desastrosas. Finalmente se solucionó programando correctamente la función.

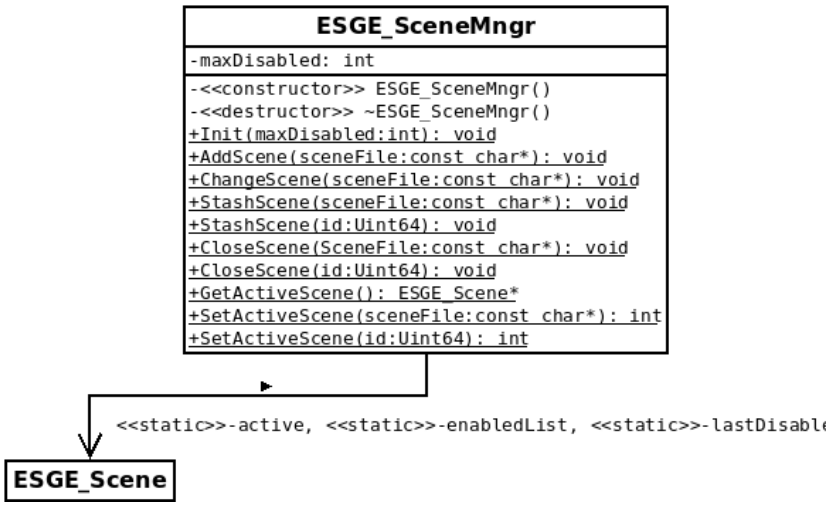

Figura 3.32: Diagrama UML de la clase ESGE_SceneMngr.

3.8 Séptima iteración: Transición y múltiples escenas

En esta última iteración se diseñaran, implementarán, y probarán, los mecanismos necesarios para permitir la transición o el añadido, la ocultación, y el cierre de escenas en el juego. A mayores, debido al tiempo sobrante y los requisitos establecidos al principio del proyecto, se diseñará también las clases encargadas de gestionar la captura de eventos relacionados con el ratón y mando de consola.

3.8.1 Diseño

Estos mecanismos serán implementados por la clase singleton *ESGE_SceneMngr* de la Figura 3.32 encargada de la gestión de escenas. Esta ofrece los casos de uso enumerados en la planificación (descritos en la Sección 2.3.8) en sus respectivos métodos que podrán ser llamados por cualquier objeto del juego.

Estos casos de uso se consiguieron a través del guardado de la última activada y dos pilas de escenas, una pila registrará cuáles se encuentran activadas y la otra las que no (ocultas). Su utilidad en los métodos se demuestra a continuación:

- ***ChangeScene*:** Primero busca en la pila de escenas desactivadas si la escena ya se encuentra cargada, si es así la activa de nuevo, sino, la carga, la añade a la pila de activadas y la guarda como última activada. Finalmente se desactivan todas las demás escenas activas, pudiendo ser cerradas si no queda espacio en la pila de desactivadas.

- ***AddScene*:** Igual a la anterior pero saltándose el último paso.

- ***StashScene*:** Busca la escena en la pila de activadas, si la encuentra, la desactiva, la añade a la pila de desactivadas y si era la última activa guarda la siguiente escena como la última activa, si no queda espacio, cierra primera desactivada.
- ***CloseScene*:** Busca la escena en la pila de activadas, si la encuentra, la desactiva para luego cerrarla y si era la última activa guarda la siguiente escena como la última activa, finalizado y destruyendo todos los objetos en el proceso, si no la encuentra entre las activadas, busca en las desactivadas y si la encuentra la cierra.
- ***GetActiveScene*:** Devuelve la última escena activada.
- ***SetActiveScene*:** Busca la escena en la pila de activadas y si la encuentra, la guarda como última activa.

El uso de estos métodos queda ilustrado en la Figura 3.33, mientras que los dos últimos se reservan para ser usados por el editor y los métodos *Create* y *Destroy* de la clase *ESGE_ObjScene*.

Finalmente, aprovechando haber satisfecho casi todos los requisitos del motor gráfico, se dispuso a diseñar una solución para los pendientes. Estos pedían poder capturar eventos relacionados con el ratón y mando de consola, por lo que, basándose en el diseño de la clase *ESGE_ObjKeyEvent*, se pensó en dos clases abstractas *ESGE_ObjMouseEvent* (de la Figura 3.34) y *ESGE_ObjKeyEvent* (de la Figura 3.35), que de la misma manera que en la que se basan, estas serán heredadas por los objetos del juego activos e interesados en ser notificados por los distintos métodos abstractos para cada tipo de evento.

3.8.2 Análisis de riesgos

El único defecto encontrado al diseño de las clases *ESGE_Scene* y *ESGE_SceneMngr* es que de nuevo, al poder realizar de forma inmediata cualquiera de sus casos de uso por un objeto que esta siendo actualizado, podría darse de nuevo el caso ilustrado en la Figura 3.19 ya que al destruir una escena, deberán hacerlo también sus objetos.

3.8.3 Re-diseño y desarrollo

De forma similar a la solución tomada en la clase *ESGE_ObjScene* se decidió modificar el diseño de *ESGE_Scene*, guardando la última acción solicitada para sea pospuesta hasta que *ESGE_SceneMngr* las actualice en su correspondiente método, que se deberá llamar en el bucle principal antes de dibujar los objetos. El resultado de estos cambios se puede ver en la Figura 3.36 y la Figura 3.37.

