Desarrollo de videojuegos de plataformas en 2D usando Simple DirectMedia Layer

Aarón Ojeda Reyes

Tutor: Agustín Trujillo Pino

Mayo 2017

Agradecimientos

A mi tutor, por su supervisión y disponibilidad a lo largo del proyecto.

A mis compañeros en esta aventura que ha supuesto finalizar la carrera. Nada hubiera sido igual sin su compañía, bromas y consejos. En especial a Yarilo, David, Yeray y Luis, con los que la relación va más allá de lo académico.

No puedo olvidar al resto de amigos fuera de la carrera. Gracias por interesarse por algo que a duras penas entendían y por obligarme a salir de «la cueva» de vez en cuando.

A mi madre, a mi padre y a mi hermano, por el inmenso apoyo desinteresado que sólo la familia sabe proporcionar.

A Itahisa, cuya paciencia y comprensión durante todo este proceso no han conocido límites.

Índice general

Parte I

Introducción

Capítulo 1

Descripción

Los juegos de plataformas (o simplemente «plataformas») son un género de videojuegos en el que el desafío principal radica en mover un avatar a través de las plataformas de un escenario, normalmente haciendo uso del salto. Dentro del género, hay un juego que brilla con luz propia y que sirve a la perfección para ilustrar más concretamente el tipo de videojuegos que han motivado la creación de este proyecto: Super Mario Bros. (Nintendo, 1985).

A principios de los años 80, la saturación del mercado con productos de muy mala calidad acabó originando la mayor recesión que ha vivido la industria del videojuego en occidente, que cuantificada se traduce en una reducción de beneficios en Norteamérica del 97 % entre 1983 y 1985[1]. Esta crisis es conocida como «el crash del 83», un golpe del que la industria se empezaría a recuperar con el lanzamiento a finales de 1985 de la consola Nintendo Entertainment System (NES) y su juego estrella Super Mario Bros. El resto es historia: con más de 40 millones de copias vendidas en su versión para NES, el juego se convirtió en el más vendido de la historia durante 24 años[2].

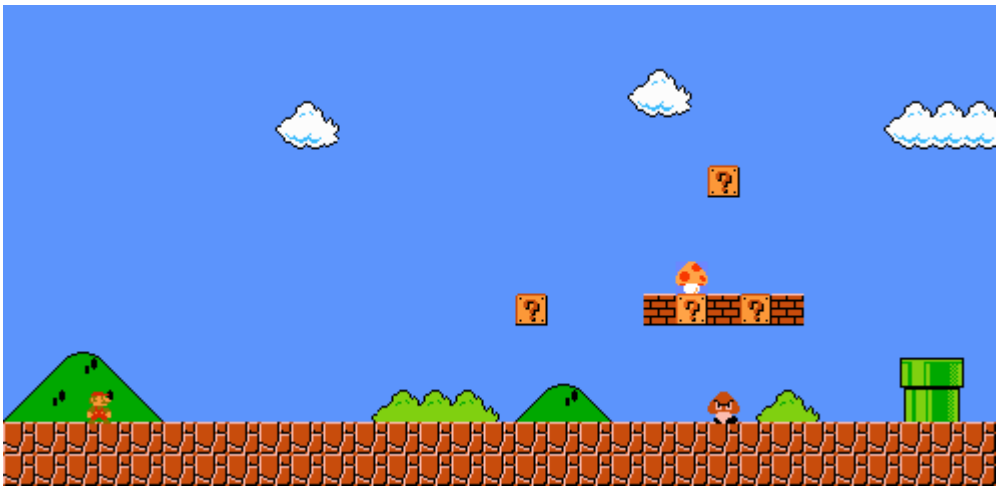


Figura 1.1 – Comienzo del nivel 1-1 de Super Mario Bros.

Ya desde el comienzo del primer nivel el juego pone las cartas sobre la mesa, con una pantalla que es en sí misma un excelente tutorial, pues de un plumazo nos enseña a interactuar con los elementos más característicos del juego: avance lateral, salto, enemigos, bloques, puntuación, power-ups y tuberías.

La decisión de hablar de Super Mario Bros. en esta introducción no ha sido casual, pues se basa en la firme creencia de que sin la existencia de dicho juego este PFC no existiría como tal, no ya porque muchos lo consideren el salvador de la industria del videojuego en occidente, sino porque el concepto de juego de plataformas que propone caló tan hondo en el sector que definió los estándares por los que se regirían los juegos de plataformas hasta hoy.

[1] De 3200 millones de dólares a 100 millones.

[2] En 2009 Wii Sports estableció un nuevo record con 60 millones de copias.

A lo largo de la primera mitad de este proyecto se estudiarán los detalles técnicos más importantes de la creación de un juego de este tipo, dando lugar a la creación de un *framework* para el desarrollo de juegos en 2D, programado haciendo uso del lenguaje de programación C++ y la biblioteca multiplataforma de bajo nivel *Simple DirectMedia Layer*, más conocida por sus siglas SDL. A esta utilidad se le dará el nombre de S2Dp, acrónimo de *SDL 2D Platformers*.

En la segunda mitad del proyecto se partirá de donde lo dejó la primera, y se probará la versatilidad de s2Dp creando un editor de niveles con interfaz gráfica de usuario, que nos permitirá crear juegos de plataformas de apariencia y desarrollo variable. Para la creación de la GUI se hará uso nuevamente del lenguaje C++, en este caso acompañado de la biblioteca Qt. Esta herramienta será el resultado final del PFC a nivel de software y se denominará S2PEditor (*SDL 2D Platformers Editor*).

Capítulo 2

Objetivos

Este proyecto se considerará exitoso si se cumplen los tres objetivos siguientes:

1. Comprender las etapas de que consta el desarrollo a bajo nivel de un videojuego en dos dimensiones, adquiriendo durante el proceso destreza en la programación con C++ y SDL, las herramientas finalmente elegidas para el desarrollo.

2. Construir un conjunto de clases y funciones reutilizables que conformen un framework orientado al desarrollo de videojuegos en dos dimensiones.

3. Haciendo uso de la herramienta resultante del punto anterior, desarrollar un editor de niveles gráfico que permita crear juegos de plataformas sencillos.

4. Desarrollar un pequeño videojuego de ejemplo haciendo uso del editor, para demostrar su utilidad.

Capítulo 3

Estructura del documento

En este capítulo se describirá la estructura de esta memoria, explicando primero la estructura general y haciendo después un recorrido por los capítulos del documento de aquí en adelante.

Como se ha comentado en los capítulos anteriores, este Proyecto de Fin de Carrera consta de dos grandes partes bien diferenciadas: el framework de desarrollo de juegos en dos dimensiones **s2Dp** y el software **S2PEditor** que permite crear juegos y editar sus niveles gráficamente haciendo uso de dicho framework. Por tanto, se ha optado por dividir la memoria en cuatro *partes*, una para cada uno de los dos desarrollos, más la introducción y la conclusiones. A continuación se describen cada una de estas partes y los capítulos que contienen.

3.1. Parte I Introdución

En esta parte se introducirá el proyecto al lector, sin entrar a comentar los dos desarrollos principales. El capítulo que nos ocupa y los dos previos pertenecen a la introducción, a la que se añaden dos más.

Capítulo 4 Planificación

Antes de empezar el proyecto se realizó una previsión de las horas que llevaría completarlo y de su distribución según tareas. En este capítulo se realizará una comparación entre esa primera previsión, presentada en el documento PFC-1, y la cantidad y reparto de horas finales.

Capítulo 5 Estado del arte

Aquí se hace un recorrido por el nacimiento y evolución de los juegos de plataformas en 2D, para continuar con un repaso a los lenguajes de programación y herramientas más usadas actualmente en el desarrollo de videojuegos.

3.2. Parte II Framework s2Dp

Esta es la primera de las dos grandes partes del PFC, centrada en el desarrollo del conjunto de clases que forman el framework s2Dp para el desarrollo de plataformas 2D.

Capítulo 6 Análisis

En base a un análisis de requisitos, y a lo estudiado en el estado del arte, se acotarán a tres las posibles herramientas para el desarrollo del framework. En la última sección del capítulo se delimitará el alcance de los juegos que podremos crear con la versión básica de s2Dp.

Capítulo 7 Comparativa de herramientas

Aquí se realizará una extensa comparativa entre las tres herramientas seleccionadas en el punto anterior, evaluando para cada una las plataformas que soporta, su facilidad de uso, su presencia en la industria y la actividad de su comunidad. De este estudio saldrá la combinación de lenguaje de programación más biblioteca finalmente elegida: C++ y SDL 2.0.

Capítulo 8 Diseño e Implementación

Éste es probablemente el capítulo al que más importancia se le ha dado de la memoria, y sin duda el más relevante a nivel técnico.

Se comenzará con una introducción a los distintos subsistemas y extensiones de la biblioteca SDL.

A continuación, a lo largo del resto de secciones del capítulo se irá explicando la estructura del código del framework, a través de las clases más importantes y las relaciones entre ellas. Además se profundizará en dichas clases a nivel de código, comentando la implementación de los métodos que se han considerado más relevantes.

En los diagramas que se muestran no está la totalidad de métodos y atributos de cada clase; sólo se presentaron los miembros más ilustrativos para comprender el diseño, ocultando los que no aportaban información de interés para entender la estructura general. Por ejemplo, se dejaron fuera de los diagramas de clase métodos auxiliares privados con funciones muy concretas, o los *getters* y *setters* que tienen algunas clases para obtener y modificar sus atributos.

Por su parte, en la implementación también hubo que filtrar qué explicar y qué no, quedando gran parte del código fuera del documento. Incluso en los métodos comentados, existen diferencias intencionadas entre el código que se muestra y el real: se han omitido funciones auxiliares integrando su código directamente en la función que las llamaba, se ha simplificado o eliminado el reporte de errores, se han omitido las comprobaciones matemáticas para la gestión de la cámara y otros aspectos, etc.

En muchas ocasiones la decisión de omitir partes importantes del diseño y sobretodo del código supuso un dilema, pues preocupaba que esto pudiera causar la impresión de que la amplitud y complejidad del desarrollo fue menor al real, pero era inviable comentarlo todo, tanto a nivel de tiempo como de espacio.

3.3. Parte III Editor de niveles S2PEditor

En esta parte de la memoria se explicará el segundo desarrollo del PFC: la construcción de S2PEditor, un software de escritorio con GUI que nos permita editar los niveles de un juego que se ejecute mediante s2Dp de forma transparente al usuario.

Capítulo 9 Análisis

En la primera sección del capítulo se capturarán los requisitos funcionales del sistema a partir de la técnica de casos de uso. En la segunda y última sección se estudiarán los enfoques posibles

para construir la interfaz del editor, resaltando las ventajas e inconvenientes de cada uno, para finalmente concluir que se integrará SDL en una biblioteca GUI existente.

Capítulo 10 Comparativa de bibliotecas GUI

En este capítulo se comenzará comparando bibliotecas GUI ligeras teóricamente compatibles con SDL, que acabarán siendo descartadas en favor del framework Qt. Se introducirá Qt y se explicará cómo compatibilizarla con las herramientas que se estaban utilizando hasta el momento: el IDE Microsoft Visual Studio y, especialmente, la biblioteca SDL.

Capítulo 11 Diseño e Implementación

Este capítulo es el más importante de la parte de la memoria dedicada al desarrollo de la herramienta S2PEditor. Se comenzará presentando un boceto de la GUI, para pasar a explicar el diseño y la implementación de los elementos más importantes de la solución adoptada.

Aunque finalmente se dedicó el mismo tiempo o más a este desarrollo que al del framework (no podemos saberlo con exactitud porque no son dos desarrollos completamente aislados, uno influye en el otro y viceversa, por lo que se solapaban constantemente), no se perseguirá en este capítulo el nivel de detalle alcanzado en el Capítulo 8 con s2Dp. El desarrollo del framework se considera más interesante desde el punto de vista de los objetivos del proyecto, no en vano es un PFC sobre desarrollo de videojuegos, y en S2PEditor se pasó mucho más tiempo con detalles relativos a programar una GUI. Esta no fue la única razón; el otro motivo es que la extensión de la documentación y el tiempo dedicado al proyecto habían superado ya con creces el estipulado, como se verá en el Capítulo 4.

No obstante, sí que se profundizó en la clase construida para integrar SDL y Qt, ya que esta tarea, compatibilizar ambas herramientas, fue el aspecto técnico que más tiempo y estudio llevó de todo el proyecto.

Capítulo 12 Desarrollo de un juego con S2PEditor

En la forma de un tutorial para construir un sencillo juego de tres niveles, se describe extensamente cómo utilizar todas las opciones de S2PEditor. Éste es el capítulo que debe consultarse si quiere tenerse una idea de las posibilidades del editor, pues el «Manual de usuario» presentado como anexo es simplemente una referencia esquemática con las opciones de la interfaz.

3.4. Parte IV Conclusiones

Aquí se agrupan, a modo de conclusiones, los capítulos posteriores a la finalización de los dos desarrollos principales.

Capítulo 13 Costes

Se hará un cálculo de las horas dedicadas a la construcción de S2Dp más S2PEditor y se multiplicará por el precio estipulado por hora.

Capítulo 14 Valoración

Este capítulo incluye una valoración objetiva del cumplimiento de los objetivos del proyecto, y una valoración subjetiva sobre el aprovechamiento del mismo a nivel personal.

Capítulo 15 Trabajo futuro

Se comentará de forma general el inmenso margen de mejora que tiene el proyecto como herramienta útil para el desarrollo de juegos, enumerando algunos posibles añadidos concretos.

Capítulo 4

Planificación

Como planificación inicial del proyecto se propuso un plan de trabajo compuesto de un total de 950 horas, distribuidas por tareas como sigue:

Fase/Tarea	**Horas**
Análisis y estudio	
Estudio previo del PFC	100
Estudio de las herramientas	140
Generación de la documentación	60
Total	300
Diseño	
Framework para el desarrollo de videojuegos en 2D	50
Editor de niveles	50
Generación de la documentación	20
Total	120
Implementación	
Framework para el desarrollo de videojuegos en 2D	165
Editor de niveles	165
Generación de la documentación	50
Total	380
Validación y resultados	
Test de validación	30
Desarrollo de un juego de ejemplo	50
Total	80
Preparación de la presentación del proyecto	
Total	70

Para compararla con el tiempo final, a esta planificación inicial habría que restarle las 70 horas de la preparación de la presentación del proyecto, ya que es algo que se hará posteriormente al desarrollo y a la generación de la documentación, por lo que se nos queda un plan inicial de trabajo de **880 horas**.

A lo largo del desarrollo del PFC, se ha sido estricto con el recuento de horas, apuntando al final de cada sesión de trabajo cuántas horas se han invertido y en qué. El reparto de horas por tarea no será completamente exacto, pues muchas veces saltamos en medio de una sesión de una tarea a otra y cortar el flujo del trabajo para ser minuciosos con el minutaje sería contraproducente, pero se ha intentado que sea lo más cercano posible a la realidad. Para el recuento se usó principalmente la herramienta *Tomighty*[1].

Es complicado realizar la comparación con la planificación inicial, puesto que el desconocimiento al inicio del proyecto llevó a cierta imprecisión al definir las tareas, no previéndose algunas de

[1] http://tomighty.org/

ellas y atomizándolas demasiado, cuando en la realidad se ha visto que el nivel de acoplamiento entre tareas en este proyecto ha sido muy alto.

Por ejemplo, no se previó que la construcción del editor implicaría muchas horas de estudio para comprender cómo usar del framework Qt y cómo integrarlo con SDL. Asimismo, viéndolo ahora resulta casi ingenuo asignar 30 horas a test de validación, siendo el *game testing* y la consiguiente detección de *bugs* un campo del desarrollo de juegos altamente técnico que no ha dado tiempo a tocar.

Por otra parte, en el recuento de horas real se ha sido mucho más específico, apuntando muchísimas más tareas de las planeadas. No obstante, se intentarán agrupar en fases análogas a las de la planificación inicial, para poder llevar a cabo la susodicha comparación. El reparto final de horas ha sido el siguiente:

Fase/Tarea	**Horas**
Análisis y estudio	
Bibliografía variada	73
Estudio de herramientas	206
Documentación	145
Total	424
Diseño	
Framework S2DP	78
Editor de niveles S2PEDITOR	92
Documentación	70
Total	240
Implementación	
Framework S2DP	244
Editor de niveles S2PEDITOR	297
Documentación	64
Total	605
Desarrollo de un juego de ejemplo	
Total	77

Estos tiempos suman un total de **1346 horas**. A continuación se compararán las horas previstas y las invertidas. Para que la comparación sea más fiable, se eliminarán de la previsión las tareas que no se han llevado a cabo (test de validación y preparación de la presentación), y agruparemos el resto de las tareas en cinco áreas comunes a ambas:

Tarea	**Horas previstas**	**Horas invertidas**	**Aumento (%)**
Análisis y estudio	240	279	16
Diseño	100	170	70
Implementación	330	541	64
Documentación	130	279	115
Juego de ejemplo	50	77	54
Total	**850**	**1346**	**58**

En todas las áreas el tiempo invertido supera al previsto. El apartado de análisis y estudio bibliográfico, que incluye el aprendizaje de las herramientas utilizadas durante el proyecto, es en el que la planificación ha sido más acertada, siendo el tiempo final sólo un 16 % mayor al previsto. En el resto de apartados las horas previstas son muy inferiores a las invertidas, con

aumentos que varían desde el 54 % al 115 %. Este último caso es el de la documentación, para la que se ha invertido más del doble del tiempo inicialmente planeado.

Con respecto al número total de horas, vemos que lo que en el plan de trabajo inicial se cifraba en 850 horas se ha ido a las 1346, lo que se traduce en un aumento del 58 %. Conociendo los datos, salta a la vista que la planificación inicial fue un fracaso en términos de horas totales. Las dos razones más evidentes son las siguientes:

- Se subestimó enormemente el tiempo que consumiría generar la documentación.
- No se supo prever la complejidad del editor gráfico de niveles y el impacto que tendría en el framework. Esto hizo que prácticamente todos los apartados del proyecto se magnificaran, en especial el Diseño y la Implementación.

Se tomará nota de los fallos cometidos con el objetivo de llevar a cabo una mejor planificación en futuros proyectos.

Dejando a un lado las horas totales, si nos fijamos en la distribución porcentual de horas por tarea (figuras 4.1 y 4.2), vemos que la prevista no dista demasiado de la final. En la planificación inicial se le dio más importancia a «Análisis y estudio» de la que finalmente acabó teniendo, mientras que con el área de «Documentación» sucedió al revés. Sin embargo, en las áreas de «Diseño», «Implementación» y «Juego de ejemplo» la diferencia entre previsión y resultados ha sido mínima. Podemos concluir por tanto que la previsión de distribución de horas por tarea ha sido acertada.

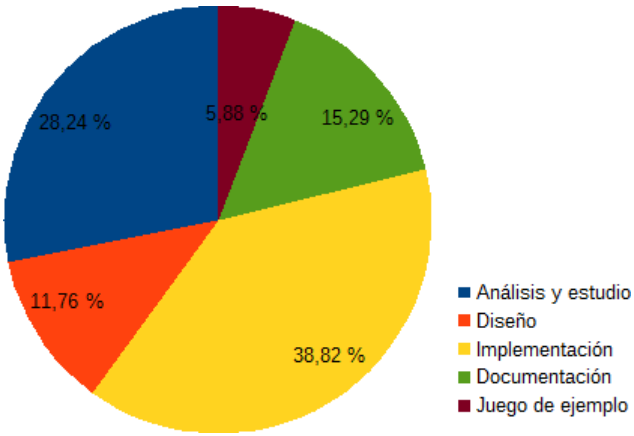


Figura 4.1 – Distribución prevista.

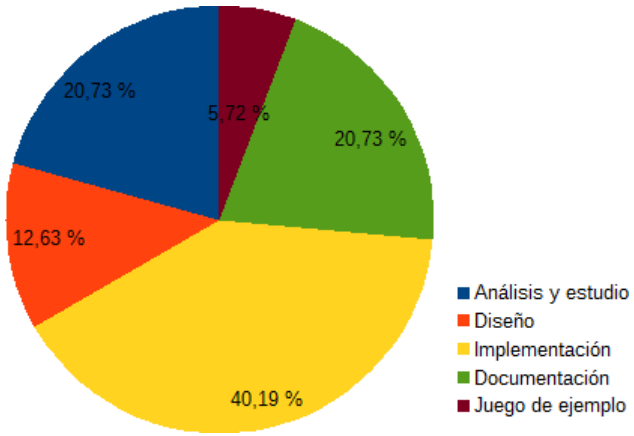


Figura 4.2 – Distribución final.