En la implementación, se aprovechó la posibilidad de tener activas varias escenas simultáneamente para permitir modificar varias escenas en la misma sesión o ejecución del editor,

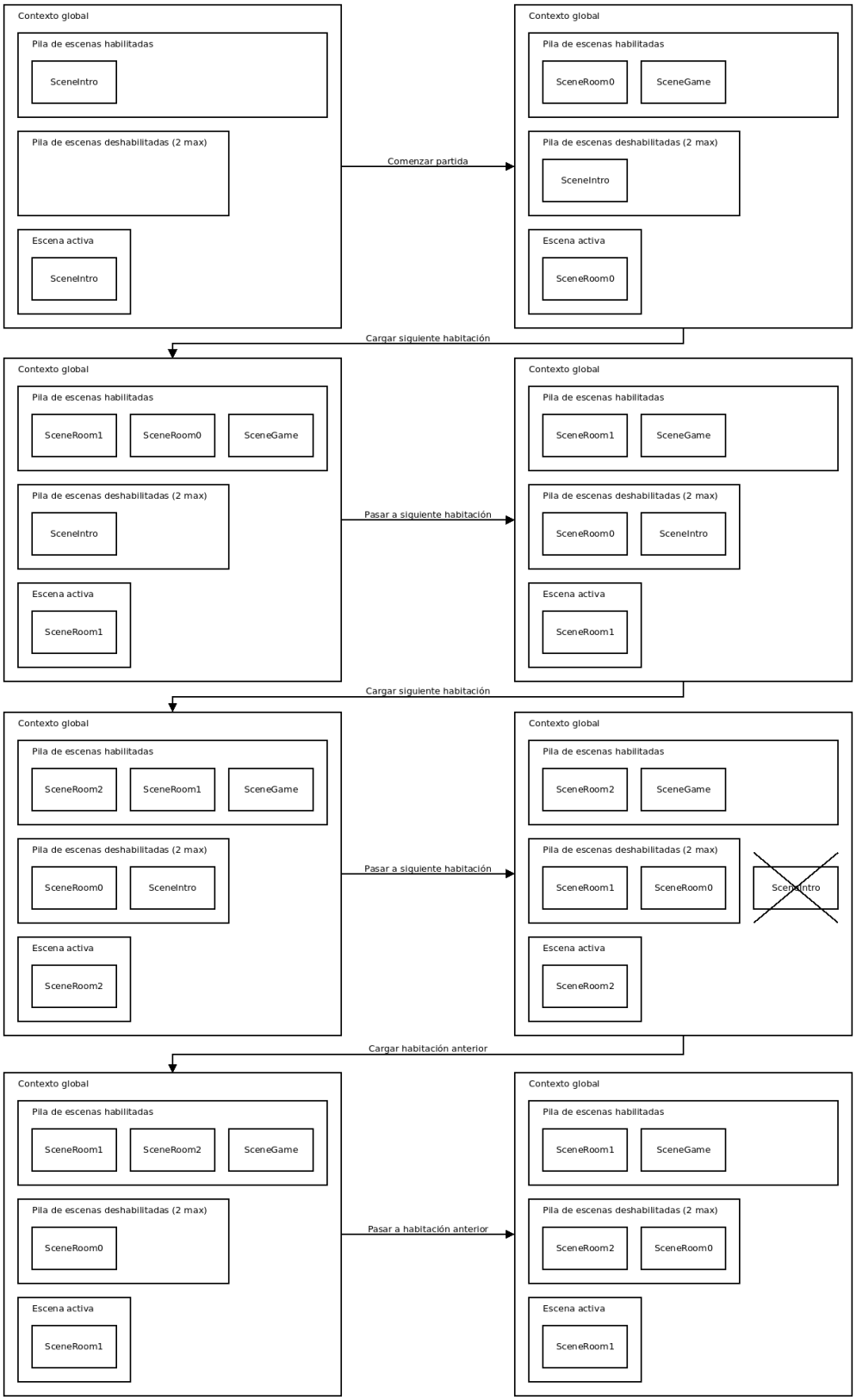

Figura 3.33: Ejemplo del flujo de escenas activadas y desactivadas.

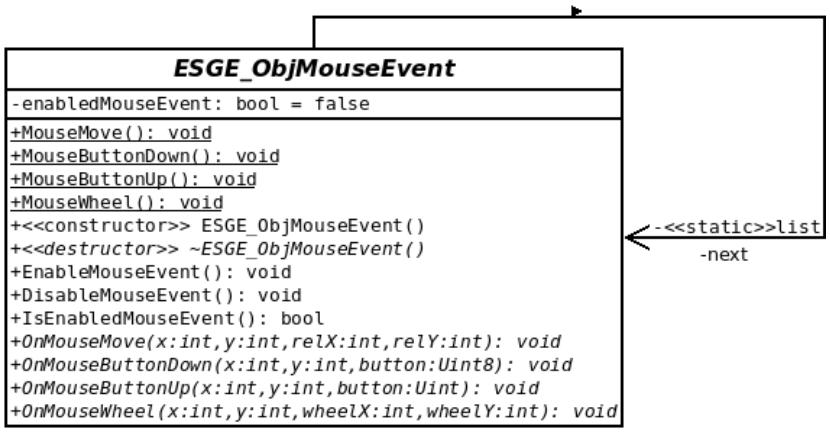

Figura 3.34: Diagrama UML de la clase ESGE_ObjMouseEvent.

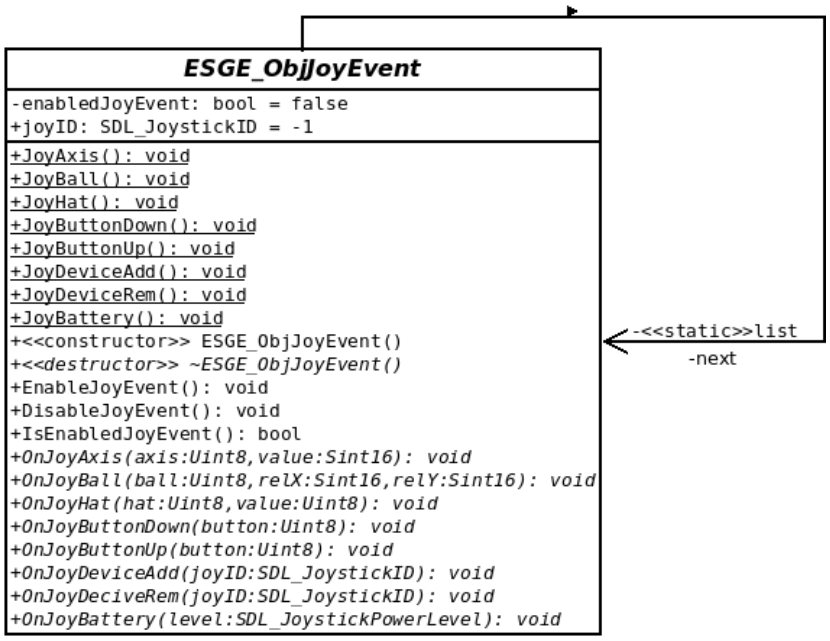

Figura 3.35: Diagrama UML de la clase ESGE_ObjJoyEvent.