Capítulo 5

Estado del arte

5.1. Juegos de plataformas

No existe consenso sobre cuál es el primer juego de plataformas de la historia, pero cuando se habla sobre los juegos que pusieron los cimientos de este género, dos nombres destacan por encima del resto: los arcades Space Panic (1980, Universal) y Donkey Kong (1981, Nintendo).

Space Panic, más que el primer juego del género suele ser considerado el antecesor. Ya Crish Crawford[1] se refirió a él como «el abuelo de los juegos de plataformas», pues aunque el protagonista no tenía la posibilidad de saltar, podía moverse por las plataformas del escenario con relativa libertad y se incluían conceptos como la gravedad o el movimiento utilizando escaleras.

Si Space Panic es el abuelo del género, Donkey Kong es el padre; éste sí que incluía el salto y lo usaba como su mecánica principal, hecho que quedaba claro con ver el nombre del protagonista: el carpintero Jump Man, que años después cambiaría de oficio y sería Mario, el fontanero que todos conocemos. El objetivo del juego era simple: llegar a lo más alto de la pantalla esquivando los barriles lanzados por Donkey y otros peligros para salvar a nuestra amada Pauline. El juego supuso un éxito sin precedentes en el género y puso sobre la mesa los nombres de Shigeru Miyamoto y Nintendo, diseñador y compañía imprescindibles para entender la historia de los juegos de plataformas.

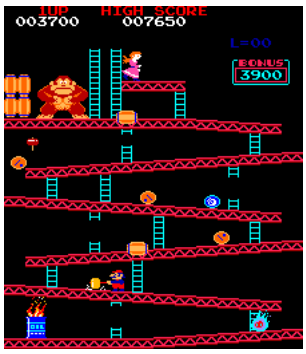


Figura 5.1 – Primera pantalla de Donkey Kong.

Poco después de Donkey Kong salió Jump Bug (1981, Rock-ola), destacable por ser el primer juego del género en incorporar scroll horizontal, pero habría que esperar hasta el lanzamiento de Pitfall! (1982, Activision) para ver el siguiente paso en la evolución del género. Pitfall!, disponible para Atari 2600, se alejaba de los clásicos arcade *single-screen* y presentaba una propuesta más cercana a la aventura aprovechando el tirón de películas como «En busca del arca perdida», permitiéndonos recorrer una peligrosa selva avanzando entre pantallas individuales conectadas entre sí.

Un año después de la salida del Pitfall, la saturación en la industria haría que acabara sumida en la mayor crisis que ha vivido hasta la fecha, conocida como «el crash del 83». Dos años después, a finales de 1985, de la mano de los responsables de Donkey Kong saldría en Japón y USA la NES con su juego estrella, Super Mario Bros. (1985, Nintendo), que además de iniciar la recuperación de la industria, sentó las bases de un género que dominaría el mercado de las consolas durante dos generaciones, llegando a su cénit en la generación de los 16 bits.

Fuera de las consolas, el género también iba avanzando como prueban el éxito arcade Bubble

[1] Diseñador de videojuegos y escritor, creador de lo que hoy conocemos como la Game Developers Conference.

Bobble (1986, Taito) o Prince of Persia (1989, Brøderbund) para Apple II, que tenían como fuerte el modo cooperativo y el realismo, respectivamente.

La llamadad «edad de oro» de los plataformas llegaría a principios de los noventa, con las consolas de 16 bits de Nintendo y Sega: Super Nintendo y Mega Drive, respectivamente. La cantidad de juegos de plataformas de calidad que recibieron estas consolas fue apabullante, pero no podemos dejar de nombrar Super Mario World (1990, Nintendo) y Sonic the Hedgehog (1991, Sega), por ser los buques insignia de sus respectivas consolas en sus comienzos, y los juegos que hicieron a muchos decidirse por una u otra.

Figura 5.2 – Super Mario World (1990)

Figura 5.3 – Sonic the Hedgehog (1991)

Podríamos poner fin al repaso de la edad gloriosa de los plataformas con el broche de oro que fue Donkey Kong Country (1994, Rare). El juego tuvo dos excelentes secuelas, pero su tardía aparición hizo que se solaparan con la nueva generación de consolas, que con su músculo técnico generaron una demanda de gráficos en tres dimensiones y experiencias más realistas,lo que haría que otros géneros como las aventuras en tercera persona, los RPG, los juegos de lucha o los de deportes coparan el mercado, relegando los plataformas en 2D a un ostracismo del que no saldrían hasta mucho después, bien entrado el nuevo milenio.

Sin duda pueden nombrarse ejemplos de plataformas 2D en la generación de 32 bits, ya sea puros como Rayman (1995, Ubisoft) o pseudo-3D (desarrollo 2D y fondos 3D) como Pandemonium! (1996, Crystal Dynamics) o Klonoa: Door to Phantomile (1997, Namco) , pero era innegable que el usuario estaba sediento de las novedosas 3D y este tipo de títulos no eran igual de rentables que en el pasado.

Pasadas unas décadas, y con el auge de la programación de videojuegos independientes, los plataformas 2D empezaban su segunda juventud. Juegos gratuitos para PC como Knytt Stories (2007, Nifflas) o I Wanna Be The Guy (2007, Kayin) encontraban sus legiones de fans, y los amantes de los plataformas encontraron su hogar en la escena *indie*. No tardó en vérsele el filón comercial a esto, y la posibilidad de mercado digital que brindaba la conexión a Internet de las consolas fue una oportunidad perfecta para que desarrollos modestos con un equipo y presupuesto alejados de las grandes producciones encontraran su hueco en el mercado de videojuegos tradicional.

Hubo muchos juegos en formato digital que contribuyeron al renacer del género y cada uno tendrá los que considere más importantes, pero se destacarán tres por el nivel de calidad alcanzado

en sus propuestas, todas muy diferentes: Braid (Number None, 2008), Super Meat Boy (Team Meat, 2010) y FEZ (Polytron, 2012). No en vano estos tres juegos inspiran la película documental Indie Game: The Movie (2012, James Swirsky & Lisanne Pajot), que relata las peripecias de sus creadores en el proceso de desarrollo y lanzamiento al mercado de sus productos. De ellos, sólo Super Meat Boy era un plataformas 2D clásico, pues Braid introduce elementos de interacción con el tiempo y FEZ precisamente ironiza con las tres dimensiones, planteando un mundo cúbico por el que nos podemos mover moviendo los cuatro laterales, cada uno de ellos bidimensional.

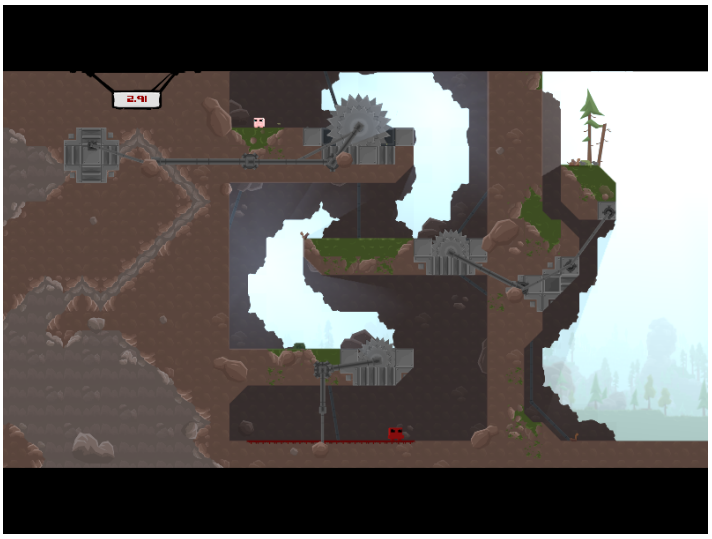


Figura 5.4 – Super Meat Boy (2010)

Fueran éstos u otros los juegos que supusieron el renacimiento de los plataformas 2D, lo que estaba claro era que habían vuelto para quedarse, como se ha demostrado en los últimos años con una realmente abrumadora cantidad de títulos de estas características en todas las plataformas digitales. Además, no ajenas a lo que está ocurriendo, las empresas tradicionales del sector se han sumado a la iniciativa relanzando sus clásicos del género o incluso atreviéndose con algún nuevo desarrollo para consolas de sobremesa, como ya hicieran Ubisoft con Rayman: Origins (2011) o Nintendo con New Super Mario Bros. Wii (2009).

No sabemos qué deparará el futuro, pero la reducción del coste de un equipo para desarrollar un videojuego sumada al surgimiento de todo tipo de herramientas software para facilitar esta tarea, parecen tener asegurada la presencia en el panorama (comercial o independiente) no sólo de los plataformas 2D, sino de cualquier género.

5.2. Herramientas de desarrollo

En la actualidad existe una gran cantidad de herramientas para el desarrollo de videojuegos, desde bibliotecas multimedia de propósito general hasta complejos motores específicos, pasando por *frameworks* a medio camino entre los dos. Sería incorrecto afirmar con rotundidad que una opción es «la mejor» en términos absolutos, pues depende de las prioridades y exigencias que el desarrollo presente, así como de las circunstancias en que se lleve a cabo.

Es inabordable abarcar todas las tecnologías disponibles con nuestro estudio, por lo se realizará un primer cribado basándose en los lenguajes de programación más demandados en el mercado laboral. Realizando una consulta en el buscador Google por los términos *game development job*, los primeros sitios que aparecen son Gamasutra[1] y Linkedin[2]. Si se contabiliza qué lenguaje o lenguajes de programación se demandan en las 25 primeras ofertas de empleo en cada uno de estos dos sitios, se obtienen los datos representados en el Cuadro 5.1.

Lenguaje	Linkedin	Gamasutra	Total
C++	16	15	31
C#	7	8	15
Java	6	8	14
C	5	7	12
AS3	3	5	8
JavaScript	1	4	5
Python	1	4	5
PHP	2	1	3
Lua	1	2	3
Objective C/C++	1	2	3
HTML5	0	2	2
Scala	1	0	1

Cuadro 5.1 – Lenguajes más demandados en las 25 primeras ofertas de cada sitio.

Hay que tener en cuenta además que esta búsqueda se ha hecho tratando a todas las ofertas por igual, sin importar si se buscan desarrolladores para dispositivos móviles, juegos integrados en redes sociales o juegos web en general. Esto hace que AS3, Java o incluso PHP suban en la lista. Si estos tipos de oferta se hubieran excluido, centrándose en juegos para escritorio y consolas, el dominio de lenguajes como C++, C o C# sería aún mayor.

Por supuesto esta breve búsqueda no debería servirse únicamente para tomar una decisión, pero complementada con la información de sitios web de programación como gamedev o game stack echange, y haciendo un repaso por los lenguajes utilizados en la programación de videojuegos profesional e independiente, refuerza la idea de que C# y sobretodo C++[2] serán los lenguajes que se consideren para la realización del proyecto. A continuación se describirán algunas de las herramientas más utilizadas para desarrollar videojuegos que sean compatibles con alguno de estos dos lenguajes.

SDL

Simple DirectMedia Layer es una biblioteca de desarrollo multiplataforma diseñada para proporcionar acceso de bajo nivel al hardware de audio, teclado, ratón, joystick y gráficos vía OpenGL y Direct3D. Es ampliamente utilizada para desarrollo de software de reproducción de vídeo, emulación y videojuegos.

SDL fue desarrollada por Sam Lantinga, que lanzó su primera versión a principios del año 1998, mientras trabajaba para Loki Software. Está escrita en C, funciona de manera nativa en C++

[2]C no se ha considerado porque se buscaba un lenguaje orientado a objetos.

y dispone de *bindings* para muchos otros lenguajes, entre ellos C#. Soporta oficialmente los sistemas operativos Windows, Mac OS X, Linux, iOS y Android.

SDL 2.0, la versión actual, se distribuye bajo la licencia zlib, la cual permite usarla libremente en cualquier software[3].

Allegro

Allegro es una biblioteca multiplataforma principalmente dirigida al desarrollo de videojuegos y aplicaciones multimedia, muy similar a SDL en su funcionalidad y enfoque. Gestiona tareas comunes de bajo nivel como crear ventanas, aceptar entrada de usuario, dibujar imágenes o reproducir sonidos. Fue creada originalmente por Shawn Hargreaves para la Atari ST a principios de la década de los noventa, de ahí su nombre, que viene de las siglas *Atari Low-Level Game Routines*.

Está escrita en C y es utilizable desde C++, existiendo también bindings para otros lenguajes. Su última versión, Allegro 5, está distribuida bajo la licencia zlib y soporta los sistemas operativos Windows, Linux, Mac OSX, iPhone y Android[4].

SFML

Simple and Fast Multimedia Library es una API portable y sencilla de usar, escrita en C++. Como menciona su principal creador, Laurent Gomila, puede entenderse como una SDL orientada a objetos[5]. Al igual que ésta, proporciona una interfaz para acceder a los diferentes componentes del PC y así facilitar el desarrollo de juegos y aplicaciones multimedia. Está dividida en cinco módulos: *system*, *window*, *audio*, *graphics* y *network*.

Como se ha dicho, está implementada en C++, aunque se han desarrollado multitud de bindings para otros lenguajes que permiten usar SFML desde C, C#, C++/CLI, D, Ruby, OCaml, Java, Python y VB.NET. Las versiones 2.1 y posteriores están disponibles para Windows (8, 7, Vista, XP), Linux y Mac OS X, y se espera que pronto se añadan a la lista iOS y Android.

El código fuente de SFML se distribuye bajo los términos de la licencia zlib/libpng[6].

Monogame

Para explicar qué es Monogame, se hace necesario mencionar XNA, un framework desarrollado por Microsoft y basado en .NET 2.0 que proporciona un conjunto de herramientas para facilitar el desarrollo de videojuegos para Windows, Windows Phone y Xbox 360.

Monogame es una implementación libre de la API XNA 4.0 que nace con el objetivo de permitir a los desarrolladores de XNA portar sus trabajos a sistemas iPhone[7]. Actualmente soporta iOS, Android, Mac OS X, Linux, Windows, Windows 8 Store y Windows Phone 8, siendo compatible con todos los compiladores de Mono y con Visual Studio.

C# es el lenguaje utilizado para programar con Monogame. Como cuenta Dean Ellis[7], uno de los integrantes del proyecto, la idea de crear videojuegos complejos utilizando lenguajes interpretados como C# fue considerada durante mucho tiempo descabellada. Sin embargo el

panorama ha cambiado y este tipo de lenguajes han demostrado ser realmente viables para el desarrollo de juegos, gracias en gran parte a XNA y al popular *engine* Unity 3D, los cuales utilizan C# como su principal lenguaje de desarrollo. Recientemente, en 2013, Microsoft anunció que XNA no se seguiría desarrollando ni recibiría más soporte, una razón más (junto a lo comentado en el párrafo anterior) por la que muchos desarrolladores están eligiendo Monogame.

Actualmente el proyecto está liderado por Tom Spilman y Steve Williams[8], y el software puede ser libremente usado (aunque hay que pagar una licencia para los framework para Android y Xamarin iOS) y modificado, al estar cubierto por la licencia MS-PL (Microsoft Permissive License)[9].

Unreal Engine

Desmarcándose bastante de lo anteriormente comentado tenemos Unreal Engine, un motor de videojuegos profesional desarrollado por Epic Games en C++ y mostrado por primera vez en 1998 con Unreal, el juego de disparos en primera persona del que toma el nombre.

Las plataformas soportadas por esta potente herramienta son Windows, PlayStation 4, Xbox One, Mac OS X, iOS, Android, Linux, SteamOS y HTML5.

Desde el 2 marzo de 2015, Unreal Engine 4 es gratuito; no obstante, una vez lanzado un juego o aplicación desarrollado con este motor, hay que pagarle a Epic Games un 5 % de derechos después de alcanzar los 3000$ de beneficio bruto por juego y por cuarto de año[10].

Unity

Unity es un motor de videojuegos desarrollado por Unity Technologies, cuya primera versión se lanzó en junio de 2005. Está escrito usando una combinación de C/C++ para su núcleo y C# para la interfaz de usuario. Por otra parte, los lenguajes en los que podemos programar usando su API son C#, JavaScript y Boo.

El número de plataformas soportadas por Unity es muy amplio, incluyendo iOS, Android, Windows Phone, Windows, Mac, Linux/Steam OS, PlayStation 4, PlayStation Vita, Xbox One, Wii U, Nintendo 3DS, Nintendo Switch, etc[11].

Unity dispone de cuatro licencias: Personal (gratuita), Plus (35$/mes), Pro (125$/mes) y Enterprise (precio adaptado). La licencia Unity Personal cubre hasta los cien mil dólares de beneficios anuales, Unity Plus cubre hasta los doscientos mil y Unity Pro no tiene límite de beneficios.

Otras herramientas

Como se ha dicho, se han tenido sólo en cuenta las herramientas pensadas para C++ y C# por acotar el análisis; no obstante, describiremos muy brevemente algunas otras (aunque no fueran compatibles con estos lenguajes), para reforzar la idea de que la muestra que presentamos es sólo una pequeña parte del total:

Game Maker::Studio Desarrollado por YoYo Games, es especialmente útil para quien carezca de conocimientos de programación. Dispone de una completa GUI con un sistema *Drag*

and Drop que hace su uso muy sencillo, y que en combinación con el uso del lenguaje GameMaker Language pude dar lugar (y lo ha hecho) a juegos comerciales.

CryEngine Potente motor gráfico creado por la desarrolladora de videojuegos Crytek, usado por primera vez con el juego Far Cry (2004, Crytek). Es gratuito para usos no comerciales y educativos.

Construct 2 Editor de juegos en 2D basado en HTML5 y desarrollado por Scirra Ltd. Al igual que Game Maker, está orientado a no programadores y promete la posibilidad de crear un juego avanzado sin escribir una línea de código.

Corona SDK Plataforma de desarrollo de juegos 2D gratuita creada por Corona Labs. El lenguaje que se usa para desarrollar en esta plataforma es Lua. Sus creadores destacan sobretodo la velocidad a la que permite desarrollar, asegurando que es más de diez veces más rápido que con otras plataformas.

Además existen multitud de bibliotecas específicas de cada lenguaje, como puedan ser LibGDX para Java y PyGame para Python, por nombrar algunas de las más conocidas para lenguajes populares en la actualidad.

Parte II

Framework s2Dp

Capítulo 6

Análisis

Pasaremos a analizar las necesidades del primero de los dos desarrollos en que se divide el proyecto. Como se ha descrito muy brevemente en la introducción, esta parte consistirá en el desarrollo de un framework para la creación de juegos, poniendo el foco en los plataformas 2D. Empezaremos estableciendo los requisitos del software y seguidamente se identificará el entorno tecnológico.

6.1. Establecimiento de requisitos

Desarrollar juegos a bajo nivel es un proceso complicado que requiere gran cantidad de tiempo y dedicación para obtener resultados. Un buen conjunto de clases reutilizable permite acelerar el desarrollo y enfocarse en el aspecto creativo de crear un juego, en lugar de estar forcejeando con detalles de implementación. El requisito principal de este primer desarrollo del PFC es, por tanto, que una vez finalizado tengamos ese conjunto de clases, en la forma de un framework que facilite la creación de futuros juegos. A continuación se enumerarán una serie de requisitos más concretos, todos ellos funcionales:

- Gestión de texturas.
- Gestión de sonido: música y efectos de sonido.
- Gestión de eventos de entrada.
- Gestión de estados de juego. Ejemplos de estados serían «pantalla inicial», «jugando», «pausado», «game over» o «pantalla final».
- Gestión de colisiones.
- Gestión del *game loop* con sus tres funciones principales: manejo de eventos, actualización y dibujado en pantalla.
- Soporte para escenarios basados en tiles y con posibilidad de añadir backgrounds.

Son unos requisitos bastante generales debido al desconocimiento inicial. A medida que se vaya estudiando la bibliografía, realizando los primeros prototipos y viendo qué complejidad aproximada tiene la implementación de cada funcionalidad, se podrán determinar mejor requisitos más concretos.