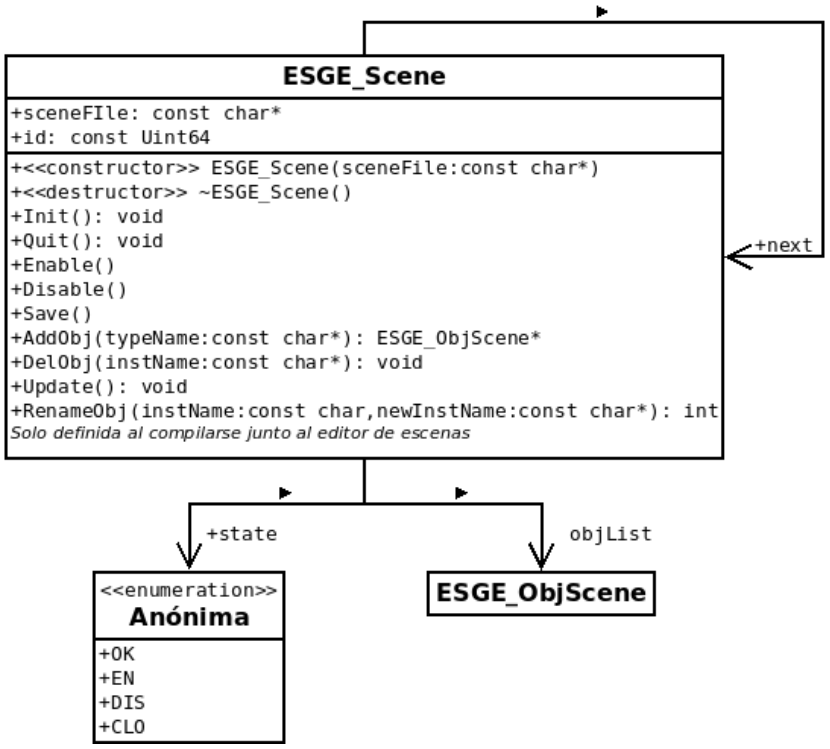

Figura 3.36: Diagrama UML de la clase ESGE_Scene versión 3.

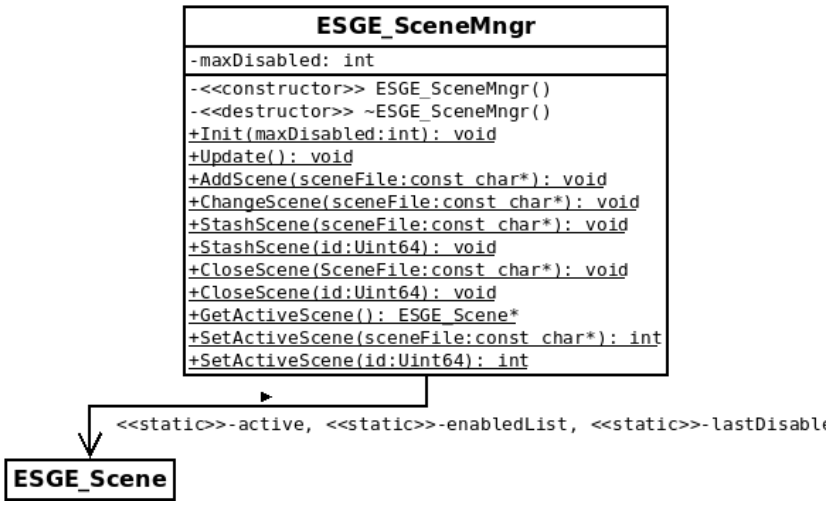

Figura 3.37: Diagrama UML de la clase ESGE_SceneMngr versión 2.

añadiendo dos comandos para seleccionar y preguntar cual escena se esta editando de las activas. Esto permite la visualización del escenario que formarían varias escenas coexistentes y modificar rápidamente cada una sin tener que ejecutar de nuevo el editor.

3.8.4 Prueba del prototipo

Con el objetivo de concluir la demostración técnica probando el nuevo gestor de escenas *ESGE_SceneMngr* y las clases de recepción de eventos de ratón y mando de consola *ESGE_ObjMouseEvent* y *ESGE_ObjKeyEvent*, se modifico la clase del objeto protagonista para heredar de estas dos últimas, permitiéndole detectar eventos como el movimiento del ratón o del *joystick* de un mando de consola que le serán útiles para realizar las mismas acciones con distintos controles.

Además, se definió un nuevo tipo de objeto que representará un gestor de habitaciones, que vienen a ser escenas con un tamaño delimitado por un rectángulo. El objeto protagonista le indicará su posición para añadir y ocultar las habitaciones por las que pase. Esto servirá de prueba para los casos de uso de la clase *ESGE_SceneMngr*. También se crearon varias escenas que representarán las habitaciones por las que se podrá navegar en la demostración técnica.

Una ilustración de los posibles escenarios que se pueden dar en la demostración técnica se muestra en la Figura 3.38

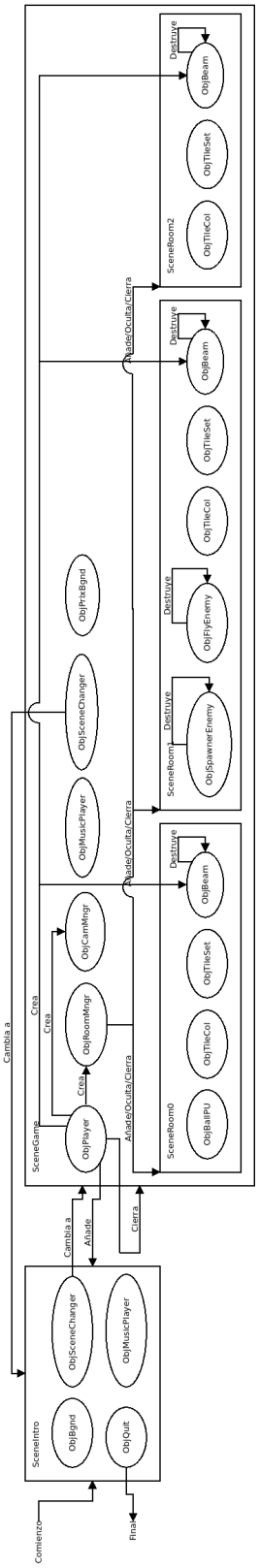

Figura 3.38: Ejemplo de los distintos contextos de objetos formados por la composición de escenas y sus transiciones.

Capítulo 4

Conclusións

En este capítulo se hará un breve recorrido al estado actual de los objetivos marcados por el proyecto junto a sus soluciones y puntos destacados, luego se propondrán los siguientes pasos para continuar con el desarrollo del proyecto, para finalmente describir los conocimientos de distintos ámbitos que fueron necesarios para conseguir analizar, diseñar, implementar y probar las soluciones de cada objetivo.

4.1 Resultados del proyecto

Para poder valorar el grado de completud del proyecto es necesario analizar las soluciones conseguidas hasta el momento y las que se podrían aplicar a los objetivos que no se pudieron acadar por razones de excesiva complejidad y falta de tiempo, y a los que aparecieron durante la duración del proyecto.

4.1.1 Objetivos cumplidos

- **Base simple pero potente:** El motor gráfico debía satisfacer esa necesidad de tener un punto de partida sencillo pero potente para comenzar a desarrollar videojuegos por aquellos que no quieren complicarse la vida con motores más sofisticados o no se lo pueden permitir. Por eso el motor tiene licencia de código abierto y se diseño buscando una sencillez global que permita ser estudiado y comprendido por el desarrollador antes de empezar a implementar los prototipos de sus juegos, mientras se mantiene un buen rendimiento de los sistemas del motor gráfico.

- **Para sistemas empotrados:** Desde el principio se contemplo la necesidad de que el motor gráfico y su demostración técnica puedan funcionar correctamente en un sistema empotrado, para ello fue necesario encontrar uno disponible, lo cual en un principio fue fácil, pero este tenía instalado el sistema operativo *Android*, lo que no debería ser un

problema para ejecutar los videojuegos pero si para desarrollarlos al imponer el uso de sus herramientas de desarrollo software, por eso se intentó instalar en el sistema empotrado *Debian* [31] usando las herramientas ADB [32] y FastBoot [33] de *Android* sin mucho éxito. Por eso se paso a usar una *Raspberry Pi 1 model B* la cual tiene un proceso de instalación de sistema operativo bastante sencillo, sin embargo, para conseguir un rendimiento máximo, se prescindió del entorno de escritorio requiriendo una instalación bastante compleja de la biblioteca de SDL para poder visualizar los videojuegos en la que se tuvo que usar conocimientos avanzados de la pila gráfica de *Linux*.

- **Carga y descarga de objetos:** Para permitir crear y cargar escenas compuestas de objetos es necesario conservar los valores iniciales de sus atributos persistentes, fue necesario crear un sistema para crear y modificar objetos de una forma dinámica que garantice la escalabilidad del motor gráfico, este sistema hace uso de funcionalidades avanzadas del lenguaje de programación C++ como son las plantillas y los métodos virtuales puros, y al prescindir por completo de su biblioteca estándar, fue necesario definir varias de sus funciones como los operadores *new* y *delete*.

- **Interacción del jugador:** Con el objetivo que el jugador pueda manejar el estado de los objetos del juego mediante distintos periféricos como pueden ser: un teclado, un ratón o mando de consola, se diseñaron e implementaron varias clases abstractas que los objetos deberán heredar para ser notificados de cada tipo de evento haciendo uso del patrón plantilla, y observador.

- **Actualización de objetos:** Para que los objetos puedan cambiar su estado interno y reaccionar ante las acciones del jugador, se creo una clase abstracta que heredarán todos los objetos interesados en actualizarse, contemplando el caso en que estos se instáncien eliminen en medio del proceso de actualización. Este caso se resolvió posponiendo la creación y destrucción de objetos a justo antes de que se dibujen.

- **Dibujado *sprites* y animaciones por capas:** Aspectos vitales para producir la imagen del videojuego que se solucionaron mediante la creación de una única ventana en la que se podrán cargar y dibujar *sprites*. Con esto, reproducir animaciones es tan fácil como seguir una secuencia temporal de dibujado de *sprites*, implementado mediante un vector de *sprites* y un reproductor de animaciones que lo recorre.

- **Reproducción de efectos de sonidos y música:** Se solucionó usando la biblioteca *Simple SDL2 Audio* para crear dos clases que abren ficheros de audio y los reproducen una solo vez si es un sonido y en bucle si es música.

- **Persistencia del estado del juego:** Gracias a la abstracción de la biblioteca para manejar la lectura y escritura de ficheros que proporciona SDL se pudo proveer al usuario

de las herramientas para guardar en ellos valores de números enteros de distintos tamaños, flotantes, y cadenas de caracteres. Esto le servirá para conservar las variables necesarias para retomar el estado del videojuego.

- **Desacople del motor respecto al juego:** Conseguido mediante dos bibliotecas que constituyen el motor gráfico, una implementa la función *main* para ejecutar el editor de escenas, y la otra para la implementa para ejecutar el juego con su bucle principal. Por lo que el desarrollador solo tendrá que enlazar la biblioteca con sus códigos objeto, donde define objetos del juego, para generar el editor de escenas o juego dependiendo de cual escoja. Esto permite usar los objetos definidos por el desarrollador en el editor de escenas y el juego sin necesidad de incluir manualmente en el motor gráfico cada definición de las clases del desarrollador que representa objetos del juego.

- **Demostración técnica:** Finalmente para probar las funcionalidades del motor gráfico, validar que este puede funcionar en sistemas empotrados y ordenadores, y ejemplificar cómo se desarrollaría un videojuego usando el motor, se desarrollo una pequeña demostración técnica con éxito, aunque no fue tarea fácil ya que en todo momento había que tener presente las limitaciones de recursos computacionales del sistema empotrado. Esto sumado al uso del lenguaje C++, constituyeron la causa de la mayoría de fallos ocurridos al ejecutar en este la demostración técnica, requiriendo un proceso de depuración bastante engorroso y tedioso.