6.2. Herramientas de desarrollo

Una vez establecidos los requisitos del framework, la primera decisión de importancia que hay que tomar es qué tecnología se va a utilizar para su desarrollo. De las herramientas para el desarrollo de videojuegos mencionadas en el estudio del estado del arte, se han descartado directamente los motores de videojuegos especializados, pues uno de los objetivos de este proyecto es tener una noción de las fases de desarrollo de un juego a un nivel relativamente bajo. Por ello en ningún momento se consideró utilizar Unity o Unreal Engine, pues manejarían por nosotros detalles como el renderizado de sprites, el sonido, la física o las colisiones, por no hablar de que la segunda parte del proyecto, la creación de un editor con su GUI, perdería su sentido debido a que estas herramientas de alto nivel ya lo incorporan.

Por tanto nuestra preferencia era utilizar una biblioteca sencilla que nos proporcionara acceso de bajo nivel a los elementos característicos que toman parte en la creación de un juego (gestión de gráficos, sonido, entrada y salida, etc.), dejando en nuestras manos el resto. Para tomar la decisión final se eligieron tres herramientas que, como adelantábamos en el estado del arte, serán compatibles con los lenguajes C++ o C#.

Para C++ hemos elegido las bibliotecas SDL y SFML, pues se adaptan completamente a nuestras necesidades. La tercera utilidad elegida fue Monogame, para además de tener un representante del lenguaje C#, poder ver la diferencia entre bibliotecas multipropósito y una herramienta más avanzada que, aunque lejos del nivel de un motor, sí que está orientada a videojuegos, como no podía ser de otra forma siendo una implementación libre de XNA, la API a la que Microsoft retiró el soporte en 2013. En el siguiente capítulo se realizará una comparativa entre estas tres herramientas, con el fin de elegir una de ellas.

6.3. Determinación del alcance de los juegos

Para que el framework sea útil, el código debe ser considerablemente genérico; no obstante, se incluirán una serie de características específicas que definirán el comportamiento básico del tipo de juegos que podemos crear de inicio con el framework, sin añadir o modificar código.

Es importante marcar unos límites para estas características disponibles de inicio ya que, por la naturaleza del proyecto, es muy sencillo verse tentado a añadir funcionalidad y exceder el alcance que debería tener, así como el tiempo que se le va a dedicar. Estos límites además redundarán directamente en las posibilidades del editor que se construirá en la segunda parte del proyecto, que básicamente será una GUI para el framework. A continuación se describe la estructura de los juegos que se podrán crear con el conjunto de clases inicial.

6.3.1. Avance

El juego comenzará con una pantalla inicial, que dará paso al primer nivel. La estructura de avance es secuencial: al concluir un nivel pasamos al siguiente y así hasta el último nivel, que al ser finalizado dará paso a la pantalla final.

Cada nivel tiene un punto de partida y puede tener una o varias metas. Puede crearse un nivel sin ninguna meta, pero esto redundará en que sea imposible finalizar el juego, por lo que no se recomienda a menos que quiera lograrse ese efecto por alguna razón.

Para finalizar el juego con éxito el jugador deberá desplazarse por los escenarios y sus plataformas teniendo como principales armas su movimiento y su salto (con el que además de moverse entre las plataformas podrá deshacerse de los enemigos), además de algunos ítems que harán más sencilla su tarea.

6.3.2. Estado del jugador

El estado del jugador vendrá determinado en cada momento por su salud, su número de vidas, su puntuación y el número de llaves en su haber.

Salud Indica lo cerca que está el jugador de perder una vida, lo que ocurrirá cuando la salud llegue a cero.

Vidas Las vidas son las oportunidades que tiene el jugador para superar todos los niveles. Al llegar a cero terminará el juego y veremos la pantalla de Game Over.

Puntuación La puntuación sirve para aumentar el contador de vidas, consiguiendo una adicional al llegar a la puntuación máxima. Si obtenemos exactamente la puntuación máxima, se reiniciará a cero, y si la superamos, la diferencia será la nueva puntuación.

Llaves Las llaves nos permiten eliminar obstáculos que nos cierren el paso. Cada obstáculo puede requerir de un número distinto de llaves.

Estos atributos tienen unos valores mínimos y máximos configurables.

La información sobre el estado del jugador se mostrará en el *Head-up display (HUD)* en la parte superior de la pantalla; el usuario elegirá los iconos que representarán la salud, las vidas, la puntuación y las llaves, pudiendo así adaptarlo a la estética del juego que se esté creando.

6.3.3. Pantallas

Podemos decir que el juego tiene cinco estados: inicio, jugando, pausado, *game over* y finalizado. Cada uno de estos estados, a excepción de «jugando», tienen la forma de una pantalla estática esperando a que pulsemos una tecla para cambiar de estado. Las pantallas son las siguientes:

Start screen Pantalla que se muestra nada más iniciarse el juego. Al pulsar la tecla Space pasaremos al comienzo del primer nivel.

Pause screen Pantalla que se nos presentará cuando el juego esté pausado. Para pausar y reanudar el juego pulsamos la tecla Enter.

Game over screen Al perder todas las vidas veremos esta pantalla. Pulsando Space nos encontraremos nuevamente en la pantalla de inicio.

End screen Pantalla que se mostrará al alcanzar el final del último nivel. Pulsando Space volvemos a la pantalla de inicio.

Además de los métodos que aquí se mencionan para pasar de un estado a otro, en cualquier momento puede pulsarse la tecla Escape para salir del juego.

6.3.4. Objetos del juego

Por «objetos del juego» nos referimos a los elementos dinámicos que nos encontraremos en los niveles mientras estemos jugando. Aunque todos los objetos de un mismo tipo compartirán un comportamiento base, tendrán una serie de atributos modificables.

Los dividiremos en tres grupos:

Enemigos

Los enemigos representan el peligro para el jugador, pudiendo restarle salud y hacerle perder vidas. Podemos configurar su apariencia, vida, velocidad, dirección, daño causado y superficie de colisión.

Left-Right Se mueve horizontalmente, cambiando de sentido al encontrarse con una tile colisionable o el borde de un precipicio.

Flying Ignora la gravedad y se mueve en cualquier dirección, modificándola al encontrar algún obstáculo según el ángulo con el que colisione.

Chaser Espera pacientemente a que el jugador entre en su campo de visión (configurable), momento en que empieza a perseguirlo en el eje horizontal hasta que vuelve a salirse de rango. No se detiene en los precipicios.

Flying chaser Igual que el anterior pero «vuela», es decir, persigue al jugador también en el eje vertical.

Para deshacerse de un enemigo, el jugador tendrá que saltar encima de él tantas veces como puntos de vida tenga.

Ítems

Los ítems son objetos que al ser recogidos aportan algún beneficio al jugador. Podemos configurar su apariencia, la cantidad que suman a su categoría al recogerlo y su superficie de colisión.

Health Aumenta la salud actual.

 **Lives** Aumenta el número de vidas restantes.

 **Score** Aumenta la puntuación.

 **Key** Aumenta la cantidad de llaves.

Otros

Aquí se incluyen los objetos que no encajaban del todo en las otras categorías.

 **Player** Es el avatar del jugador y, por tanto, el objeto principal del juego. Podemos configurar su apariencia, superficie de colisión, velocidad de movimiento, velocidad de salto, altura de salto y tiempo que dura la invulnerabilidad tras recibir daño.

 **Door** Obstáculo que necesita de un número determinado de llaves para ser destruido. Podemos configurar su apariencia, superficie de colisión y cantidad de llaves requeridas para que desaparezca.

 **Check point** Punto de control de aspecto configurable en el que reapareceremos tras perder una vida, siempre que esa vida no sea la última. Podemos configurar su apariencia y superficie de colisión.

 **End of level** Al tocarlo pasamos al siguiente nivel o, si el nivel actual es el último, el juego concluye y vemos la pantalla final. Podemos configurar su apariencia y superficie de colisión.

6.3.5. Sonido

Habrá dos tipos de sonidos, las pistas de música y los efectos de sonido. Por defecto cada nivel podrá tener una pista de música asociada que sonará en bucle. Por su parte, habrá una serie de efectos de sonido que los objetos del juego emitirán bajo ciertas circunstancias:

- **Player**: Al saltar y al morir.
- **Enemigos**: Al morir y al herir al jugador.
- **Ítems**: Al ser recogidos.
- **Check point** y **End of Level**: al ser alcanzados.
- **Door**: Al ser abiertas.

6.3.6. Controles

Los juegos creados tendrán un control por defecto sencillo basado en no más de cuatro teclas.

Acción	*Comando*
Mover hacia la derecha	Flecha der.
Mover hacia la izquierda	Flecha izq.
Saltar	Space
Pausar juego	Enter
Reanudar juego	Enter
Comenzar juego	Space
Reiniciar tras *game over*	Space
Reiniciar tras finalizar	Space
Salir del juego	Escape

Capítulo 7

Comparativa de herramientas

Para realizar la elección definitiva se valorarán los siguientes aspectos: plataformas soportadas, facilidad de uso, presencia en la industria y actividad de la comunidad.

7.1. Instalación y facilidad de uso

Para tener una ligera impresión sobre la facilidad de uso de cada conjunto de herramientas seleccionado, se instalarán las tres y se realizará un programa de prueba. El programa de prueba mostrará una ventana con fondo negro y un círculo rojo a modo de botón en el centro (Figura 7.2); al pulsar el botón izquierdo del ratón sobre el círculo se cargará una imagen desde disco (Figura 7.1) y ocupará toda la superficie de la ventana.

Estas pruebas se han realizado en el sistema operativo Windows y el entorno de desarrollo (IDE) elegido ha sido Microsoft Visual Studio Ultimate 2013 (en adelante MVS) en los tres casos.

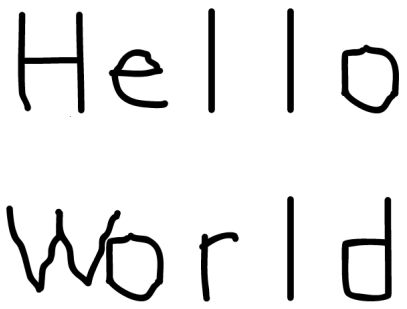


Figura 7.1 – Imagen *Hello World.*

Figura 7.2 – Imagen *Red Circle.*

7.1.1. C++/ SDL 2.0

Para instalar SDL 2.0 con MVS se ha seguido el tutorial de Lazy Foo[12]. Él utiliza la versión 2010 Ultimate de MVS, pero el proceso a seguir es el mismo en el caso de SDL (como se verá, no sucede lo mismo con Monogame).

Una vez iniciado un nuevo proyecto, es necesario configurar sus propiedades para que MVS sepa dónde encontrar los archivos de cabecera y binarios (disponibles en la página oficial de SDL) y enlazar con la biblioteca. Asimismo, para poder compilar código SDL hay que mover la biblioteca de enlace dinámico (archivos .dll) al directorio de trabajo del proyecto o al directorio del sistema.

Una vez hecho esto ya está todo listo para programar una aplicación usando la API de SDL. A continuación se divide el código del programa de prueba en bloques, cada uno seguido de una explicación:

```
#include <SDL.h>
#include <SDL_image.h>
#include <stdio.h>
#include <math.h>

//Screen dimension constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
//Button radius constant
const int BUTTON_RADIUS = 50;
```

Los ficheros que se incluyen son la propia SDL, SDL_image[13], el sistema de entrada salida estándar de C (stdio) y por último math, la librería numérica de C para operaciones y transformaciones matemáticas comunes.

SDL_image es una biblioteca adicional de SDL (el IDE se configura de igual forma que para la biblioteca SDL estándar) que permite cargar archivos de imagen distintos a BMP, que son los únicos soportados de forma nativa por SDL.

Después de la inclusión de archivos, se declaran tres constantes numéricas: dos para el ancho y alto de la ventana y una que nos indica el radio del círculo que hará de botón. A continuación se comienza con la función principal:

```
int main( int argc, char* args[] )
{
  //The window we'll be rendering to
  SDL_Window* window = NULL;

  //The surface contained by the window
  SDL_Surface* screenSurface = NULL;

  //The images we will load and show on the screen
  SDL_Surface* buttonImage = NULL;
  SDL_Surface* helloWorld = NULL;
```

Al principio se declaran una ventana SDL y unas superficies de pantalla, que no son más que imágenes 2D, pudiendo ser la imagen contenida dentro de una ventana (para lo que se usará screenSurface) o una imagen cargada desde un archivo (buttonImage y helloWorld).

```
SDL_Rect destRect;
destRect.x = SCREEN_WIDTH / 2 - BUTTON_RADIUS;
destRect.y = SCREEN_HEIGHT / 2 - BUTTON_RADIUS;
destRect.w = BUTTON_RADIUS * 2;
destRect.h = BUTTON_RADIUS * 2;
```

destRect es la estructura de tipo SDL_Rect encargada de almacenar el área rectangular donde se cargará la imagen del círculo que actuará como botón. Sus medidas y posición se han definido de manera relativa a las dimensiones de la pantalla y el radio del círculo.

La API de renderizado de SDL 2.0 permite dibujar puntos, líneas y rectángulos de forma directa[14]; sin embargo, para dibujar líneas curvas o elipses existen dos opciones: dibujar punto a punto utilizando la función matemática de la forma geométrica que se desea dibujar, o cargar una imagen que la contenga. Se ha optado por la segunda opción, utilizando la imagen en formato PNG (para que pueda tener fondo transparente) de un círculo rojo de 50 píxeles de radio, como el de la Figura 7.2.

A continuación se inicializa SDL:

```
if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
{
  printf( "SDL could not initialize! SDL_Error: %s\n", SDL_GetError() );
}
```

Inicializar la biblioteca es un paso necesario para poder llamar a cualquiera de sus funciones. A la función SDL_Init se le pasa como argumento sólo la *flag* SDL_INIT_VIDEO puesto que sólo se va a utilizar el subsistema de vídeo (SDL se divide en varios subsistemas). En caso de que haya algún error al inicializar, se imprime por pantalla haciendo uso de la función SDL_GetError.

```
else
{
  //Create window
  window = SDL_CreateWindow( "Hello World", SDL_WINDOWPOS_UNDEFINED,
     SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN )
     ;
  if( window == NULL )
  {
    printf( "Window could not be created! SDL_Error: %s\n", SDL_GetError() );
  }
```

Si SDL se ha inicializado correctamente, se crea la ventana. El primer argumento establece el título de la ventana, es decir, la parte rodeada de rojo en la Figura 7.3. El segundo y tercer argumento definen la posición (x, y) de la pantalla en la que la ventana se creará, mientras que el cuarto y el quinto indican el ancho y el alto de la ventana, definidos ya al inicio del archivo. Por último se le pasan las flags que indican acciones especiales asociadas a la creación de la ventana; en este caso se especifica mediante SDL_WINDOW_SHOWN que la ventana se muestre al crearse. Al igual que al inicializar SDL, se comprueba si ha habido algún error y en tal caso se imprime por pantalla.

Figura 7.3 – Título de la ventana

Si la ventana se ha creado correctamente, se intenta inicializar SDL_image:

```
else
{
  //Initialize PNG loading
  int imgFlags = IMG_INIT_PNG;
  if (!(IMG_Init(imgFlags) & imgFlags))
  {
    printf("SDL_image could not initialize! SDL_image Error: %s\n",
       IMG_GetError());
  }
```

Queremos inicializarla para que cargue PNG, así que le pasamos a IMG_Init la flag correspondiente. IMG_Init devuelve las flags que se han cargado correctamente, por tanto si el valor devuelto no contiene las flags (en este caso sólo una) que se le han pasado, significa que hay un error. De no haberlo, se pasa al siguiente bloque:

```
else
{
```

```
  //Get window surface
  screenSurface = SDL_GetWindowSurface(window);

  //Load images
  helloWorld = SDL_LoadBMP("C:/Users/Aaron/Google Drive/PFC/imagenes/
     hello_world.bmp");
  buttonImage = IMG_Load("C:/Users/Aaron/Google Drive/PFC/imagenes/redCircle.
     png");

  if (helloWorld == NULL ||buttonImage == NULL)
  {
    printf("Unable to load images!");
  }
```

Se accede a la superficie de la ventana mediante la función SDL_GetWindowSurface y se asigna a la variable que la contendrá: screenSurface. A continuación, si alguna de las dos imágenes que se necesitan no se carga correctamente, se informa del error. Nótese la diferencia entre la imagen BMP, que al ser soportada de forma nativa se carga con la función SDL_LoadBMP, y la imagen PNG, para la que hay que utilizar la función IMG_Load de la biblioteca SDL_image.

Si el subsistema de vídeo, la biblioteca SDL_Image, la ventana y las imágenes se han cargado correctamente, ya está todo listo para entrar en el bucle principal:

```
else
{
  //Main loop flag
  bool quit = false;

  //Event handler
  SDL_Event e;

  //To check if the user has clicked the left mouse button over the red
     circle
  bool redButtonPressed = false;
  int xDif = 0;
  int yDif = 0;
```

El bucle principal, también llamado a veces *game loop*, estará activo mientras la aplicación siga en funcionamiento, esperando por eventos (pulsaciones de teclado o ratón, cierre de la ventana, etc.), frente a los que el programa responderá de una forma u otra. Antes de entrar en el bucle se declaran una serie de variables: quit y redButtonPressed de tipo booleano para comprobar si el usuario ha cerrado la ventana o ha pulsado sobre el botón rojo, e de tipo SDL_Event para comprobar los eventos que se producen, y xDif e yDif, dos variables auxiliares de tipo entero para comprobar si cuando el usuario pulse el botón izquierdo del ratón, el cursor está encima del botón rojo.

Se comprobarán pues dos eventos. El primero es si el usuario ha cerrado la ventana (haciendo click en la X de la esquina superior derecha).

```
//While application is running
while (!quit)
{
  //Handle events on queue
  while (SDL_PollEvent(&e) != 0)
  {
```

```
    //User requests quit
    if (e.type == SDL_QUIT)
    {
      quit = true;
    }
```

El bucle principal o *game loop* se ejecutará mientras quit sea falso. El bucle de eventos (del inglés *event loop*) se controla mediante la función SDL_PollEvent(&e). Lo que hace esta función es tomar el evento más reciente de la cola de eventos y asignarlo a la variable de tipo SDL_Event que le hemos pasado.

Lo primero que se comprueba es si se ha producido un evento del tipo SDL_QUIT; esto es, si el usuario ha cerrado la ventana. En ese caso quit se hace verdadero para que no se ejecute la siguiente iteración del bucle principal. Seguidamente se comprueba si se ha hecho click izquierdo del ratón sobre el círculo rojo.

```
//If the left button is pressed
if (!redButtonPressed)
{
  if (e.type == SDL_MOUSEBUTTONDOWN)
  {
    if (e.button.button == SDL_BUTTON_LEFT)
```

Si el botón rojo ya ha sido pulsado no es necesario volver a comprobarlo, ya que se habrá cargado la imagen de la Figura 7.1 y el programa no hará nada más, excepto terminar cuando se cierre la ventana. Si no se ha pulsado, se comprueba si se produce un evento de tipo SDL_MOUSEBUTTONDOWN (es decir, si se produce una pulsación de algún botón del ratón) y si ha sido el izquierdo, en cuyo caso se realiza la siguiente comprobación:

```
{
  //Check if the mouse is over the button
  xDif = e.button.x - SCREEN_WIDTH / 2;
  yDif = e.button.y - SCREEN_HEIGHT /2;
  //Pythagorean theorem
  redButtonPressed = sqrt(pow(xDif, 2.) + pow(yDif, 2.)) < BUTTON_RADIUS;
}}}}
```

Este último bloque del bucle de eventos contiene las operaciones matemáticas para comprobar si el botón izquierdo del ratón ha sido pulsado dentro del área del círculo rojo. Para ello se halla la distancia euclidiana (deducida a partir del Teorema de Pitágoras) entre el cursor del ratón y el origen del círculo, y si es menor al radio de éste significa que se ha pulsado sobre él, por lo que cambiamos el estado de redButtonPressed al resultado de la comprobación.