4.1.2 Objetivos pendientes

- **Distintos géneros de juegos**: Debido a la restricción de duración del proyecto, no fue posible desarrollar múltiples demostraciones técnicas que prueben que es posible crear distintos géneros de videojuegos usando este motor gráfico. Sin embargo, con tiempo, esto no sería fácil de solucionar, ya que no se observo ningún aspecto técnico que limite las posibilidades de desarrollar un videojuego con distintas mecánicas.

- **Multiplataforma**: Por la misma razón que la del anterior objetivo pendiente sumado a no disponer de una gran variedad de plataformas donde probar el motor gráfico con su demostración técnica, no ha sido posible demostrar de forma efectiva la portabilidad de este. No obstante, al solo depender el motor de la biblioteca SDL y esta garantizar estar disponible múltiples plataformas, no debería haber gran problema para conseguir que el motor y la demo funcionen en estas.

4.1.3 Objetivos nuevos

- Al haber usado listas enlazadas para almacenar los objetos del juego en las escenas y clases abstractas activables de las que heredan, se esta produciendo una segmentación en la memoria y fallos caché que se podría evitar usando una gestión de la memoria basada en regiones[24] recomendada por el director del proyecto.
- Durante el diseño del sistema de actualización de objetos, se pensó en la posibilidad procesar simultáneamente los objetos independientes entre sí en varios hilos de ejecución para aprovechar el potencial de arquitecturas de procesadores multi-núcleo. Se abandonó la idea por complejidad pero podría ser interesante retomarlo en un futuro.
- Debido a usar el lenguaje C++ sin RTTI (Run Time Type Information) por cuestiones de compatibilidad con determinados sistemas donde su compilador no dispone de esta característica, resulta imposible hacer Down Casting, no es muy eficiente y tampoco se considera muy buena práctica salvo en circunstancias donde no queda otra opción por cuestiones de arquitectura del software. Por eso sería necesario seguir la solución del libro [7] para este problema.
- Después de usar bastante el editor, resulta muy necesaria la interacción con este mediante un ratón para poder seleccionar los objetos y cambiarlos de lugar de forma sencilla, sin el uso de comandos.

4.2 Trabajos futuros

Una vez satisfechos los pendientes y nuevos objetivos, el siguiente paso sería añadir nuevas funcionalidades al motor gráfico para hacerlo más sofisticado y potente presentadas a continuación:

- Con el fin de mejorar la experiencia del desarrollador de videojuegos que use este motor gráfico, resulta imprescindible crear una interfaz gráfica al editor de escenas más cómoda que el uso de comandos por una terminal.
- Un aspecto limitante en motor gráfico es el uso de la API de renderizado de SDL que no permite producir efectos gráficos vistosos usando la GPU, por eso sería interesante ofrecer el uso directo de *shader*s mediante la biblioteca OpenGL ES y la creación de su contexto en la ventana de SDL.
- Algo que requeriría más trabajo es sin duda el motor de físicas, que actualmente no tiene en cuenta la interacción entre objetos móviles colisionables. Sería necesario dotarlos de masa para simular unas físicas más realistas y ofrecer más formas que solo

rectángulos. Otra opción sería usar un motor de físicas bidimensionales ya existente como Box2D [34].

- También sería importante usar la sub-biblioteca *SDL_mixer* [35] para permitir abrir una mayor cantidad de formatos de ficheros de audio y gestionar de forma más adecuada la reproducción de sonido.
- Otra sub-biblioteca de SDL que conviene usar para posibilitar la lectura de más formatos de ficheros de imágenes sería *SDL_image* [35].
- Para conseguir desarrollar videojuegos multi-jugador *online* es necesario gestionar conexiones TCP y UDP, esto se podría hacer empleando la sub-biblioteca *SDL_net* [35].
- Hacer que el desarrollador implemente su propio sistema de renderizado de texto puede resultarle tedioso y complejo, por eso sería buena idea incluir la sub-biblioteca *SDL_ttf* [35] en el motor gráfico para rasterizar caracteres del sistema.
- El motor gráfico carece actualmente de un sistema de *widgets* para diseñar interfaces gráficas de usuario, podría usarse las bibliotecas *ImGUI* o *MicroUI* [35] para cubrir este aspecto.
- La biblioteca SDL 2 esta en una transición no retro-compatible a SDL 3 [36], la cual ofrece nuevas funcionalidades pero requiere una migración del código que use la anterior versión para aprovecharlas.

Debido a las repercusiones estructurales que tendrían todos estos cambios y a los nuevos conocimientos adquiridos durante el desarrollo del proyecto, resultaría necesario rehacer el motor gráfico casi de cero para garantizar su escalabilidad y consistencia.

4.3 Conocimientos empleados

Al empezar este proyecto se contaba con conocimientos sobre la estructura de los computadores, los componentes de los sistemas operativos, la gestión de distintos hilos de ejecución, el paradigma orientado a objetos, los patrones de diseño software, las metodologías de desarrollo software, las tecnologías usadas en computación gráfica, y el desarrollo de videojuegos, adquiridos a lo largo de los estudios realizados en la titulación de Ingeniería Informática por la mención de Computación. Aparte de conocimientos conceptuales, en estos estudios también se aprendió a usar herramientas como el lenguaje C [37], el gestor de pendencias Make [38], el compilador GCC, la biblioteca para manejar hilos *pthread*, el analizador de memoria Valgrind [39], la biblioteca gráfica OpenGL, la biblioteca para desarrollar juegos con python

PyGame [40], y el motor gráfico Unity [41] que fueron fundamentales para el proyecto, bien mediante su uso o como referencia.

También se contaba con experiencia desarrollando videojuegos 2D de aspecto clásico desde los catorce años de edad, haciendo uso del motor gráfico 2D Game Maker Studio [42] del que se tomó gran inspiración. Lo que permitió formarse en un dialecto del lenguaje de programación C++ y la generación de gráficos 2D mucho antes si quiera de entrar en el grado. Formando un pequeño equipo de desarrollo de videojuegos con amigos para delegar ellos las otras disciplinas necesarias en este ámbito.