Con esto finaliza el bucle de eventos, pero no el bucle principal, pues aún falta mostrar en la ventana lo que corresponda:

```
//Apply the image
if (!redButtonPressed)
{
  SDL_BlitSurface(buttonImage, NULL, screenSurface, &destRect);
}
else
{
  SDL_BlitSurface(helloWorld, NULL, screenSurface, NULL);
}
```

```
printf("%d\n", SDL_GetTicks());
//Update the surface
SDL_UpdateWindowSurface(window);
}}}}}
```

Si el botón rojo no se ha pulsado, se coloca la superficie buttonImage (que contiene el círculo de la Figura 7.2), en el centro de la ventana, como se ve en la Figura 7.4. Si por el contrario el usuario ya lo ha pulsado, la superficie helloWorld (que contiene la imagen de la Figura 7.1) ocupa toda la ventana, como se muestra en la Figura 7.5. Estas acciones se realizan por medio de la función SDL_BlitSurface.

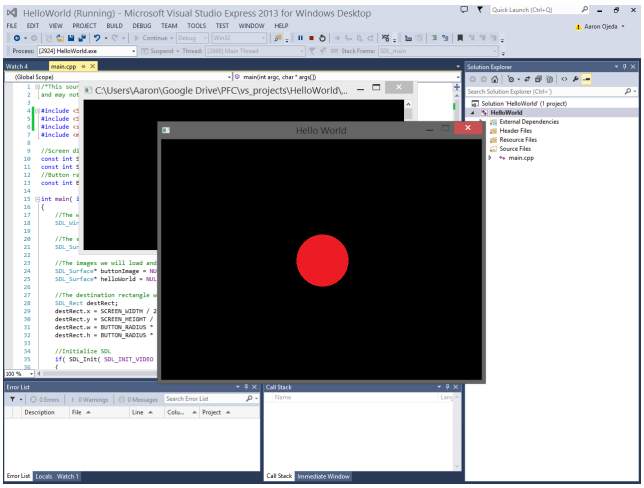


Figura 7.4 – SDL. Aspecto de la ventana antes de pulsar el botón rojo

SDL_BlitSurface toma una superficie origen y la copia en una superficie destino. El primer y el tercer parámetro del procedimiento son la superficie origen y la superficie destino respectivamente. El segundo parámetro es el área rectangular de la superficie origen que se va a copiar; como se van a copiar las imágenes enteras, se deja a NULL. El cuarto parámetro es el área rectangular de la superficie destino donde se va a copiar la imagen origen: en el caso de buttonImage se especifica que se copie en las coordenadas indicadas por destRect, ya que queremos que el círculo rojo esté en el centro de la ventana; con helloWorld no es necesario, ya que queremos que el origen de la imagen sea el de la ventana (la esquina superior izquierda), así que se deja a NULL.

Que se haya dibujado algo en la superficie de la ventana no implica que se vaya a ver. Para asegurarse de ello, una vez se haya dibujado todo lo que se quiera que se muestre en la ventana, es pertinente una llamada a la función SDL_UpdateWindowSurface pasándole como parámetro la ventana (window). Con esa instrucción finaliza el bucle principal, y sólo queda liberar los recursos que se hayan utilizado y finalizar el programa.

```
//Destroy window
SDL_DestroyWindow( window );

//Deallocate surface
SDL_FreeSurface( helloWorld );
helloWorld = NULL;
SDL_FreeSurface(buttonImage);
buttonImage = NULL;
```

```
//Quit SDL subsystems
IMG_Quit();
SDL_Quit();

return 0;
}
```

No es necesario liberar la superficie screenSurface, pues SDL_DestroyWindow se encarga de liberar los recursos asociados a la ventana.

Para construir este programa de prueba se ha utilizado código propio combinado con líneas de código extraídas de los tutoriales de Lazy Foo[15], en especial de las lecciones 1, 2, 3 y 6.

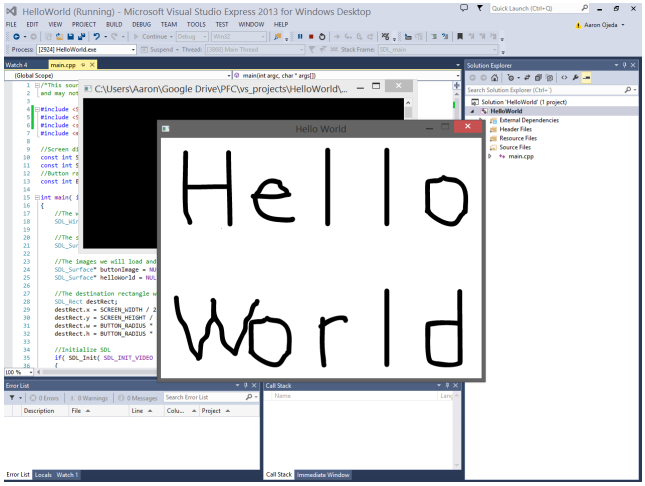


Figura 7.5 – SDL. Aspecto de la ventana tras pulsar el botón rojo

7.1.2. C++ / SFML 2.1

La manera de configurar el proyecto de MVS para instalar SFML es muy similar a la llevada a cabo con SDL, exceptuando algunas particularidades que se explican en el tutorial *SFML and Visual Studio* de la web oficial de la biblioteca[16]. La principal diferencia es que hay que enlazar la aplicación a la librería que coincide con la configuración que estamos modificando: Debug o Release. Así, siendo «xxx» el módulo de SFML que se va a utilizar, la librería se llamará sfml-xxx-d.lib para la versión Debug, y sfml-xxx.lib para la versión Release.

En este caso se van a utilizar los módulos *graphics*, *window* y *system* para Debug, por lo que se enlazará la aplicación con sfml-graphics-d.lib, sfml-window-d.lib y sfml-system-d.lib.

También se probó la opción de integrar directamente SFML en el ejecutable enlazando a la biblioteca estática; de esta forma ya no sería necesario lidiar con las DLL. Para ello se les añade el sufijo «s» a las bibliotecas: sfml-xxx-s-d.lib para Debug y sfml-xxx-s.lib para Release, además de añadir la macro SFML_STATIC en las opciones del preprocesador del proyecto.

En caso de que se elija la opción de enlazar a la biblioteca estática, es importante tener en cuenta que no funcionará con MVS 2013, ya que la última versión estable en este momento es para VC++ 11[17]. Se probó el entorno de desarrollo MVS 2010 con la versión de la biblioteca para VC++ 10 y funcionó correctamente. No obstante, el ejemplo a continuación se ejecutó en

MVS 2013 con la versión dinámica de la librería de VC++ 11, la cual no plantea problemas en la versión 2013 del IDE de Microsoft.

A continuación se comenta el código escrito para realizar el programa de prueba:

```
#include <SFML/Graphics.hpp>
#include <math.h>

// Screen dimension constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
// Button radius constant
const int BUTTON_RADIUS = 50;

int main()
{
  // Render window constructor
  sf::RenderWindow window(sf::VideoMode(SCREEN_WIDTH, SCREEN_HEIGHT), "Hello
     World");
```

Primero se incluyen los archivos necesarios: el módulo *graphics* y la biblioteca *math*. Después, como se hizo con SDL, se declaran las constantes que representan las dimensiones de la ventana y el radio del círculo que actuará como botón.

Dentro de la función principal, se declara una variable de la clase sf::RenderWindow, un tipo especial de ventana que permite dibujar las entidades proporcionadas por el módulo *graphics*. El primer argumento del constructor define el tamaño de la ventana (el tamaño interno, sin los bordes ni la barra con el título) y el segundo su título. La clase del primer argumento, sf::VideoMode, tiene algunas funciones estáticas interesantes para obtener la resolución del escritorio o la lista de modos de vídeo compatibles con pantalla completa[18].

```
// Red button in the middle of the screen
sf::CircleShape button(BUTTON_RADIUS);
button.setFillColor(sf::Color::Red);
button.setPosition(SCREEN_WIDTH / 2 - BUTTON_RADIUS, SCREEN_HEIGHT / 2 -
    BUTTON_RADIUS);

// Load a texture from an image file on disk
sf::Texture helloTexture;

if (helloTexture.loadFromFile("C:/Users/Aaron/Google Drive/PFC/imagenes/
    hello_world.bmp"))
{
```

En el código de arriba se declaran las dos variables que nos permitirán mostrar el círculo rojo (Figura 7.2) y la imagen que se mostrará al pulsar sobre él (Figura 7.1).

El módulo *graphics* de SFML proporciona la clase sf::CircleShape para dibujar formas circulares. Se le pasa al constructor el radio y se utilizan las funciones de dicha clase setFillColor y setPosition para definir el color y la posición del círculo en la ventana[19].

Para crear un sf::Sprite (una entidad rectangular con una imagen asociada[20]) se necesita una textura. Una textura es una imagen; se le denomina «textura» y no «imagen» porque tiene una función muy específica: ser mapeada a una entidad 2D. En SFML, la clase que encapsula las texturas es sf::Texture. Una vez declarada una instancia de esa clase, se carga la imagen de disco

(SFML soporta la mayoría de los formatos de imagen más comunes) y, si no se produce ningún error, se crea el sf::Sprite a partir de la textura:

```
// Create a sprite from the previous texture
sf::Sprite helloSprite;
helloSprite.setTexture(helloTexture);

// Variables to do the math for checking if the user has clicked the left
   mouse button over the red circle
sf::Vector2i mousePosition(0, 0);
sf::Vector2i difference(0, 0);

// Check if red button has been pressed
bool redButtonPressed = false;

// run the program as long as the window is open
while (window.isOpen())
{
```

Después de crear el sf::Sprite, se declaran tres variables: mousePosition y difference se utilizan para cuando el usuario pulse el botón izquierdo del ratón sobre un punto de la ventana, calcular la distancia euclidiana desde ese punto al centro del círculo rojo y, si es menor que el radio del círculo, significará que el cursor estaba dentro de su superficie. redButtonPressed, de tipo booleano, sirve para controlar si el usuario ya ha pulsado sobre el círculo rojo.

A continuación se entra en el bucle principal, que se ejecutará mientras la ventana window siga abierta. No es necesaria una explicación detallada ya que, como se verá, el bucle es casi idéntico al de SDL:

```
// check all the window's events that were triggered since the last iteration
    of the loop
sf::Event event;
while (window.pollEvent(event))
{
  // "close requested" event: we close the window
  if (event.type == sf::Event::Closed)
    window.close();
```

Se crea una instancia de la clase sf::Event para comprobar los eventos que ocurran en la ventana window. Si el usuario ha ejecutado la acción para cerrar la ventana, se llama a la función close para cerrarla.

```
if (event.type == sf::Event::MouseButtonPressed)
{
  if (event.mouseButton.button == sf::Mouse::Left)
  {
    // If the red button has not been pressed
    if (!redButtonPressed)
    {
      mousePosition = sf::Mouse::getPosition(window);
      difference = sf::Vector2i(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2) -
         mousePosition;
      // Pythagorean theorem
      redButtonPressed = sqrt(pow(difference.x, 2) + pow(difference.y, 2)) <
         BUTTON_RADIUS;
    }}}}
```

En el código de arriba se comprueba si se ha producido un click izquierdo del ratón sobre la ventana y si ha sido dentro de la superficie del círculo rojo, en cuyo caso redButtonPressed pasa a ser verdadero.

```
window.clear();
// We show the red button or - when pressed - the "hello_world" image
if (redButtonPressed)
{
  window.draw(helloSprite);
}
else
{
  window.draw(button);
}

window.display();
}}

return 0;
}
```

Por último se dibuja en la ventana la imagen que corresponda: el círculo rojo si aún no se ha pulsado sobre él o, cuando esto ocurra, la imagen con el texto «Hello World». La función clear limpia la ventana con el color que se le pase (negro por defecto), draw dibuja lo que le pasemos (siempre y cuando sea un objeto dibujable, de tipo sf::Drawable) y display lo muestra. Si no se llama a display no se mostrará lo que se haya dibujado en la ventana, como ocurría con SDL si no se llamaba a la función SDL_UpdateWindowSurface. Este ciclo clear/draw/display es, según la documentación oficial[18], la única forma correcta de dibujar elementos; probar otras estrategias redundará en la obtención de resultados extraños debido al *double-buffering*.

Los recursos en SFML son liberados por medio del destructor cuando salen del ámbito en que fueron definidos, por lo que no es necesario llamar a ninguna función que se encargue de eso.

7.1.3. C# / Monogame

Para instalar y probar MonoGame, se han seguido los tutoriales de Whitaker[21]. Al ser una implementación de la API XNA 4.0, la manera de dejar preparado MVS para trabajar con él difiere drásticamente de la seguida con SDL y SFML.

Desde la página web oficial del proyecto MonoGame[8] se pueden descargar los instaladores de las distintas versiones; en este caso se descargó la última versión disponible, la 3.2, y se ejecutó como se haría con cualquier ejecutable en Windows. Al terminar la instalación, si todo ha ido bien se verá cómo al iniciar MVS y crear un proyecto, existirá la opción de que sea un proyecto Monogame, como se ve en la Figura 7.6.

Una vez integrado MonoGame en MVS, surge una dificultad que tiene su origen en el hecho de que Monogame 3.2 hereda la gestión de contenido de XNA 4.0, la cual se realiza por medio de un conjunto de procesos denominado *content pipeline*.

Mientras se está creando un juego con XNA, se va volcando el contenido que éste requiera (imágenes, sonidos, música, modelados 3D, fuentes de texto, etc.) en el mentado *content pipeline*, de manera que tras pasar por varias fases transparentes al usuario, el contenido queda organizado

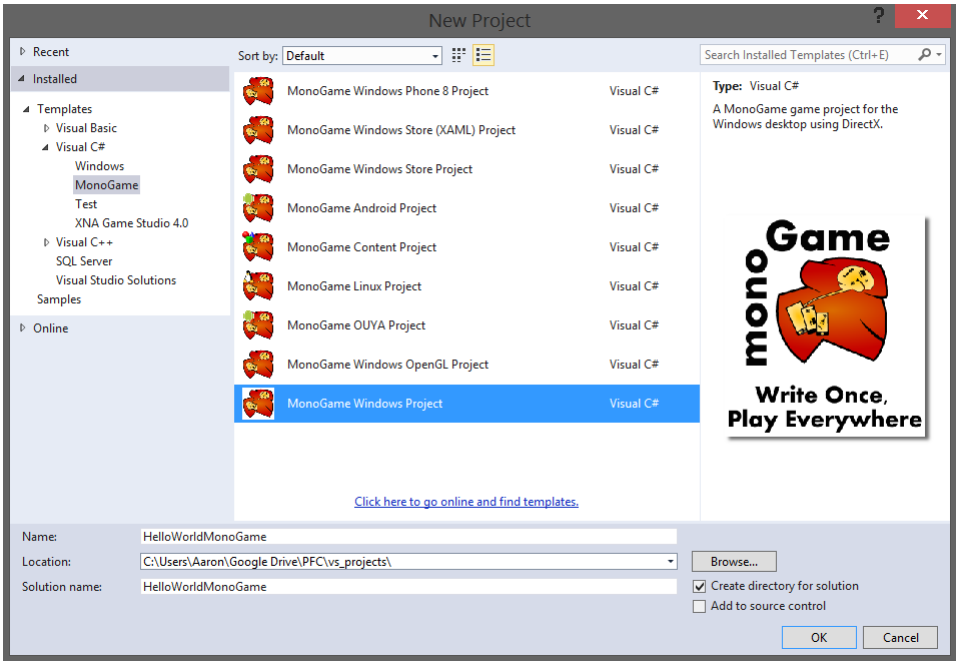


Figura 7.6 – Creación de un proyecto Monogame en MVS

y accesible de forma fácil y rápida desde el IDE. El equipo de desarrollo de MonoGame está trabajando en la creación de su propio *content pipeline*, y debería estar listo para la versión 3.3; no obstante, de momento no parece que el desarrollo esté cercano a su finalización.

En resumen: para utilizar MonoGame 3.2 es necesario tener instalado XNA 4.0, ya que el primero hace uso del *content pipeline* del segundo. El problema es que XNA 4.0 no funciona en versiones de MVS posteriores a la 2010, por lo que las alternativas más sencillas que se nos presentan son las siguientes:

- Trabajar con MVS 2010.
- Utilizar un *script* programado por Whitaker que permite la instalación de XNA en MVS 2013.

Se ha optado por la segunda opción, siguiendo las claras instrucciones proporcionadas por el creador del script[22]. Llegados a este punto ya es posible realizar en MVS 2013 un proyecto MonoGame para Windows.

Si creamos un nuevo proyecto eligiendo la plantilla de Windows tal y como se mostraba en la Figura 7.6, MonoGame generará automáticamente un código funcional con la estructura de la aplicación, que al ejecutarse generará la ventana de la Figura 7.7. Por tanto, sin necesidad de escribir una línea de código, ya se ha construido y mostrado una ventana. Antes de empezar a escribir el programa de prueba, se estudiará el código generado por el framework, el cual se divide en dos ficheros, Program.cs y Game1.cs. Es importante recordar que, al contrario que con SDL y SFML, con MonoGame se programa en el lenguaje C#.

Program.cs es el punto de entrada del juego: simplemente inicializa una instancia del juego y lo ejecuta. La mayoría de las veces no es necesario modificarlo.

El otro fichero generado, Game1.cs, es el realmente interesante. El primer bloque de código que se genera es el siguiente:

```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
using Microsoft.Xna.Framework.GamerServices;
using System.Diagnostics;
#endregion

namespace HelloWorldMonoGame
{
```

Se especifican los módulos que se van a utilizar y se crea el namespace con el nombre que le hayamos dado al proyecto. A continuación se crea la clase principal:

```
public class Game1 : Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
```

Como se puede ver, la clase Game1 hereda de la clase Game, la cual proporciona las funcionalidades básicas que todo juego debería tener. Lo siguiente que se verá es la declaración de dos variables: la primera es el enlace al dispositivo gráfico, y la segunda es la herramienta que se utilizará para dibujar *sprites* en pantalla. A partir de aquí se comienzan a definir los métodos de la clase, empezando por el **constructor** y siguiendo por métodos para **inicializar**, **cargar contenido**, **actualizar** y **dibujar**. Iremos comentándolos por orden.

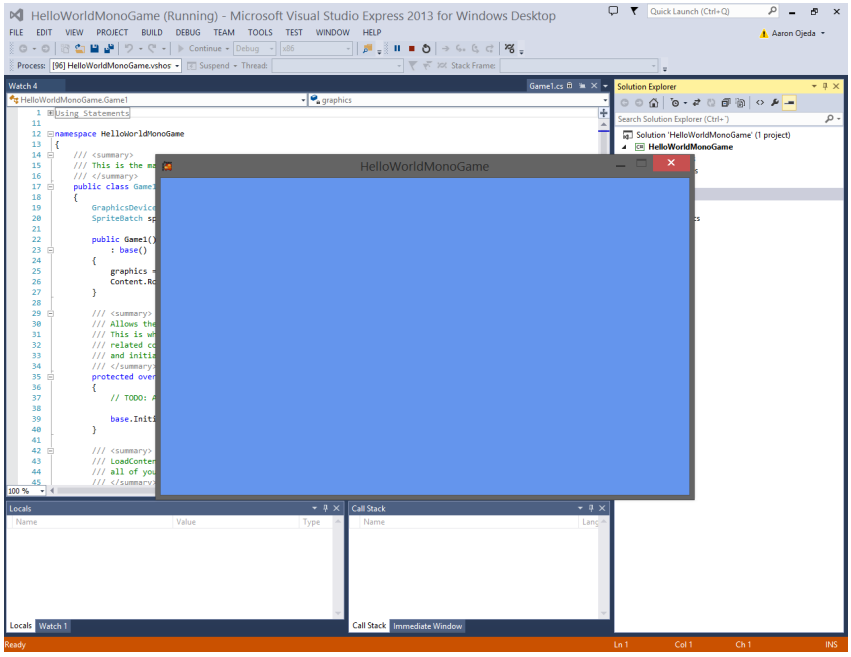


Figura 7.7 – Resultado al ejecutar el código generado por MonoGame para la plantilla de Windows

Constructor

Es un constructor típico en el que se prepara el administrador del dispositivo gráfico y el gestor de contenido.

```
public Game1()
    : base()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}
```

Initialize

Este método es llamado poco después del constructor, antes de que el bucle principal del juego (el comúnmente llamado *game loop*) comience.

```
protected override void Initialize()
{
    // TODO: Add your initialization logic here
    base.Initialize();
}
```

LoadContent

Aquí puede ser cargado el contenido del juego: música, modelados 3D, texturas, etc. No es obligatorio cargar el contenido dentro de este método, pero es el lugar ideal para juegos pequeños o para iniciarse en el framework. Al igual que Initialize, este método se llama antes del *game loop*.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);
}
```

Update

Este método ya se encuentra dentro del bucle principal, y es donde se define gran parte de lo que ocurre en el juego: actualización del mundo, control de colisiones, eventos de entrada recibidos, sonidos, etc. Aunque es configurable, por defecto se llamará a Update 30 veces por segundo. En este caso en concreto se está comprobando si el usuario presiona la tecla Escape (o si pulsa el botón Back en el caso de que hubiera un mando conectado) y, de ser así, se cierra la ventana.