Además, recientemente también se obtuvo experiencia entorno a los sistemas empotrados y el bajo nivel en las prácticas en empresa cursadas. En estas fue necesario programar microcontroladores en el lenguaje C de una forma muy distinta a la aprendida en el grado, con limitaciones técnicas muy evidentes y sin disponer de sistema operativo ni de la biblioteca estándar de C. Sin embargo, el control del dispositivo esta completamente disponible para el desarrollador, escribiendo directamente en registros para activar módulos lógicos desactivados para ahorro energético, etc. También ser usaron otras tecnologías como el lenguaje C++, Make, GDB y la comunicación serie UART relacionadas con el proyecto.

Cabe añadir que, con el objetivo de desarrollar este proyecto, fue necesaria una formación de cuatro años, realizada independiente a los estudios, sobre el uso de la biblioteca SDL (Simple DirectMedia Layer) [43] la cual ofrece una enorme cantidad de funciones para cada uno de los apartados de un videojuego de una forma abstracta que nos libra de gestionar manualmente, mediante *macros*, las dependencias en cada plataforma, facilitando en gran medida el proceso de porteo de los videojuegos a distintos sistemas operativos y plataformas. Esta formación se realizo mediante el uso de su wiki [44], el seguimiento de tutoriales [45, 46], y el estudio de su código fuente [36] y proyectos que usaron esta biblioteca [16, 17].

Toda esta formación tomo un papel crucial en tareas del proyecto como la planificación del mismo, el análisis de requisitos, la instalación headless del sistema operativo *Raspbian* y el la biblioteca SDL en la *Raspberry Pi 1 model B*, el diseño de las clases siguiendo buenas prácticas y patrones software, la gestión de riesgos, la implementación de pruebas, la compilación y generación de ejecutables, y la depuración de estos.

Como resultado de participar en este proyecto, se han mejorado y consolidado los conocimientos previos en el campo de la Ciencias de la Computación, al mismo tiempo que he adquirido nuevas habilidades y conocimientos en la área de Ingeniería de Computadores. Esto demuestra la capacidad para integrar estos dos campos y alcanzar con éxito los objetivos establecidos para el proyecto, a pesar de no haber cursado formalmente la segunda área mencionada.

Apéndices

Apéndice A

Manual de iniciación del desarrollador/a

En este capítulo se describen brevemente los pasos básicos para que el desarrollador/a de videojuegos pueda crear su primer prototipo de juego usando el motor gráfico del proyecto.

A.1 Preparativos

Antes de nada, es necesario instalar la biblioteca dinámica SDL. Para ello, si se encuentra en un sistema con *Linux* y *apt*, bastará con ejecutar el siguiente comando:

```
sudo apt-get install libsdl2-dev
```

En caso contrario, puede compilar manualmente SDL siguiendo la guía que se encuentra en su repositorio[36].

Tras esto, se deberá clonar el repositorio del proyecto [1].

Luego, se compilará las dos bibliotecas estáticas del motor gráfico usando la herramienta *Make* en la carpeta "build" del proyecto.

Después, se debe crear una nueva carpeta dentro de "test" donde se encontrarán todos los recursos de nuestro juego, imágenes, sonidos, música, escenas, y nuestro código fuente. Llamaremos a esta "demo1" por ejemplo.

Finalmente, será necesario copiar el fichero "Makefile" del directorio "test/demo0" al nuestro con el fin de construir las herramientas necesarias para desarrollar y probar el juego.

[1] https://github.com/NicoVazCan/ESGE

A.2 Definición de un nuevo tipo objeto del juego

A continuación se explicará el proceso de definición de nuestro nuevo tipo de objeto dentro del juego.

Estando dentro de nuestra carpeta deberemos crear un nuevo fichero de código fuente del lenguaje C++. En este definiremos un nueva clase donde sus instancias representan objetos del juego de un mismo tipo.

Para permitir que sus objetos aparezcan en escenas debe siempre heredar de la clase abstracta *ESGE_ObjScene* proporciona por el motor gráfico al incluir la cabecera de C++ "ESGE_scene.h".

A mayores, para darle un aspecto se tendrá que heredar de la clase abstracta *ESGE_ObjDraw* e incluir su cabecera "ESGE_objDraw.h".

Tras eso, se implementará el método virtual *OnDraw* donde se dibujará un rectángulo de cierto color usando el método estático *DisplayDrawRect* de la clase *ESGE_Display* del cabecero "ESGE_display.h"

Después, se implementaran los métodos virtuales *OnEnable* y *OnDisable* de la clase abstracta *ESGE_ObjActive* de la cabecera "ESGE_objActive.h" para activar el dibujado con el método *EnableDraw* y desactivarlo con el método *DisableDraw* respectivamente. Lo mismo se hará con los métodos virtuales *OnEditorEnable* y *OnEditorDisable* para que el dibujado también este habilitado en el editor. Pero estos dos últimos métodos solo deberán se declarados y definidos cuando la macro *ESGE_EDITOR* este definida.

Finalmente, se deberá usar la macro *ESGE_TYPE* a la que se le entregará el nombre de nuestra clase.

A.3 Generación de los ejecutables

Para generar los ejecutables de nuestro editor de escenas y juego, se deberá usar la herramienta *Make* en nuestro repositorio, que compilará nuestro fichero código fuente para ser enlazado con la biblioteca del motor gráfico con el editor y la del juego.

A.4 Creación de una escena y sus objetos

Ya disponiendo del ejecutable del editor, procederemos a crear una nueva escena. Para ello se deberá ejecutar entregándole el nombre de la escena a crear, si no se le especifica, tomará "scene.bin" por defecto.

Una vez dentro del editor, podemos observar que se abre una nueva ventana en negro debido a que se encuentra vacía.

Para añadir nuestro objeto en ella bastará con poner el siguiente comando por la terminal del editor de escenas:

```
add [nombre de la clase del objeto]
```

Si todo fue bien, debería observar en la ventana del editor una figura rectangular que representa nuestro objeto.