```
protected override void Update(GameTime gameTime)
{
  if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed
      || Keyboard.GetState().IsKeyDown(Keys.Escape))
         Exit();

    base.Update(gameTime);
}
```

Draw

Al igual que Update, por defecto este método es llamado 30 veces por segundo. Una vez se haya actualizado lo que se quiere que se vea en pantalla, Draw lo dibujará. Es importante no realizar ninguna actualización en Draw, de la misma manera que no se debe dibujar en el método Update. En este caso se le está diciendo a GraphicsDevice que limpie la ventana con el color CornflowerBlue, dando como resultado lo que se veía en la Figura 7.7.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);
    base.Draw(gameTime);
}
```

Programa de prueba

A continuación se pasará a comentar el código añadido para el programa de prueba:

Se añaden varios miembros privados a la clase Game1:

```
private const int SCREEN_WIDTH = 600;
private const int SCREEN_HEIGHT = 480;
private const int buttonRadius = 50;

private Texture2D helloWorld;
private Texture2D redcircle;

private bool buttonPressed = false;
```

Los tres primeros miembros enteros indican las dimensiones de la ventana y el tamaño (radio) del círculo. Los dos siguientes, de tipo Texture2D, son necesarias para cargar las dos imágenes que se utilizarán (Figura 7.2 y Figura 7.1). buttonPressed es un miembro booleano que comienza siendo false y pasará a true desde que el usuario haga click con el botón izquierdo del ratón dentro del círculo rojo.

Se añadirán cuatro líneas al constructor, quedando como sigue:

```
public Game1()
    : base()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    this.IsMouseVisible = true;

    // Changing the window size
    graphics.PreferredBackBufferWidth = SCREEN_WIDTH;
    graphics.PreferredBackBufferHeight = SCREEN_HEIGHT;
    graphics.ApplyChanges();
}
```

Las dos primeras líneas dentro del cuerpo del constructor ya existían. A continuación se ha incluido una línea para que cuando el ratón se sitúe sobre la ventana el puntero siga siendo visible (es invisible por defecto) y tres líneas más para cambiar las dimensiones de la pantalla,

que por defecto son de 800 píxeles de ancho por 480 píxeles de alto; se ha cambiado a 640x480 para que sea del mismo tamaño que las que se han programado con SDL y SFML, ya que además son las dimensiones de la imagen de la Figura 7.1.

El método Initialize se queda exactamente igual a como estaba. No ocurre lo mismo con LoadContent:

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    redcircle = Content.Load<Texture2D>("circle");
    helloWorld = Content.Load<Texture2D>("hello_world");
}
```

Se han añadido dos líneas de código para cargar el contenido del programa de prueba, es decir, las dos imágenes. El contenido se gestionará a través del mentado *content pipeline* de XNA, para lo cual es necesario llevar a cabo una serie de acciones previas.

Una vez se haya instalado XNA como se explicó previamente, añadimos un nuevo proyecto de tipo MonoGame Content a nuestra solución de MVS (en Visual Studio, una solución es un conjunto de uno o más proyectos complementarios). En lugar de crearse únicamente un nuevo proyecto, MVS creará dos, uno con el nombre que se le haya dado (en este caso HelloWorldMonoGame_) y otro con ese nombre seguido de «Content» (Figura 7.8). Este último, el llamado HelloWorldMonoGame_Content es el verdadero proyecto para gestionar el contenido; es un poco desconcertante y sería mejor que fuera el único existente, pero es el pequeño precio a pagar por tomar prestado el *content pipeline* de XNA. El otro proyecto, HelloWorldMonoGame_, es un proyecto intermedio que se encarga de realizar los enlaces oportunos a través del nodo Content References.

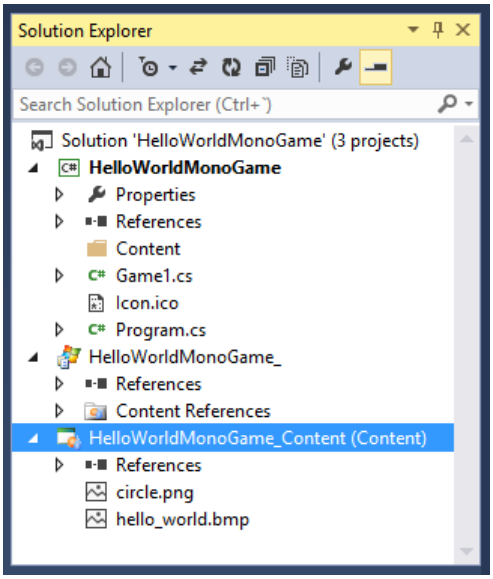


Figura 7.8 – Proyectos de la solución HelloWorldMonoGame

El último paso es añadir a las referencias del proyecto principal el que se acaba de crear (el que tiene el nombre acabado en «Content»). Hecho esto, ya es posible añadir recursos al proyecto;

como se ve en la Figura 7.8, se han añadido las dos imágenes necesarias para completar el programa de prueba: circle.png y hello_world.bmp. Nuevamente, para entender y dejar preparada la gestión de contenido se ha seguido el tutorial de Whitaker dedicado a ello[23].

Éstas eran las líneas de código introducidas en el método LoadContent:

```
redcircle = Content.Load<Texture2D>("circle");
helloWorld = Content.Load<Texture2D>("hello_world");
```

Nótese que, una vez añadidos recursos al proyecto, para cargarlos puede escribirse su nombre sin la extensión (.png y .bmp en este caso). Al contrario que XNA, MonoGame no obliga a hacerlo, por lo que se puede elegir si incluir la extensión en el nombre o no.

El método Update se ha modificado para comprobar si el usuario ha pulsado dentro del círculo:

```
protected override void Update(GameTime gameTime)
{
  if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed
     || Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    // TODO: Add your update logic here
    MouseState mouseState = Mouse.GetState();
    if (mouseState.LeftButton == ButtonState.Pressed)
    {
        // Do whatever you want here
        //We check if the left button has been clicked inside the circle (
           button)
        if (!buttonPressed)
        {
            if (Math.Sqrt(Math.Pow(mouseState.X - GraphicsDevice.Viewport.
               Width / 2, 2) + Math.Pow(mouseState.Y - GraphicsDevice.
               Viewport.Height / 2, 2)) < buttonRadius)
            {
                buttonPressed = true;
            }
        }
    }

    base.Update(gameTime);
}
```

El primer if se ha dejado intacto. Posteriormente, de igual forma a como se hacía en SDL y SFML, se comprueba si se ha pulsado el botón izquierdo del ratón en el interior del círculo rojo, a través del cálculo de la distancia euclidiana entre el lugar donde se ha hecho click y el centro del círculo. En caso de que así sea, el miembro booleano buttonPressed pasa a ser verdadero.

Por último, se ha incluido en el método Draw la lógica necesaria para comprobar si el botón se ha pulsado o no y mostrar en la ventana la imagen que corresponda:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    // TODO: Add your drawing code here
    spriteBatch.Begin();
    if (buttonPressed)
```

```
    {
        spriteBatch.Draw(helloWorld, new Rectangle(0, 0, GraphicsDevice.
            Viewport.Width, GraphicsDevice.Viewport.Height), Color.White);
    }
    else
    {
        spriteBatch.Draw(redcircle, new Vector2(GraphicsDevice.Viewport.Width
            / 2 - redcircle.Width / 2, GraphicsDevice.Viewport.Height / 2 -
            redcircle.Height / 2), Color.White);
    }
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Lo primero que se ha hecho ha sido sustituir el color con el que se limpia la pantalla, cambiando el azul por defecto a negro, para que el aspecto de la ventana sea exactamente igual al que hemos visto con SDL y SFML. A continuación, si el usuario ha pulsado el botón, se muestra la imagen con el texto «Hello World» ocupando toda la ventana; si no lo ha pulsado, se muestra el círculo rojo en el centro.

El dibujado se realiza por medio del método Draw del objeto spriteBatch. El primer argumento indica qué textura se quiere utilizar, el segundo es el rectángulo (o vector) que indica la posición en la que el sprite será dibujado y el tercero es el color con el que se teñirá el sprite (si no se quiere teñir, se usa el blanco).

7.1.4. Resumen

Instalación

Las bibliotecas SDL 2.0 y SFML 2.1 han sido instaladas sin ningún problema. MonoGame 3.2 ha sido la tecnología cuya instalación más trabajo ha generado, por la peculiaridad comentada previamente según la cual MonoGame hace uso del *content pipeline* de XNA, el cual no funciona para versiones de MVS superiores a la 2010. Por tanto, si se quiere usar MVS 2013, o cualquiera posterior a MVS 2010, es necesario ejecutar un script que permita instalar XNA en esa versión. Además, una vez creada la solución en MVS, es necesario añadir un proyecto de contenido y establecerlo como dependencia del proyecto principal.

Una vez se tienen claro los pasos, ninguna de las tecnologías debería ocasionar problemas en su instalación. No obstante, si hubiera que establecer un orden, y siempre teniendo en cuenta que se está utilizando MVS, se podría decir que SDL 2.0 y SFML 2.1 han sido las más sencillas de instalar, quedando MonoGame 3.2 en último puesto.

Facilidad de uso

Aunque con un programa de prueba tan sencillo como el que se ha realizado no es suficiente para comprender la facilidad de uso de un lenguaje, biblioteca o framework, sí que puede servir para hacerse una idea muy básica de cómo será trabajar con esa tecnología.

SDL es muy potente y proporciona todo lo necesario para construir un juego. Esto no quita que sea la que menos facilidades le proporciona al programador inicialmente, pues al estar programada en C no sigue el modelo de programación orientada a objetos (OOP) y, en general, no hace uso de las características de lenguajes de más alto nivel.

Laurent Gomila, creador de SFML (programada en C++), explica que podemos pensar en su biblioteca como «una SDL orientada a objetos». En el programa de prueba ya se pueden observar características propias de ese paradigma de programación, principalmente el uso de clases, lo cual proporciona ventajas como el uso de constructores y la comodidad que supone liberar recursos por medio de los destructores.

Por otra parte, MonoGame es un framework, y como tal genera código automáticamente, proporcionando la estructura o esqueleto de un juego mediante la creación de métodos para inicializar, cargar contenido, actualizar o dibujar, que se llamarán con un orden y frecuencia ya establecidos (aunque es configurable). De esta forma, escribiendo el código pertinente en esos métodos, ya el juego estará listo para su ejecución con menos esfuerzo que el requerido por SFML y SDL. Además, al utilizar el *content pipeline* de XNA, el programador dispondrá de una forma muy cómoda de gestionar el contenido de su aplicación. Todo esto, unido al hecho de que el código de se escribe en C#, un lenguaje interpretado de más alto nivel que C++, hace que MonoGame parezca a priori la opción más fácil de usar, cómoda y rápida a la hora de producir resultados de las tres que se han utilizado.

A pesar de lo comentado, la facilidad de uso depende de otros factores aparte del nivel de abstracción de la tecnología que se utilice, sobre todo de la experiencia y las preferencias del programador. Dicho esto, sí que es cierto que analizando las tecnologías de manera aislada (y recalcando que el programa de prueba realizado es muy simple), el orden descendente de facilidad de uso más probable es: MonoGame 3.2, SFML 2.1 y SDL 2.0.

7.2. Presencia en la industria

En este apartado se analizará si las tecnologías elegidas son utilizadas en el desarrollo profesional e independiente de videojuegos.

7.2.1. SDL

El conjunto de bibliotecas SDL ha sido utilizado en un gran número motores, emuladores y juegos*[24]*: desde grandes clásicos de la escena independiente más pura como Dwarf Fortress o todas las versiones de DROD hasta superproducciones del calibre de Half-Life 2 y Neverwinter Nights, pasando por proyectos independientes más conocidos, actuales y exitosos comercialmente como World of Goo, VVVVVV o FTL.

Viendo la cantidad y la calidad de los juegos que hacen uso de ella, SDL no deja dudas respecto a su aplicación a nivel profesional.

Figura 7.9 – DROD, Half-Life 2 y FTL: Faster than Light. Juegos que hacen uso de SDL

7.2.2. SFML

Es difícil encontrar proyectos de envergadura que hagan uso de esta API, haciéndola palidecer ante el catálogo que hace uso de SDL. Probablemente, el juego más importante desarrollado hasta la fecha haciendo uso de SFML es Atom Zombie Smasher, disponible en Steam y con una puntuación media de 75 sobre 100 en Metacritic[25].

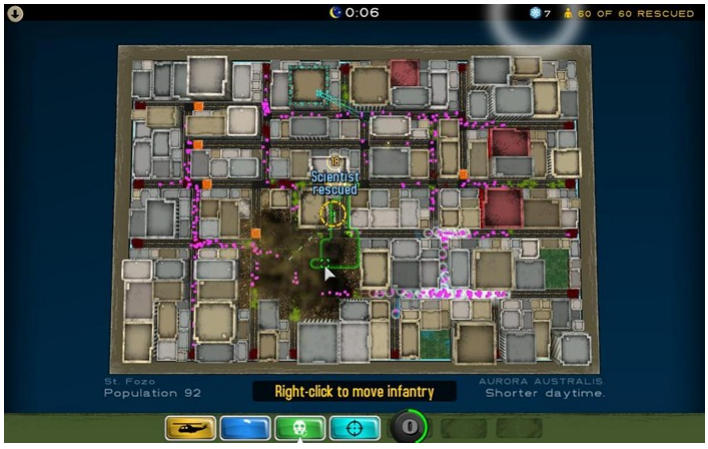


Figura 7.10 – Atom Zombie Smasher, creado con SFML

No hay que olvidar que SFML es casi 10 años más joven que SDL, por lo que si siguen saliendo juegos como el mencionado es muy posible que su popularidad y aplicación en la industria se vean incrementadas. Sin embargo, si a día de hoy hay una razón para recomendar esta biblioteca, ésa no es su presencia en la industria del videojuego.

7.2.3. MonoGame

Los juegos de mayor renombre desarrollados con Monogame han sido FEZ, una de las obras maestras de los últimos años, y los excelentes productos lanzados por el estudio Supergiant Games: Bastion y Transistor. De menor relevancia pero también muy conocidos son Skulls of the Shogun y los dos juegos que Tribute Games ha desarrollado con este framework: Wizorb y Mercenary Kings.

Aunque no sea un catálogo de juegos que sobresalga por su cantidad, hay que tener en cuenta que XNA, la API que implementa Monogame, no dejó de tener el soporte de Microsoft hasta 2013. Antes de esa fecha, los únicos motivos para utilizar Monogame eran portar los trabajos realizados con XNA a otras plataformas y razones ideológicas que justificaran la elección de una tecnología de código abierto.

Si incluyéramos los juegos creados con XNA, la lista se ampliaría drásticamente, pues además de muchos juegos de escritorio para Windows habría que incluir la totalidad del catálogo de Xbox Live Arcade, la plataforma de distribución digital de Microsoft para Xbox 360, centrada en pequeños y medianos juegos descargables tanto de compañías de peso en el sector como de desarrolladores independientes.

Figura 7.11 – Bastion y FEZ, dos juegos de éxito creados con MonoGame.

7.3. Actividad de la comunidad

Para medir de forma aproximada la actividad de la comunidad de cada tecnología, se ha optado por contar el número de preguntas etiquetadas con el nombre de cada una de ellas en Stack Overflow[26] y Game Development Stack Exchange[27].

Stack Overflow es probablemente la web más importante de preguntas y respuestas (Q&A) sobre programación. A raíz de ella surgió Stack Exchange, una red de sitios web Q&A sobre diversos temas en múltiples campos, entre los que se encuentra Game Development Stack Exchange, centrada en los desarrolladores independientes y profesionales de videojuegos.

Se han tenido en cuenta sólo las preguntas realizadas desde enero de el año 2013 hasta la fecha actual (octubre de 2014), para no penalizar a las tecnologías más modernas. El término introducido para la búsqueda es