Mientras se encuentra el editor, podrá moverse por la escena usando las teclas direccionales o manteniendo la tecla "espacio" y arrastrándose con el ratón. También, podrá acercar o alejar la imagen con las teclas "+" y "-".

A continuación deberá guardar el objeto en el fichero de la escena mediante el siguiente comando:

```
save
```

Finalmente, para salir del editor una vez guardados los cambios realizados en la escena inserte este comando:

```
quit
```

A.5 Prueba del juego

Para probar el juego que ha desarrollado, deberá usar el ejecutable del juego generado previamente indicándole la escena de la que partir, "scene.bin por defecto.

Debería observar la misma ventana que en editor, sin embargo, en el juego no podrá mover, acercar o alejar la imagen.

A.6 Seguir aprendiendo

Para seguir aprendiendo sobre las distintas funcionalidades que se le pueden otorgar a los objetos del juego puede referenciarse en la demostración técnica de la carpeta "test/demo0" y generar la documentación del motor gráfico usando el programa *doxygen* en la carpeta "docs" del proyecto. Tras esto, debería encontrar dos carpetas dento de "docs/output", una llamada "html" con la documentación den formato HTML (abierto mediante el fichero "index.html"), y la otra "latex" con un *Makefile* para generar un fichero PDF con la documentación.

Lista de acrónimos

AABB Axis Aligned Bounding Box. iv, 39, 41

Allegro Atari Low Level Game Routines. 28, 29

API Application Programming Interface. 26–29, 33, 34, 36, 56, 66

DRM Direct Rendering Manager. 34

KMS Kernel Mode Setting. 34

ROM Read Only Memory. 34

RTTI Run Time Type Information. 66

SDL Simple DirectMedia Layer. 1, 27–29, 33, 34, 36, 37, 54, 56, 64–68, 70

SFML Simple Fast Multimedia Library. 27, 28

UART Universal Asynchronous Receiver-Transmitter. 20

Glosario

bitmap Estructura o fichero de datos que representa una rejilla rectangular de píxeles o puntos de color, denominada matriz, que se puede visualizar en un monitor, papel u otro dispositivo de representación.. 18, 26

bucle principal En los videojuegos, se entiende como bucle principal la repetición del todos los procesos necesarios para su funcionamiento hasta su cese.. 12, 29

búfer En informática, un búfer de datos (o simplemente búfer) es una región de una memoria utilizada para almacenar temporalmente datos mientras se trasladan de un lugar a otro.. 31, 55, 56

cadena de herramientas Conjunto de programas informáticos (herramientas) que se usan para crear un determinado producto (normalmente otro programa o sistema informático).. 17, 19, 21, 22

camino crítico Secuencia de los elementos terminales de la red de proyectos con la mayor duración entre ellos, determinando el tiempo más corto en el que es posible completar el proyecto.. 15

Down Casting Acción de pasar una referencia de una clase base a una de sus clases derivadas.. 26, 66

Duck typing En los lenguajes de programación orientados a objetos, se conoce como duck typing o tipado pato el estilo de tipificación dinámica de datos en que el conjunto actual de métodos y propiedades determina la validez semántica, en vez de que lo hagan la herencia de una clase en particular o la implementación de una interfaz específica.. 25

endianidad Formato en el que se almacenan los datos de más de un byte en un ordenador.. 55, 56

entorno de escritorio Conjunto de software para ofrecer al usuario de una computadora una interacción amigable y cómoda mediante ventanas.. 34, 37, 64

flag En programación, la bandera o flag se refiere a uno o más bits que se utilizan para almacenar un valor binario o código que tiene asignado un significado.. 37

framebuffer Memoria de acceso aleatorio (RAM) dedicada a albergar el color de cada pixel de la pantalla.. 31

hash Función que tiene como entrada un conjunto de elementos, que suelen ser cadenas, y los convierte en un rango de salida finito, normalmente cadenas de longitud fija.. 46

headless Sistema o dispositivo informático que se ha configurado para funcionar sin monitor (la ”cabeza” que falta), teclado ni ratón. Controlado normalmente a través de una conexión de red, serie.. 34, 68

macro En la programación, es el conjunto de comandos que se invocan con una palabra clave, opcionalmente seguidas de parámetros que se utilizan como código literal. Los macros son manejados por el compilador y no por el ejecutable compilado. . 48, 49, 68, 71

port Adaptación de un programa a otra plataforma.. 27, 29

profiling Proceso de recopilar datos detallados sobre el rendimiento de una aplicación o programa con el fin de identificar áreas donde se pueden realizar mejoras y optimizaciones.. 22

renderizar Proceso de generación de una imagen o animación a partir de modelos 3D y texturas utilizando hardware y software especializado.. 12, 13

shader Programa informático que realiza cálculos gráficos escrito en un lenguaje de sombreado que se puede compilar independientemente.. 66

sprite Término para referirse a una imagen bitmap de reducida resolución y paleta de colores en el ámbito del desarrollo de videojuegos 2D. Usualmente forma parte de una imagen más grande compuesta de más sprites (sprite sheet).. ii, 18, 49–54, 64

stream En informática, un stream es una secuencia de elementos de datos disponibles a lo largo del tiempo. Un stream puede concebirse como los elementos de una cinta transportadora que se procesan de uno en uno en lugar de en grandes lotes.. 55, 56

texturas Imagen bidimensional que se aplica a una superficie tridimensional para mejorar su apariencia visual.. 13

tile Una tesela o tile es la parte gráfica de cada videojuego que puede ser utilizada para completar partes de un fondo por medio de un set de teselas o tileset.. 54

widget Un elemento de control gráfico (widget GUI) es una parte de una interfaz gráfica de usuario (GUI) que permite a un usuario de ordenador controlar una aplicación de software.. 67

Bibliografía

[1] "Vídeo sobre la historia del comienzo de los videojuegos." [En línea]. Disponible en: https://youtu.be/LlTldKFhwEA?si=yCTETG-PIK0sBRuK

[2] "Vídeo sobre la historia de Atari." [En línea]. Disponible en: https://youtu.be/7F76o9jLpGE?si=J57E6uOXyQ23x22C