```
[tag] created: 2013..2014
```

, donde «tag» se ha sustituido por los siguientes valores: sdl, sfml, monogame y xna. XNA no entra estrictamente entre las tecnologías que se han valorado, pero se ha considerado interesante incluirla en la búsqueda debido a que gran parte de las preguntas realizadas sobre ella son aplicables a MonoGame. Los resultados se muestran en el Cuadro 7.1.

	Stack Overflow	GameDev Stack Exchange
SDL	2888	304
SFML	1377	158
MonoGame	1101	451
XNA	6020	2328

Cuadro 7.1 – Número de preguntas etiquetadas con el nombre de cada tecnología.

Si bien es fácil deducir que la comunidad de SFML es la que menos preguntas realiza, no lo es tanto saber cuál de las otras dos es la más activa, pues sería tan injusto ignorar los resultados de XNA como sumarlos directamente a MonoGame. No obstante, como es inabarcable ir analizando cada una de las preguntas con la etiqueta «xna» para saber si son aplicables a MonoGame o no, parece lógico considerar a SDL la ganadora, ya que en total se han realizado más de el doble de consultas que en cada una de sus dos competidoras de manera aislada (esto es, sin contar XNA).

7.4. Conclusión

Como se dijo al principio de este documento, la elección de la tecnología en la que se desarrollaría el proyecto se haría en base a cuatro ítems: plataformas soportadas, facilidad de uso, presencia en la industria y actividad de la comunidad. Se han ido estudiando uno por uno y extrayendo conclusiones; a continuación se muestra un resumen:

Plataformas soportadas

Cada una de las tecnologías soporta de manera oficial las siguientes plataformas:

- **SDL:** Windows, Mac OS X, Linux, iOS y Android.
- **SFML:** Windows, Linux y Mac OS X.
- **MonoGame:** Windows, Mac OS X, Linux, iOS, Android, y Windows Phone 8

Facilidad de uso

Como cabía esperar dado el uso de C# y su orientación a videojuegos, MonoGame parece la tecnología que más facilidades da al programador, seguida por SFML y SDL, respectivamente.

Presencia en la industria

SDL es la que más ha sido utilizada tanto en juegos comerciales como independientes. A una distancia considerable se encuentra MonoGame y, en un clarísimo último puesto, SFML.

Actividad de la comunidad

En base a las preguntas realizadas en Stack Overflow y Game Development Stack Exchange, la comunidad más activa en 2013 y 2014 ha sido la de SDL, seguida de la de MonoGame y, nuevamente en último puesto, la de SFML.

Basándose en el análisis realizado, se le dará una puntuación entre 1 y 5 a cada tecnología en cada apartado. Los resultados se recogen en el Cuadro 7.2. La falta de soporte oficial de SDL para Windows Phone 8 no se ha considerado suficiente como para bajarle la puntuación en el apartado de multiplataforma.

	Multiplataforma	Usabilidad	Industria	Comunidad	Total
SDL	5	2	5	5	17
SFML	2'5	3	1	3	10'5
MonoGame	5	4	4	4	17

Cuadro 7.2 – Tecnologías puntuadas por apartados

Es importante resaltar que el empate entre la biblioteca SDL y el framework MonoGame se ha debido a la mayor facilidad de uso de este último, ya que SDL tiene una puntuación perfecta en el resto de apartados. La facilidad de uso no es una característica que deba menospreciarse; sin embargo, como se dijo en la introducción de este apartado, uno de los objetivos del proyecto es adquirir una experiencia en el desarrollo de videojuegos a bajo nivel, lo que unido a que el que subscribe tiene más experiencia programando en C++ que en C#, hace que **la opción elegida sea la combinación del lenguaje** C++ **y la biblioteca *Simple DirectMedia Layer***.

Capítulo 8

Diseño e implementación

Establecidos los requisitos, el alcance y la tecnología que se va a utilizar para el desarrollo (lenguaje C++ con la biblioteca SDL 2.0), ya se puede pasar a las etapas de diseño e implementación del software.

La tarea de diseñar un framework para desarrollo de juegos se antojaba complicada desde un principio, pues se desconocía tanto cómo crear un juego de manera ordenada, como los detalles de la biblioteca SDL, pues el programa de prueba hecho en la comparativa del Capítulo 7 era muy simple. Lo primero que se intentó solventar fue esto último, consultando documentación de SDL de diversas fuentes (mención especial a los tutoriales de Lazy Foo[15]) mientras se construía un pequeño prototipo al que ni siquiera se le puede llamar juego, utilizando para los gráficos dibujos hechos en una libreta.

Figura 8.1 – Captura del primer prototipo creado con SDL.

El prototipo y los ejemplos de la bibliografía consultada sirvieron para comprender la estructura de SDL y familiarizarse con su API, pero a la vez dejaban entrever que si se quería realizar un diseño decente habría que acudir a alguna fuente más extensa. La documentación oficial de SDL tiene una sección dedicada a libros recomendados, de los cuales el primero es *SDL Game Development* de Shaun Mitchell[28], del que se dice lo siguiente:

> «Written as a practical and engaging tutorial, SDL Game Development guides you through the development of your own framework and the creation of two exciting, fully-featured games.»

Si a eso le añadimos que está enfocado a juegos en 2D y que, de los dos ejemplos de aplicación del framework, uno es un juego de plataformas (el otro es un *Shoot 'em up* de desarrollo lateral), no es difícil entender por qué se decidió comprar el libro y seguirlo cuidadosamente, implementando la primera versión del framework en el proceso. En un principio se temió que esto entrara en conflicto con los intereses académicos del proyecto, al estar implementando un framework ya diseñado; sin embargo, ese «miedo» se desvaneció a medida que avanzaba el desarrollo, pues el

resultado final, después de todas las modificaciones y correcciones que se le han hecho (sobretodo tras empezar a construir el editor de niveles), dista tanto del propuesto por Mitchell que sólo son equiparables a grandes rasgos.

Se explica la elección y seguimiento de este libro por dos razones: para hacerle justicia al autor, y para justificar el hecho de que no se haya realizado un diseño al uso. Esto es, no se ha llevado a cabo un diseño de todo el sistema ni se han dibujado esquemas antes de programarlo, ya que se ha ido siguiendo la documentación mientras se implementaba la solución que se proponía, asumiendo y comprendiendo el diseño presentado. Por tanto, los diagramas que se muestran se han realizado mediante ingeniería inversa [29].

8.1. Simple DirectMedia Layer (SDL)

Antes de pasar a describir la solución adoptada, es conveniente dedicarle un espacio al diseño de la propia biblioteca SDL. La versión utilizada para el proyecto fue SDL 2.0, una revisión que supuso un avance importante respecto a la versión 1.2 de la biblioteca, siendo el añadido más llamativo (al menos para nosotros) el soporte para gráficos 2D acelerados por hardware, lo cual ayudará a mejorar el rendimiento de los juegos que se desarrollen con el framework.

La biblioteca SDL está dividida en varios subsistemas que nos ofrecen soporte a distintas partes del hardware. Éstos son:

Subsistema de vídeo Interfaz con el hardware de vídeo. Se encarga de todo lo relacionado con la gestión de ventanas y gráficos.

Subsistema de audio Subsistema encargado de la reproducción de sonidos.

Subsistema de gestión de eventos Es el que permite la interactividad. Principalmente nos permitirá saber el estado del teclado y el ratón en cualquier momento.

Timers SDL nos dará acceso a timers de alta resolución, especialmente útil para hacer que nuestro videojuego trabaje a la misma velocidad con independencia del equipo en el que se ejecute.

Además hay otros subsistemas, como el de *joysticks* o el de hilos, pero los enumerados son los que se utilizarán. Además de los subsistemas, SDL cuenta con extensiones oficiales, bibliotecas adicionales que amplían su funcionalidad. Se han utilizado las siguientes:

SDL_image Por defecto SDL sólo permite cargar imágenes en formato BMP. Con esta extensión tendremos acceso a muchos más formatos de datos, en concreto los siguientes: TGA, BMP, PNM, XPM, XCF, PCX, GIF, JPG, TIF, LBM, PNG. En este proyecto se usarán imágenes con formato PNG.

SDL_mixer El subsistema de audio de SDL es bastante tedioso de usar en comparación con los demás. SDL_mixer facilita la tarea, permitiéndonos reproducir música y efectos de sonido de manera más sencilla.

SDL_ttf Nos permite dibujar textos en pantalla con la fuente que deseemos, siempre que sea ttf (*TrueType font).*

A continuación se explicarán con mayor detalle los aspectos de la biblioteca SDL de más relevancia para nuestro framework.

8.1.1. Inicialización

Para usar SDL antes tenemos que inicializarla, y para ello se utiliza la función SDL_Init(Uint32 flags). Esta función debe ser llamada antes que cualquier otra función. El parámetro flags puede tomar distintos valores según qué parámetro queramos inicializar. Los posibles valores son:

SDL_INIT_TIMER	Subsistema de timer
SDL_INIT_AUDIO	Subsistema de audio
SDL_INIT_VIDEO	Subsistema de video; automáticamente inicializa el subsistema de eventos
SDL_INIT_JOYSTICK	Subsistema de joystick; automáticamente inicializa el subsistema de eventos
SDL_INIT_HAPTIC	Subsistema háptico (feedback forzado)
SDL_INIT_GAMECONTROLLER	Subsistema de mandos; automáticamente inicializa el subsistema de joystick
SDL_INIT_EVENTS	Subsistema de eventos
SDL_INIT_EVERYTHING	Todos los subsistemas de arriba
SDL_INIT_NOPARACHUTE	Esta flag se ignora, está por motivos de compatibilidad

Por tanto, si quisiéramos usar los subsistema de vídeo, audio, timers y eventos, llamaríamos a la función con los siguientes parámetros (recordemos que el subsistema de vídeo carga el de eventos automáticamente):

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMERS);
```

También podemos inicizalizar todos los subsistemas mediante la flag SDL_INIT_EVERYTHING o inicializar los subsistemas por separado. Para esto último llamaríamos primero a SDL_Init(0) seguido de SDL_InitSubsystem() con la flag del subsistema deseado.

8.1.2. Subsistema de video

Ya en el ejemplo que se utilizó para comparar SDL con otras herramientas (epígrafe 7.1.1) pudo verse cómo crear una ventana usando este subsistema, pero se explicará aquí de manera más completa. Las ventanas en SDL se representan mediante una estructura de tipo SDL_Window. Para obtener un puntero a esta estructura se utiliza la siguiente función:

```
SDL_Window* SDL_CreateWindow(const char* title, int x, int y, int w, int h,
   Uint32 flags)
```

Los parámetros son el título de la ventana, la posición (x, y), el ancho, el alto y flags para indicar acciones especiales como que la ventana ocupe la pantalla completa, pueda minimizarse

o carezca de bordes. Como no sabemos en qué pantallas se va a ejecutar el juego, podemos pasar como parámetros x e y las macros SDL_WINDOW_UNDEFINED o SDL_WINDOW_CENTERED, para una posición cualquiera o centrada, respectivamente. Así, si queremos crear una ventana de 640x480 píxeles sin importarnos su posición en pantalla y con el título «Ventana de ejemplo», haremos lo siguiente:

```
SDL_window* window = SDL_CreateWindow("Ventana de ejemplo",
    SDL_WINDOW_UNDEFINED, SDL_WINDOW_UNDEFINED, 640, 480, 0);
```

Si se ha creado correctamente, la función devuelve un puntero a una ventana SDL con las características especificadas; si no, devuelve NULL.

SDL puede usar dos estructuras para dibujar en pantalla. Una es la estructura SDL_Surface (la que usamos en el ejemplo de la comparativa), que contiene una serie de píxeles y se dibuja utilizando procesos de renderizado software (sin usar la GPU). La otra es SDL_Texure; ésta puede ser usada para renderizado acelerado por hardware. Queremos que nuestros juegos sean tan eficiente como sea posible, por lo que nos centraremos en usar SDL_Texture.

Para dibujar texturas en una ventana necesitamos crear una variable de tipo SDL_Renderer. Para crear un *renderer* usaremos la función SDL_CreateRenderer, con la siguiente especificación:

```
SDL_Renderer* SDL_CreateRenderer(SDL_Window* window, int index, Uint32 flags)
```

El primer parámetro es la ventana en la que el renderer va a dibujar. El segundo es el índice del driver de renderizado a utilizar, o -1 para usar el primero que soporte las flags que se le pasen; en nuestro caso estamos trabajando en Windows, y al pasar -1 SDL utilizará automáticamente el driver *direct3d*. El último parámetro son una serie de flags, que pueden tener los siguientes valores:

SDL_RENDERER_SOFTWARE	Usa renderizado por software
SDL_RENDERER_ACCELERATED	Usa aceleración hardware
SDL_RENDERER_PRESENTVSYNC	Sincroniza la tasa de actualizado del renderer con la de la pantalla
SDL_RENDERER_TARGETTEXTURE	Soporta renderizado en una textura

Si queremos crear un renderer para dibujar en la ventana que creamos previamente, utilizando aceleración hardware y con VSYNC, pasaremos a la función los siguientes parámetros:

```
SDL_Renderer renderer = SDL_CreateRenderer(window, -1,
    SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
```

Estas estructuras de datos y funciones son las principales para poner a punto el subsistema de video.

8.1.3. Subsistema de eventos

El subsistema de eventos se inicializa junto al de vídeo, por lo que no es necesario inicializarlo específicamente. La estructura SDL_event es el núcleo de toda la gestión de eventos de SDL. SDL_event es una unión que contiene las estructuras para los distintos tipos de evento usados en SDL. Usarlo es cuestión de saber qué miembro de la unión corresponde a cada tipo de evento. De todos los campos de la estructura, los siguientes son los que más nos interesan:

Tipo de datos	Nombre	Evento
Uint32	type	Tipo del evento. Común a todos los eventos.
SDL_KeyboardEvent	key	Evento de teclado.
SDL_MouseMotionEvent	motion	Evento de movimiento del ratón.
SDL_MouseButtonEvent	button	Evento de botón del ratón.
SDL_QuitEvent	quit	El usuario ha cerrado la ventana del juego.

Como se indica, todos los eventos tienen un campo en común, llamado type. Este campo nos servirá para saber qué tipo de evento se trata y acceder al campo correspondiente con la información específica de ese evento. Cada tipo de evento tiene asociado una estructura, que en ocasiones coincide. A continuación se muestra la relación entre el valor del campo type (constantes definidas por SDL) y la estructura del evento, así como el campo de SDL_event donde se almacena:

Tipo de evento	Estructura	Campo de SDL_Event
KEYDOWN	SDL_KeyboardEvent	key
KEYUP		
SDL_MOUSEMOTION	SDL_MouseMotionEvent	motion
SDL_MOUSEBUTTONDOWN	SDL_MouseButtonEvent	button
SDL_MOUSEBUTTONUP		

Por ejemplo, en el caso del teclado, los tipos de evento KEYDOWN y KEYUP, que nos indican cuando se pulsa o se suelta una tecla, se guardarán en una estructura de tipo SDL_KeyboardEvent, en el campo key de SDL_event. Si quisiéramos conocer más información, como qué tecla es, acudiremos al campo key del evento.

Para capturar los eventos podemos utilizar la función SDL_PollEvent:

```
int SDL_PollEvent(SDL_Event* event)
```

La función devuelve 1 si hay algún evento pendiente o 0 si no. Si hay algún evento disponible se almacena en el parámetro event. En el siguiente código vemos una captura de eventos típica:

```
SDL_Event event;

while (SDL_PollEvent(&event))
{
  switch (event.type)
  {
  case SDL_KEYDOWN:
    printf("Se ha pulsado una tecla\n");
    break;

  case SDL_MOUSEBUTTONDOWN:
    printf("Se ha pulsado un botón del ratón\n");
    break;
  }
}
```

Primero se declara la variable event de tipo SDL_event donde guardaremos la información sobre los eventos. A continuación se utiliza la función SDL_PollEvent, que irá extrayendo uno a uno los eventos de la cola hasta que esté vacía (y devuelva 0) y guardándolos en event. Comparando event.type con las constantes de SDL para cada tipo de evento, sabremos de qué tipo es.

En el ejemplo se indica cuándo se ha producido una pulsación de teclado (SDL_KEYDOWN) o un click de ratón (SDL_MOUSEDOWN). En el primer caso el evento se guardará en el campo key de event, en una estructura SDL_KeyboardEvent; en el segundo se guardará en el campo button, de tipo SDL_MouseButtonEvent.

8.1.4. Subsistema de timers

Usaremos las funciones de este subsistema para controlar el tiempo que transcurre, algo esencial en un videojuego ya que no todos los equipos van a la misma velocidad. La función más común para medir el tiempo en SDL es SDL_GetTicks:

```
Uint32 SDL_GetTicks(void)
```

Nos devuelve el número de milisegundos transcurridos desde que se inició la biblioteca. Otra función importante es la siguiente:

```
void SDL_Delay(Uint32 ms)
```

Utilizaremos SDL_Delay para detener la ejecución del programa durante el número de milisegundos pasados como parámetro. Como mínimo espera el tiempo indicado, pero podría haber un pequeño error (en torno a 10 ms) según el sistema operativo.

Además, la biblioteca cuenta desde 2011 con una API para timing de alta resolución, con precisión superior al milisegundo, pensada principalmente para *profiling*. Las funciones para aprovechar esta nueva API son las siguientes:

```
Uint64 SDL_GetPerformanceCounter(void)
Uint64 SDL_GetPerformanceFrequency(void)
```

SDL_GetPerformanceCounter nos da el valor actual del contador de alta resolución, mientras que SDL_GetPerformanceFrequency nos devuelve la cuenta por segundo de dicho contador. Por tanto, combinando las dos puede saberse el tiempo transcurrido.

8.1.5. Gestión de audio

Como se ha comentado, para la gestión de audio utilizaremos la extensión SDL_mixer. Para inicializarla se utiliza la siguiente función:

```
int Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize)
```

Los parámetros son, por orden: la frecuencia de reproducción del sample (en Hertzios), su formato, el número de canales de para la salida (1 mono, 2 estéreo) y el número de bytes utilizado por el sample. Son menos intuitivos que los vistos hasta ahora y pueden variar dependiendo de las necesidades específicas, pero una llamada a la función con los siguientes valores funcionará correctamente para la mayoría de los juegos

```
Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT, 2, 1024);
```

La función devolverá 0 cuando si no hay errores y -1 en caso contrario. Si la inicialización ha sido exitosa, ya se pueden empezar a cargar archivos de audio, tanto música como efectos de sonido.

SDL_mixer utiliza dos tipos para los sonidos, Mix_Chunk para efectos de sonido y Mix_Music para pistas de música. La diferencia más importante es que varios efectos de sonido pueden reproducirse simultáneamente, pero sólo una pista de música.

8.1.6. Finalizar SDL

Cuando vayamos a terminar el programa o a dejar de usar SDL, debemos cerrarla. De ello se encarga la función SDL_Quit(void), que se encarga de limpiar y cerrar todos los subsistemas inicializados. Para finalizar los subsistemas individualmente se utiliza SDL_QuitSubsystem pasándole la flag del sistema que se desee cerrar.

Antes de finalizar los subsistemas por medio de SDL_Quit, deberemos liberar los recursos que hayamos utilizado. Cada tipo de recurso tendrá asociado una función para liberarlo; como ejemplo se muestran dos de las más importantes, para liberar la memoria usada para la ventana y el renderer:

```
void SDL_DestroyWindow(SDL_Window* window)
void SDL_DestroyRenderer(SDL_Renderer* renderer)
```

8.2. Estructura general del juego

Todo juego consiste en una secuencia de obtener la entrada del usuario, actualizar el estado del juego y mostrarlo en pantalla. Esto es lo que llamamos el bucle de juego o *game loop*[1], que se ejecutará indefinidamente hasta que se salga del juego. Previamente al game loop habrá que iniciar el juego y al terminar habrá que salir del juego y liberar los recursos. En la Figura 8.2 puede verse el diagrama de lo que sería la estructura básica de un juego, en base a la cual se diseñará el framework. Los distintos módulos se irán explicando en sus secciones correspondientes, también indicadas en el diagrama.

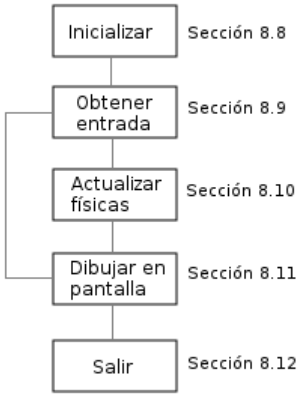


Figura 8.2 – Ejemplo de estructura típica de un juego.

El framework tendrá una clase central llamada Game, que contará con métodos para llevar a cabo las tareas mencionadas: inicialización, game loop y finalización. En la Figura 8.3 vemos

[1]Utilizaremos el término *game loop* por ser el más extendido incluso en la bibliografía en castellano.

los miembros más significativos que toman parte en el proceso. El método init se encarga de la puesta a punto inicial. Los métodos handleEvents, update y render componen el game loop. La finalización del programa y la liberación de recursos serán tarea del destructor. Los métodos quit y running son simples, la primera pone a false la flag m_bRunning y la segunda nos devuelve el valor de ésta; se usan para saber cuándo salir del game loop y finalizar el juego.

Además de los miembros mencionados, podemos ver otros tres atributos: m_pWindow será la ventana SDL en que se ejecute el juego, m_pRenderer el renderer SDL que dibuje las texturas en la ventana, y m_pGameStateMachine será una máquina de estados para controlar los estados del juego. Se explicarán en mayor profundidad en las secciones en que intervengan.

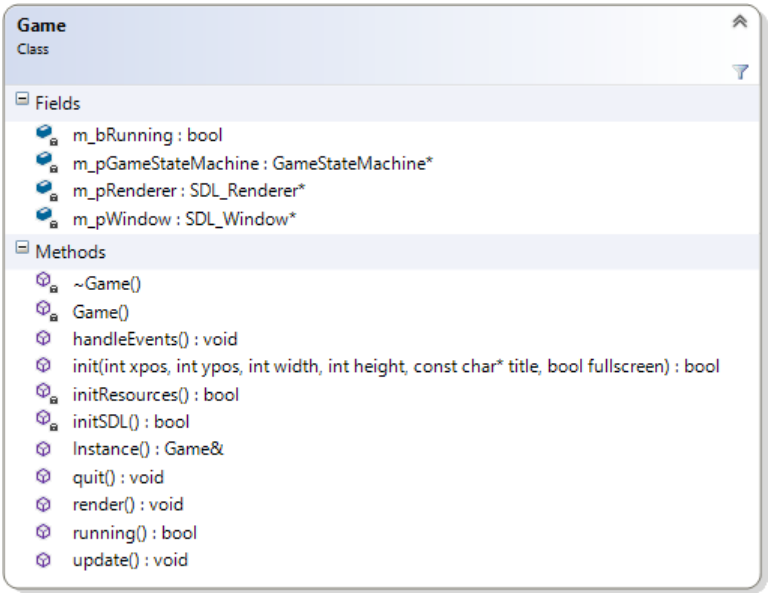


Figura 8.3 – Clase principal del framework: Game

La clase Game se implementará siguiendo el patrón de diseño *singleton*, cuya función es asegurar que una clase tiene una única instancia y proporcionar un punto de acceso global a ella. Esto puede hacerse creando un método (que nosotros llamaremos Instance) que nos devuelva una referencia a la instancia de la clase, creándola antes si es la primera vez que se llama; además habrá que hacer el constructor privado para impedir la creación de nuevas instancias de la clase.

A lo largo del capítulo se presentarán otras clases que también serán implementadas con el patrón singleton. Este patrón genera cierta controversia, y cuenta con sus defensores y detractores debido a sus ventajas e inconvenientes, que no discutiremos aquí. Si se eligió utilizarlo fue porque la bibliografía consultada así lo hacía y porque nos era muy útil para algunos módulos del framework; no obstante, para eliminar parte de sus inconvenientes, se implementó la variación conocida como *singleton de Meyers*[2].

Implementación del patrón singleton

Cada vez que se crea una instancia de un objeto, un constructor debe ser llamado. Por tanto, la manera más sencilla de evitar que los objetos de una clase en particular sean creados es

[2]Llamado así en honor a Scott Meyers, el primero que publicó esa implementación en 1996[30].

declarar el constructor de esa clase como privado. Además, puesto que sólo habrá una instancia, eliminaremos el constructor copia y el operador de asignación, que desde C++11 puede hacerse igualando su declaración a delete.

```
class Game {
public:
  static Game& Instance();
  Game(const Game&) = delete;
  Game& operator=(const Game&) = delete;
  ...
private:
  Game();
  ...
};
```

Con esto ya impedimos la creación de objetos; lo siguiente es limitar el número de instancias de la clase a uno, para lo que utilizaremos un método estático, cuya declaración puede verse en el fragmento de código previo. De esta forma además nos aseguramos de que la instancia es creada la primera vez que se ejecuta el método y, por tanto, si el método nunca es llamado, la instancia no se llega a crear. La definición del método Instance es sencilla:

```
Game& Game::Instance()
{
  static Game instance;
  return instance;
}
```

Cada vez que se entre en el método se comprobará si el objeto necesita ser creado. La primera vez, se creará y se devolverá su referencia; las veces siguientes se comprobará que ya existe y se devolverá su referencia.

Además, para reforzar la idea de que sólo hay una instancia de esta clase, definiremos el tipo TheGame, que será un *alias*[3] de Game:

```
typedef Game TheGame;
```

Ahora ya podemos usar nuestra clase Game como un singleton. Para utilizar sus métodos, simplemente incluimos la cabecera de la clase (Game.h) y los llamamos como sigue (como ejemplo se ha usado el método update):

```
TheGame::Instance().update();
```

8.3. Objetos del juego

Todos los juegos tienen objetos[4], como el avatar del jugador, enemigos, NPC (*non-player characters*) o ítems. Cómo gestionarlos y hacerles un seguimiento es una decisión importante a tomar. Para ello se elegirá un diseño que será un ejemplo claro de aplicación de Programación

[3]Para todos los singleton se creará un alias con «The» seguido del nombre de la clase.

[4]Por objetos nos referimos a entidades. Para referirnos a los objetos que puedes recoger jugando usaremos «ítems».

Orientada a Objetos (OOP), y que empezará a dar sentido a nuestra elección de C++ en lugar de C.

Muchos de los objetos compartirán datos y funciones básicas, como puedan ser actualizar el estado del objeto o dibujarlo en pantalla. Por tanto crearemos una clase abstracta llamada GameObject, con los miembros que se muestran en la Figura 8.4. GameObject es una de las clases más complejas de S2Dp, por lo que el número de miembros aumentó significativamente a lo largo del desarrollo del proyecto, a medida que se iban añadiendo nuevas funcionales al framework y al editor de niveles. Los métodos comunes a todos los objetos que cubre esta primera versión de la clase son:

- **Dibujado:** El método virtual puro draw se encargará de mostrar el objeto en pantalla.
- **Actualización:** Habrá objetos estáticos, pero la mayoría de instancias de objetos que pueblen nuestro juego necesitarán actualizarse de algún modo, lo que haremos mediante el método virtual puro update.
- **Carga de atributos:** Mediante el método virtual puro load, se indicará el valor de los atributos principales del objeto. LoaderParams es una clase auxiliar para cargar estos atributos, que se explicará más adelante en esta misma sección.
- **Identificación:** Las clases derivadas de GameObject implementarán el método virtual puro type para saber de qué tipo es el objeto. Aunque C++ puede extraer el tipo de un objeto en tiempo de ejecución mediante sus mecanismos de *Run-time Type Identification* (RTTI), tener una función que nos devuelva el nombre del objeto puede ser útil en muchos casos, o si simplemente no quiere usarse RTTI.
- **Liberar recursos**: No debería hacer falta que los objetos lo hagan, porque los recursos son gestionados por una serie de *managers* que se explicarán más adelante, pero si por cualquier razón fuera necesario que los objetos realizaran algún tipo de limpieza, lo harán en el destructor.

Queremos que todos los objetos implementen estos métodos, por lo que los declararemos como virtuales puros (exceptuando el constructor y el destructor).

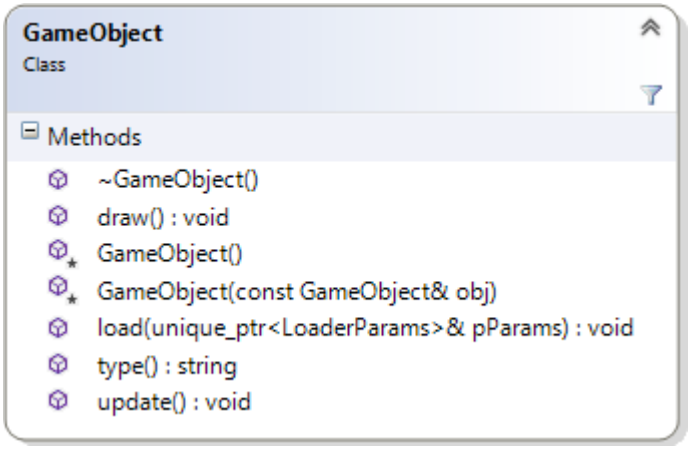


Figura 8.4 – Métodos comunes a todos los GameObject.

Como se comentó en el epígrafe 6.3.4 del análisis, en el framework implementaremos una batería de objetos con comportamientos más o menos típicos. Por supuesto un futuro usuario de S2Dp puede añadir y modificar los objetos que quiera, adaptándolos al tipo de juego que vaya a hacer, pero hay una serie de objetos que es común ver en la mayoría de juegos (especialmente en plataformas 2D) y que darán lugar a un número de clases:

- **Avatar del jugador:** Nos referiremos al avatar controlado por el jugador como *player* o simplemente «jugador», y en consecuencia llamaremos la clase Player.
- **Enemigos**: Crearemos una clase básica Enemy de la que heredarán a su vez los cuatro tipos de enemigo descritos en el análisis: LeftRight, Flying, Chaser y FlyingChaser.
- **Ítems:** La clase Item representa objetos que puede recoger el jugador. Además se ha hecho que hereden de la clase otros objetos que, aunque no son lo que solemos entender por ítems, comparten comportamiento con éstos (simplemente están quietos esperando a detectar una colisión con el jugador), por lo que conviene agruparlos ya que sus métodos son idénticos exceptuando type (ver Figura 8.4). Las clases que heredan de la clase Item son: LivesItem, HealthItem, ScoreItem, KeyItem, LockedDoor, CheckPoint y LevelFinish.

En la Figura 8.6 pueden verse estas clases derivadas de la clase abstracta principal GameObject. Haciendo uso del polimorfismo, ahora nuestra clase Game podrá tener un contenedor de punteros a GameObject y en Game::update y Game::draw simplemente tendremos que recorrer dicho contenedor llamando a las funciones draw y update específicas de cada uno de los *game objects* herederos. En la siguiente Sección 8.4 introduciremos los estados del juego y veremos que esto no es exactamente así, pues en lugar de la clase Game será el estado actual el que tenga el contenedor de objetos, pero el concepto es el mismo.

Sabiendo las clases que heredarán de GameObject, podemos decidir qué atributos tendrá la clase base. De cada objeto nos hará falta saber la posición, la imagen que dibujar (en adelante *sprite*) y la velocidad. Además necesitamos saber los sonidos que usará y algunos valores numéricos especiales, que tendrán diferente significado según el objeto. A continuación se explica qué atributo cubre cada una de estas necesidades:

Posición El atributo m_position contendrá la posición bidimensional del objeto. Será de la clase Vector2D, creada para gestionar operaciones vectoriales comunes.

Sprite El sprite que dibujar variará según la animación que esté realizando el objeto. La clase m_animations será un mapa asociativo de pares *clave-valor* con las animaciones de cada objeto. La clave será el nombre o ID de la animación y el valor un objeto de tipo Animation, clase que construiremos para almacenar información sobre una animación: imagen que la contiene, número de frames, etc.

Velocidad La velocidad inicial del objeto en los ejes x e y nos vendrá dada por m_xSpeed y m_ySpeed.

Sonidos De los objetos que se crearán, ninguno necesita más de dos sonidos (ver Sección 6.3.5). Declararemos dos atributos de texto genérico que contendrán la ID de dos sonidos. En la función load de las clases derivadas se asignarán esos sonidos a los atributos específicos que correspondan.

Valores especiales De manera similar a como hacemos con el sonido, tendremos tres valores numéricos genéricos que las clases derivadas asignarán a sus atributos específicos.

En la Figura 8.5 vemos el nombre y el tipo de los atributos de GameObject. Puede parecer engorroso tener atributos genéricos como m_value2, m_value3 o m_snd2ID, cuando realmente no *todos* los objetos los necesitan y podrían declararse directamente en las clases derivadas, con el nombre del atributo específico. Por ejemplo, los únicos objetos que necesitarán tres valores especiales son Player (longitud del salto, velocidad del salto y tiempo de invulnerabilidad tras recibir daño) y Chaser (vida, daño y campo de visión), al igual que hay algunos que sólo necesitan uno, como ScoreItem (cantidad de puntuación que suma).

Si se ha tomado la decisión de ponerlos en la clase base, ha sido debido a que estos atributos principales serán los que se puedan configurar desde ficheros externos (como se verá en la Sección 8.6) o a través de la GUI del editor, sin modificar el código inicial. Cuando estemos llevando a cabo la serialización para pasar un nivel diseñado de la estructura de datos a un fichero XML o cuando estemos parseando un archivo para crear los objetos necesarios, los trataremos de forma genérica sin saber qué clase de GameObject serán. De no ser así, habría que diferenciar la lectura o escritura del objeto según su subtipo, aumentando el riesgo de errores al intentar escribir o leer atributos que el objeto no posee. Se consideró que la opción elegida haría mucho más sencilla y limpia la tarea.

Estos atributos, sumados a los miembros que ya habíamos descrito (Figura 8.4), compondrán los miembros principales de la clase GameObject. Sin duda nos harán falta más miembros auxiliares, tanto métodos como atributos, por ejemplo para saber si el objeto es sólido, si hay que actualizarlo (podría no haber entrado en juego aún), si ha muerto en la iteración anterior del juego (y por tanto debemos destruirlo), etc., pero los descritos formarán el núcleo de la clase.

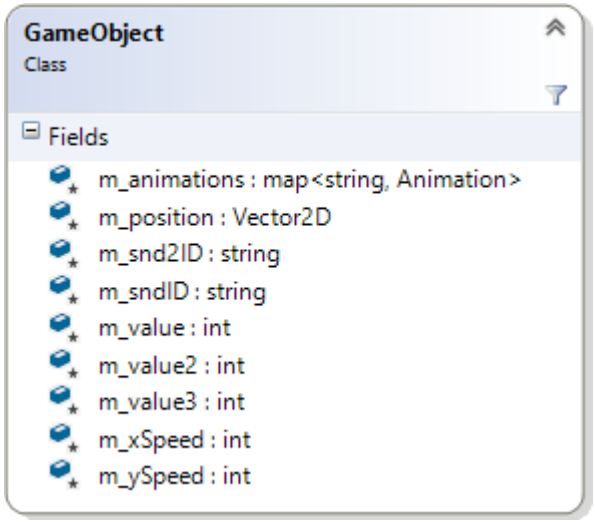


Figura 8.5 – Atributos principales de GameObject.

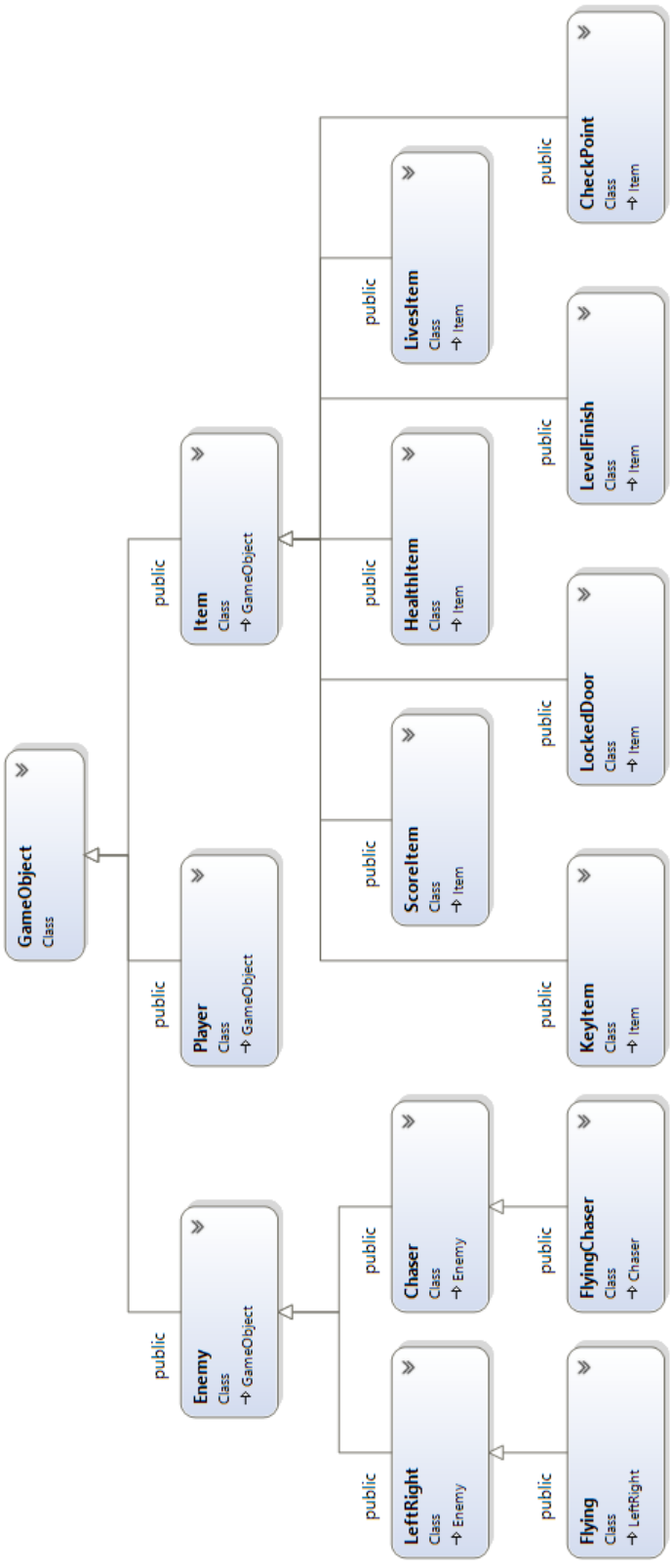


Figura 8.6 – Herederos de la clase GameObject.

Implementación de los objetos

Se explicarán detalles del comportamiento y la implementación de los métodos más importantes de GameObject en las secciones posteriores en que tomen parte, por ejemplo update en la sección de actualizado (8.10) o render en la sección de dibujado (8.11). Explicaremos sin embargo aquí la necesidad del método load y de la clase auxiliar LoaderParams, pues es sensato preguntarse por qué no darle a los atributos el valor correcto directamente en el constructor.

Empezaremos por la clase LoaderParams, que como su nombre indica está destinada a la carga de parámetros. Esta clase (ver Figura 8.7) tiene como atributos los mismos atributos principales de GameObject mostrados en la Figura 8.5 (con la salvedad de que en lugar de tener un Vector2D para la posición, tiene dos valores enteros x e y). Además tiene un constructor al que hay que pasarle esos atributos y un *getter* por cada atributo, de los que en la Figura 8.7 sólo se muestran algunos por economizar espacio.

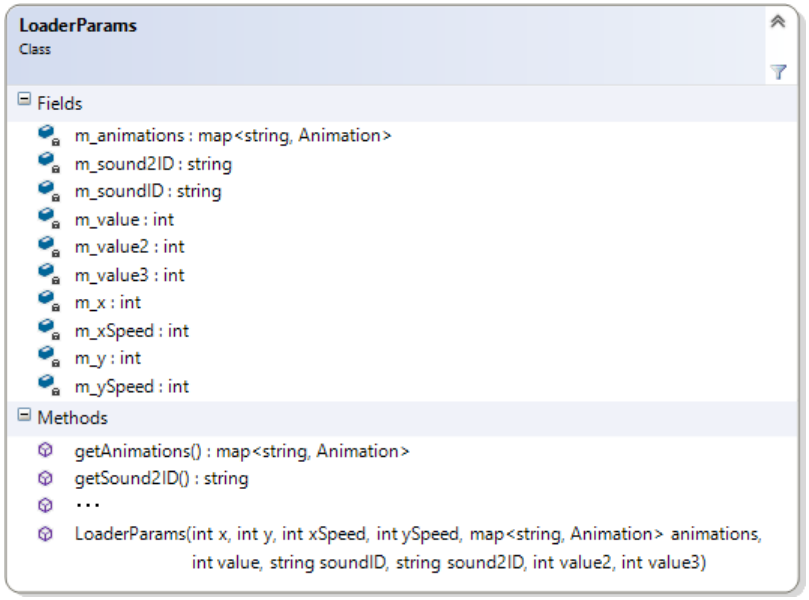


Figura 8.7 – Clase LoaderParams.

El propósito de la clase LoaderParams es facilitar los cambios que se quieran realizar en el framework, en concreto en la clase GameObject y, más específicamente, en sus atributos. Supongamos que prescindimos de esta clase y, por tanto, el método load tiene la siguiente especificación:

```
virtual void GameObject::load(int x, int y, int xSpeed, int ySpeed, std::map<
    std::string, Animation> animations, int value, std::string soundID, std::
    string sound2ID, int value2, int value3) = 0;
```

El problema viene cuando queramos modificar los atributos principales de GameObject, algo que con total seguridad ocurrirá si se usa para crear algún juego con características distintas a los creados para este proyecto[5]. Cuando ocurra esto, tendríamos que modificar los métodos load de todas las clases que deriven de GameObject, lo cual es más tedioso que modificar simplemente la clase LoaderParams. Por eso se ha declarado el método GameObject::load de la siguiente manera:

```
virtual void load(std::unique_ptr<LoaderParams> const &pParams) = 0;
```

[5]De hecho, durante el desarrollo del PFC estos atributos cambiaron varias veces.

Así, si cambiamos los atributos principales de GameObject, sólo tendremos que trasladar esos cambios a LoaderParams. Se ha utilizado un std::unique_ptr, clase introducida en el estándar C++11 que asegura que el destructor de su clase será implícitamente invocado cuando se abandone el ámbito en que fue creado.

Esto explica la utilidad de LoaderParams, pero no por qué se utiliza una función load en lugar de pasar directamente LoaderParams al constructor. Esto es así debido a que, como se verá en la Sección 8.6, los GameObject se crearán a través de una factoría de objetos, y el valor de sus atributos principales se extraerá de un fichero externo. Si definiéramos el valor de los atributos en el constructor, habría que estar pasando LoaderParams como argumento a través de las funciones de la factoría. Teniendo un método load para *cargar* los atributos principales, nos evitamos esto, pues podemos implementar un constructor sin parámetros que cree el objeto inicializando sus atributos a los valores por defecto y, una vez creada la instancia del objeto por la factoría, llamar al método load para darles el valor que finalmente tendrán.

GameObject::load se ha implementado como sigue:

```
void GameObject::load(unique_ptr<LoaderParams> const &pParams)
{
  m_position = Vector2D(pParams->getX(), pParams->getY());
  m_animations = pParams->getAnimations();
  m_xSpeed = pParams->getXSpeed();
  m_ySpeed = pParams->getYSpeed();
  m_value = pParams->getValue();
  m_value2 = pParams->getValue2();
  m_value3 = pParams->getValue3();
  m_sndID = pParams->getSoundID();
  m_snd2ID = pParams->getSound2ID();
}
```

El código es sencillo, simplemente se carga el valor adecuado en los atributos de GameObject a través de los *getters* de LoaderParams. Como puede observarse, son los mismos atributos mostrados en la Figura 8.5.

8.3.1. Animaciones

Podemos definir una animación como el proceso utilizado para crear sensación de movimiento a través de la sucesión de una secuencia de imágenes. Cada una de estas imágenes será un *frame* de la animación; en el cine estos frames toman la forma de fotografías, mientras que un juego en 2D son sprites. Todos los frames de una animación reunidos en una imagen formarían lo que conocemos como hoja de sprites o *sprite sheet.* Normalmente, una sprite sheet no se limita a aglutinar los frames de una única animación, sino que contiene varias; en la Figura 8.8 podemos ver un ejemplo:

La creación de sprites es una disciplina compleja que muchos consideran un arte en sí misma (no en vano existe el término *pixel art*), y es más propia de un diseñador o dibujante que de un programador; por ello, para el proyecto no se construyeron sprites propios, sino que se utilizaron sprites creados por artistas que hayan tenido a bien ofrecerlos libres de derechos. Pueden consultarse junto al resto de recursos utilizados en el Apéndice B.