[3] "Vídeo sobre la historia de Nintendo (cap. 1)." [En línea]. Disponible en: https://youtu.be/8vAKPKnzwew?si=7GuWuKWVAe3iNevL

[4] "Vídeo sobre la historia de Nintendo (cap. 2)." [En línea]. Disponible en: https://youtu.be/gmFR_3NiEJs?si=PrGMSB2jsf-MRuO1

[5] "Vídeo sobre la historia de los videojuegos y los primeros ordenadores personales." [En línea]. Disponible en: https://youtu.be/9KAj7_vgXeg?si=6YjwkaGF35hekOZA

[6] "Vídeo sobre la historia de ID Software." [En línea]. Disponible en: https://youtu.be/ckNsQxpSGaY?si=Gy1LcPh01wlHQ4bt

[7] D. Eberly, *3D Game Engine Design: A Practical Approach to Real-time Computer Graphics*. Morgan Kaufmann, 2007.

[8] J. Schell, *The Art of Game Design: A book of lenses*. Taylor & Francis, 2008.

[9] "Guía salarial del sector de tecnologías de la información en galicia entre 2015 y 2016." [En línea]. Disponible en: https://es.scribd.com/document/288511179/Guia-Salarial-Sector-TI-Galicia-2015-2016#

[10] "Información sobre el kit de desarrollo dragonboard 810." [En línea]. Disponible en: https://www.lantronix.com/products/dragonboard-810-development-kit/#undefined

[11] "Información sobre la Raspberry Pi 1 model B." [En línea]. Disponible en: https://raspberry-projects.com/pi/category/pi-hardware/raspberry-pi-model-b

[12] "Información sobre la Raspberry Pi 3 model B." [En línea]. Disponible en: https://www.raspberrypi.com/products/raspberry-pi-3-model-b

[13] "Información sobre la Rock Pi 4." [En línea]. Disponible en: https://rockpi.org/rockpi4

[14] "Información sobre la biblioteca SDL." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Simple_DirectMedia_Layer

[15] "Videojuego deponia." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Deponia_(video_game)

[16] "Videojuego super mario 64 hecho de cero con sdl." [En línea]. Disponible en: https://github.com/sm64-port/sm64-port

[17] "Motor gráfico de half life 1." [En línea]. Disponible en: https://github.com/ValveSoftware/halflife/tree/master

[18] "Repositorio de la biblioteca SGLIB." [En línea]. Disponible en: https://github.com/Marian-Vittek/sglib

[19] "Repositorio de la biblioteca Simple SDL2 Audio." [En línea]. Disponible en: https://github.com/jakebesworth/Simple-SDL2-Audio

[20] R. Nystrom, *Game Programming Patterns*. Genever | Benning, 2014. [En línea]. Disponible en: https://gameprogrammingpatterns.com

[21] "Eventos de los objetos en el motor game maker studio 2." [En línea]. Disponible en: https://manual.yoyogames.com/#t=Quick_Start_Guide%2FObjects_And_Instances.htm

[22] "Técnica del algoritmo del pintor." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Painter's_algorithm

[23] "Técnica del z-buffer." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Z-buffering

[24] "Información sobre la gestión de la memoria basada en regiones." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Region-based_memory_management

[25] "Manual para la compilación cruzada de SDL en una Raspberry Pi." [En línea]. Disponible en: https://github.com/libsdl-org/SDL/blob/main/docs/README-raspberrypi.md

[26] "Información sobre el subsitema de renderizado kms." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Direct_Rendering_Manager#Kernel_mode_setting

[27] "Información sobre el subsitema de renderizado drm." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Direct_Rendering_Manager

[28] "Manual para hacer instalar Raspbian en una Raspberry Pi." [En línea]. Disponible en: https://www.raspberrypi.com/documentation/computers/getting-started.html

[29] "Información sobre el patrón del bucle de eventos." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Event_loop

[30] "Información sobre el paradigma de cola de mensajería." [En línea]. Disponible en: https://en.wikipedia.org/wiki/Message_queue

[31] "Imagen de boot de debian para los procesadores dragonboard." [En línea]. Disponible en: https://releases.linaro.org/96boards/dragonboard845c/linaro/debian/21.12

[32] "Manual de la herramienta adb para android." [En línea]. Disponible en: https://developer.android.com/tools/adb

[33] "Manual de la herramienta fastboot para android." [En línea]. Disponible en: https://source.android.com/docs/setup/build/running?hl=es-419

[34] "Portal del motor de físicas bidimensionales box2d." [En línea]. Disponible en: https://box2d.org

[35] "Portal con las distintas bibliotecas a usar junto a sdl." [En línea]. Disponible en: https://wiki.libsdl.org/SDL2/Libraries

[36] "Repositorio de la biblioteca SDL." [En línea]. Disponible en: https://github.com/libsdl-org/SDL/tree/main

[37] "Manual de referencia de los lenguajes de programción C y C++." [En línea]. Disponible en: https://en.cppreference.com/w

[38] "Manual para usar la herramienta Make." [En línea]. Disponible en: https://www.gnu.org/software/make/manual/make.html

[39] "Manual para usar la herramienta Valgring." [En línea]. Disponible en: https://valgrind.org/docs/manual/manual.html

[40] "Portal de la biblioteca de python pygame para el desarrollo de videojuegos." [En línea]. Disponible en: https://www.pygame.org/

[41] "Manual de la API del motor gráfico unity." [En línea]. Disponible en: https://docs.unity3d.com/ScriptReference/index.html

[42] "Manual de la API del motor gráfico game maker studio 2." [En línea]. Disponible en: https://manual.yoyogames.com/#t=Content.htm

[43] "Biblioteca SDL." [En línea]. Disponible en: https://www.libsdl.org

[44] "Wiki de la biblioteca SDL." [En línea]. Disponible en: https://wiki.libsdl.org/SDL2/FrontPage

[45] "Tutoriales para usar la biblioteca SDL con C++." [En línea]. Disponible en: https://lazyfoo.net/tutorials/SDL

[46] "Tutoriales para usar la biblioteca SDL con C++." [En línea]. Disponible en: https://www.sdltutorials.com