Cuando vayamos a dibujar los objetos del juego, tenemos que saber qué sprite pintar, que dependerá de la animación actual del objeto y del frame de dicha animación que corresponda

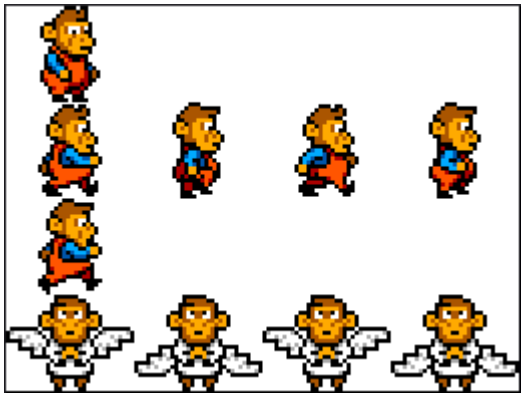


Figura 8.8 – Ejemplo de hoja de sprites.

mostrar. Para gestionar esta información crearemos una clase Animation, con los miembros que se muestran en la Figura 8.9 y se comentan a continuación:

m_textureID ID de la imagen con la sprite sheet de la animación.

m_frames Vector de frames de la aplicación. El tipo SDL_Rect nos da las coordenadas de un rectángulo, que nos indicarán la posición relativa del frame en la sprite sheet.

m_currentFrame Índice del vector de frames correspondiente al frame actual, es decir, el que hay que dibujar.

m_spriteSheetRow Fila de la sprite sheet que contiene los frames de la animación.

m_frameTime Tiempo de espera entre frames o, lo que es lo mismo, velocidad a la que cambiamos de uno a otro.

m_collider Indica la superficie de colisión (o *hitbox*) del objeto cuando está realizando esta animación.

Animation(...) Constructor al que le pasamos como parámetros el valor que tendrán los atributos recién descritos, excepto currentFrame que se va calculando automáticamente.

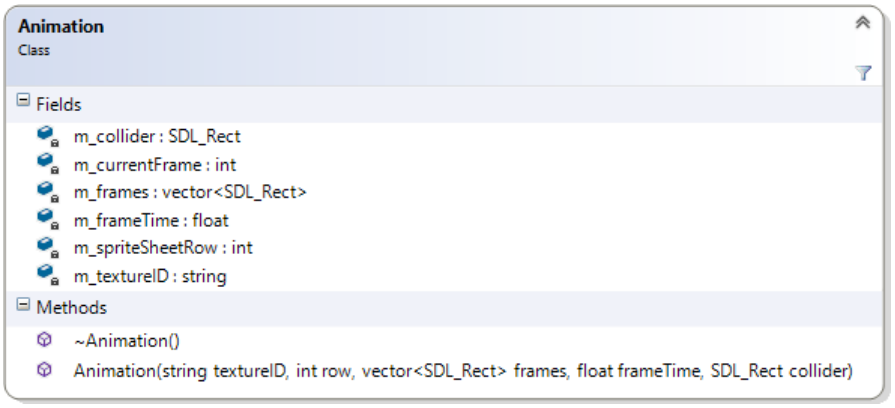


Figura 8.9 – Miembros de la clase Animation.

Al inicio de la sección, presentando los miembros de los objetos del juego decíamos que tendríamos un atributo m_animations, un mapa asociativo para almacenar las animaciones de cada objeto. De esta forma no se restringe a cada objeto un número de animaciones concreto. No obstante, para poder programar el código de los objetos que tendrá el framework de base, debemos definir qué animaciones tiene cada objeto, acordes a su comportamiento descrito en el análisis (apartado 6.3.4). Serán las siguientes:

- Los **enemigos** tendrán dos animaciones, una para cuando estén quietos y otra para cuando caminen. La ID de estas animaciones será, respectivamente, «idle» y «walk».
- Hay una serie de **objetos inmóviles** (clase Item y sus derivadas). Estos objetos sólo tendrán la animación «idle».
- El **player** será el que más animaciones tiene, ya que es el que más acciones puede realizar. Además de «idle» y «walk» para cuando está quieto y en movimiento, tendrá «jump» y «death», para el salto y la muerte.

Así, el atributo m_animations de un objeto Item tendrá únicamente un miembro, una Animation cuya ID es «idle», y sólo tendrá que preocuparse de qué frame mostrar. Sin embargo, un objeto Player tendrá un mapa de cuatro animaciones con las IDs mencionadas, por lo que además del frame habrá que decidir qué animación mostrar en cada momento.

Si se quisiera aumentar la funcionalidad del framework añadiendo, por ejemplo, una animación «agachado» para el jugador, habría que modificar el código de Player para especificar en qué circunstancias mostrar la nueva animación (por ejemplo cuando el usuario pulsara la flecha abajo en el teclado).

Implementación de las animaciones

Sabiendo lo comentado en los párrafos anteriores, la implementación de la clase Animation no tiene mucho misterio, pues se compone únicamente de un constructor y una serie de *getters* para obtener información de la clase.

El cuerpo del constructor está vacío y su única función es inicializar los atributos de la clase:

```
Animation::Animation(std::string textureID, int row, std::vector<SDL_Rect>
    frames, float frameTime, SDL_Rect collider) :
m_textureID{ textureID },
m_spriteSheetRow{ row },
m_frames{ frames },
m_frameTime{ frameTime },
m_collider{ collider }
{
}
```

Como parámetros se le pasan los valores de todos los atributos de la clase (Figura 8.9) excepto m_currentFrame, que va cambiando según el tiempo que haya pasado desde que comenzó la animación.

Además se implementarán *getters* para obtener información de la animación. Estos *getters* son métodos muy sencillos que simplemente nos dan información del estado de los miembros privados de la clase. En algunos casos devolverán directamente el valor de un atributo, por ejemplo si queremos obtener el vector con todos los frames de la animación:

```
const std::vector<SDL_Rect> Animation::getFrames()
{
  return m_frames;
}
```

En otros casos será información derivada, como muestra el siguiente método para obtener únicamente el frame actual de la animación:

```
const SDL_Rect Animation::getCurrentFrame()
{
  if (m_frames.size())
  {
    return m_frames[m_currentFrame];
  }
  else return{ 0, 0, 0, 0 };
}
```

Debido a su sencillez, estos *getters* no se incluyeron en el diagrama de la Figura 8.9, y por la misma razón no se mostrará aquí la implementación de todos. De hecho, las clases que se han comentado hasta ahora (y muchas de las que se comentarán) también tienen *getters* y *setters* que se han obviado, pero en el caso de Animation, al ser pequeña, se ha optado por explicarlos brevemente.

8.4. Estados del juego

Cuando ejecutamos un juego por primera vez, esperamos ver algún tipo de pantalla con información sobre desarrolladores y distribuidores, seguida quizá de una pantalla de inicio que puede ser estática o dinámica (un menú). Al iniciar el juego, podemos pausarlo, salir, ver una pantalla de fin del juego si perdemos las vidas, etc. Todas estas secciones diferentes son lo que llamamos estados del juego o *game states*, y debemos diseñar un mecanismo para que la transición entre ellos sea lo más cómoda posible.

La manera de cambiar de un estado a otro será mediante un autómata finito, en adelante FSM de sus siglas en inglés *Finite State Machine*. Nuestra FSM podrá tener un número finito de estados y realizar cambios de un estado a otro (conocidos como transiciones entre estados), de los cuáles sólo uno podrá estar activo (conocido como el estado actual). De esta manera podremos definir estados fuera de la clase Game, y que cada uno de ellos se encargue de cargar, actualizar y dibujar según sus necesidades.

Para representar los estados se creará una clase abstracta GameState, cuyo miembros principales vemos en la Figura 8.11. El atributo m_gameObjects será un contenedor con punteros a los objetos del juego que gestione el estado. Los métodos update y render, encargados de actualizar y dibujar los objetos del estado, son métodos virtuales puros que por tanto deberán ser implementados por todas las clases derivadas de GameState, que serán los distintos estados de que disponga el juego. En el constructor se cargarán los recursos que necesite el estado actual y en el destructor se liberarán. Por último, los miembros m_isValid, getIsValid, setIsValid y dequeState se usarán para llevar a cabo un borrado seguro de los estados; se dará más información sobre esto dentro de unos párrafos, cuando se explique la implementación.

La FSM deberá poder realizar las siguientes acciones con los estados:

- **Borrar un estado y añadir otro**. Esta opción se usará para cambiar completamente de estado, sin dar la opción de volver al anterior. Por ejemplo cuando se llega a la pantalla de game over.
- **Añadir un estado sin borrar el anterior**. Útil para pausar el juego principalmente.
- **Borrar un estado sin añadir ninguno**. De esta forma podremos borrar el estado de pausa o cualquiera que haya sido puesto encima de otro.

Se creará una clase GameStateMachine (Figura 8.10) que cumpla estas características. La clase tendrá un atributo m_gameStates que será un vector de punteros a los estados del juego, de los cuales el último en haber sido añadido será el actual. Además tendrá los métodos update y render para actualizar y dibujar el estado actual.

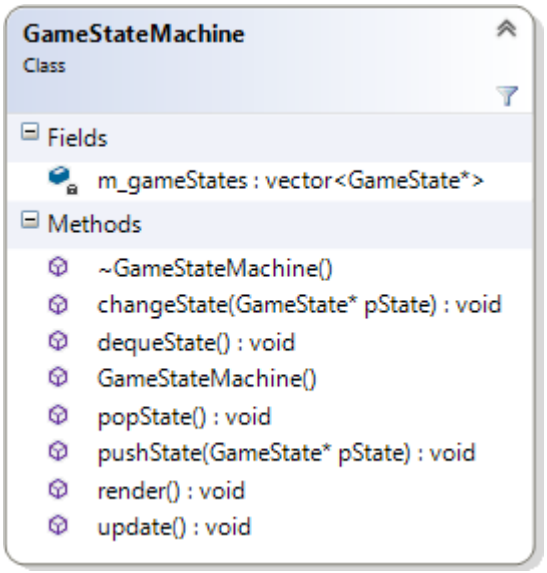


Figura 8.10 – Clase GameStateMachine.

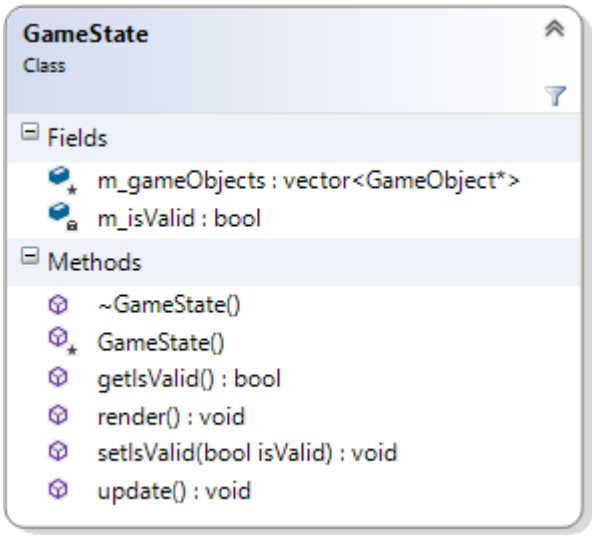


Figura 8.11 – Clase GameState.

Si volvemos a la Figura 8.3, veremos que la clase central Game tiene un atributo m_pGameStateMachine de tipo GameStateMachine; este atributo será la FSM de nuestro juego. De esta forma, cuando se llame a los métodos Game::update y Game::render, éstos llamarán a m_pGameStateMachine::update y m_pGameStateMachine::render, que a su vez llamarán a las funciones update y render del estado actual, que será una clase derivada de GameState.

El resto de métodos de GameStateMachine (quitando el destructor y el constructor) se encargarán de realizar las acciones que enumeramos hace un momento. La función pushState añadirá un estado sin borrar el anterior, changeState borrará el estado actual antes de añadir uno nuevo y, por último, popState eliminará el estado actual sin añadir ninguno.

Cada usuario puede implementar los estados que quiera y establecer las condiciones de las transiciones entre ellos, así como sus funciones de actualización y renderizado. No obstante, se implementarán de inicio cinco estados (ver apartado 6.3.3) para darle al framework un comportamiento básico:

- **StartState**: Estado inicial del juego, compuesto por una imagen estática que se mostrará nada más ejecutarlo. Al pulsar la tecla SPACE iniciaremos el juego, pasando al estado PlayState.

- **PlayState**: Estado principal, en el que se desarrollará la acción del juego.
- **PauseState**: Estado de pausa, al que accedemos pulsando la tecla ENTER desde PlayState. Para reanudar el juego, volver a pulsar ENTER.
- **EndState**: Pantalla final del juego, a la que accederemos al llegar al final del último nivel. Si pulsamos SPACE reiniciaremos el juego.
- **GameOverState**: El jugador ha perdido todas las vidas. Pulsar SPACE para reiniciar el juego.

Además, en cada estado se deberá controlar que si se pulsa la tecla ESCAPE salgamos del juego. Todos los estados heredarán de la clase abstracta GameState (Figura 8.12) y, como se dijo previamente, tendrán que implementar los métodos update y render.

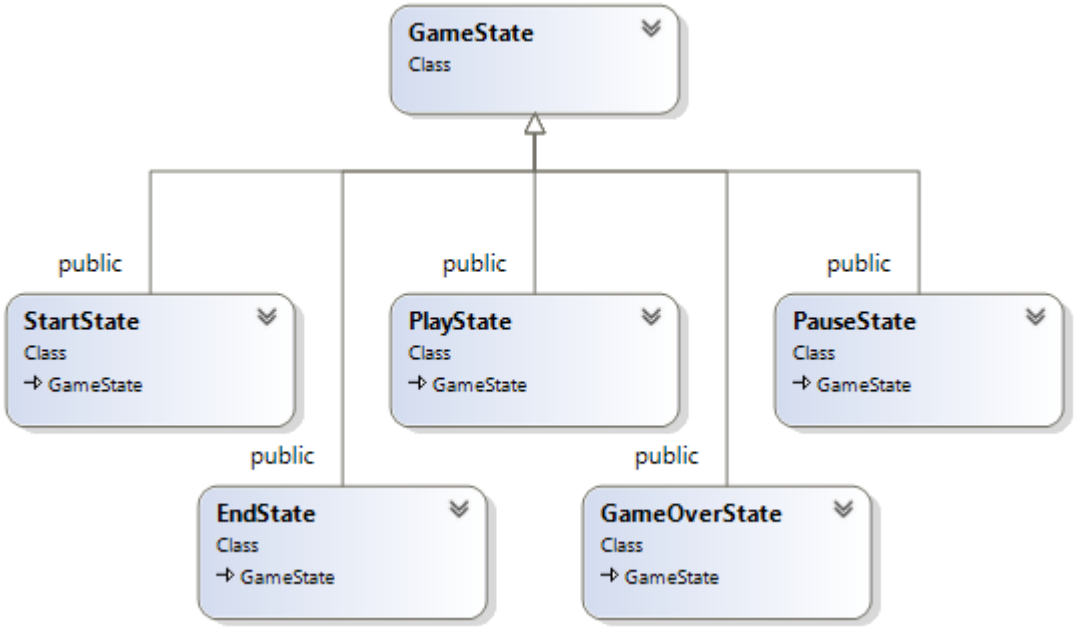


Figura 8.12 – Clase abstracta GameState y sus herederos.

Implementación de los estados

A continuación podemos ver la implementación de GameState, la clase abstracta para representar estados del juego:

```
class GameState
{
public:
  virtual ~GameState(){}

  virtual void update() = 0;
  virtual void render() = 0;

  bool getIsValid() const { return m_isValid; }
```

```
  void setIsValid(bool isValid) { m_isValid = isValid; }

protected:
  GameState() : isValid(true){}

  std::vector<GameObject*> m_gameObjects;

private:
  bool m_isValid;
};
```

Como se puede ver, los métodos update y render se han definido como virtuales puros y, por tanto, serán las clases derivadas las que tengan que definirlos según el estado del juego del que se trate.

Se ha incorporado el atributo m_isValid debido a que cuando queremos eliminar un estado de la FSM, no podemos hacerlo directamente, ya que pueden quedar elementos del estado por procesar. Para entenderlo, imaginemos por ejemplo que se creara un estado MenuState para el menú principal, con dos botones: *start* y *options*. Cada uno de estos botones sería un GameObject y estaría dentro del vector m_gameObjects. Una implementación del método update para actualizarlos podría ser:

```
void MenuState::update()
{
  for (auto object : m_gameObjects)
  {
    object->update();
  }
}
```

Si el usuario pulsa el botón *start* deberíamos pasar al siguiente estado mediante GameStateMachine::changeState. ¿Qué ocurriría si la FSM directamente borra el MenuState? El método MenuState::update no ha terminado, y en la siguiente iteración del bucle intentaría actualizar el botón *options*, el cual es parte de una estructura que ya no existe, lo que causaría comportamientos indefinidos que en muchos casos derivarían en errores graves. En resumen, se realizarían llamadas a métodos de objetos previamente borrados.

Para solucionar este problema, en lugar de eliminarlo, la FSM marcará el estado como no válido. Posteriormente podremos llamar al método GameStateMachine::dequeState para borrar los estados no válidos de forma segura; en este ejemplo, haríamos la llamada después del bucle. Recordemos que podemos acceder a la FSM mediante el método getStateMachine de la clase Game, que es un singleton.

```
void MenuState::update()
{
  for (auto object : m_gameObjects)
  {
    object->update();
  }
  TheGame::Instance().getStateMachine()->dequeState();
}
```

De esta forma nos aseguramos que cuando se borre el estado MenuState, ya habrán terminado de procesarse todos sus objetos. A continuación pasamos a explicar la implementación de los

métodos pushState, popState, changeState y dequeState de la FSM, lo que ayudará a ver esto más claro.

Empezaremos con pushState, que añade un estado sin borrar el anterior. Un ejemplo de uso claro es cuando pausamos el juego, momento en que hay que pasar al estado de pausa (PauseState) conservando el estado anterior (PlayState).

```
void GameStateMachine::pushState(GameState *pState)
{
  m_gameStates.push_back(pState);
}
```

Este método es el más sencillo, simplemente añadimos el nuevo estado al final del vector m_gameStates, atributo que contiene los estados de la FSM. Esto lo convertirá en el estado actual.

Para borrar un estado sin añadir ninguno se utiliza el método popState. Esta acción se realizaría por ejemplo para reanudar el juego, volviendo de PauseState a PlayState.

```
void GameStateMachine::popState()
{
  if (!m_gameStates.empty())
  {
    if (m_gameStates.back()->getIsValid())
    {
      m_gameStates.back()->setIsValid(false);
    }
  }
}
```

Esta función comprueba si existe algún estado en la FSM y, de ser así, lo marca como no válido.

Si lo que queremos es realizar un cambio total de estado, es decir, añadir un estado borrando el anterior, usaremos el método changeState. Este será probablemente el método más común para realizar transiciones entre estados, algunos ejemplos son: pasar de StartState a PlayState, de PlayState a GameOverState o de GameOverState a StartState.

```
void GameStateMachine::changeState(GameState *pState)
{
  if (!m_gameStates.empty())
  {
    if (m_gameStates.back()->getIsValid())
    {
      m_gameStates.back()->setIsValid(false);
    }
  }

  m_gameStates.push_back(pState);
}
```

Este método es una combinación de popState y pushState: comprobamos si hay estados, de ser así marcamos como no válido el estado actual y, por último, añadimos el nuevo estado al vector m_gameStates.

Hasta ahora sólo hemos modificado la validez de los estados, pero no se ha eliminado ninguno. De esto, como se ha comentado, se encargará la función dequeState, a la que llamaremos cuando estemos seguros de que no quedan elementos del estado por procesar.

```
void GameStateMachine::dequeState()
{
  for (auto it = m_gameStates.begin(); it != m_gameStates.end();)
  {
    if (!(*it)->getIsValid())
    {
      delete *it;
      it = m_gameStates.erase(it);
    }
    else
    {
      ++it;
    }
  }
}
```

Lo que hace este método es recorrer el vector de estados por medio de un iterador, borrando los estados marcados como no válidos.

8.5. Niveles y escenarios

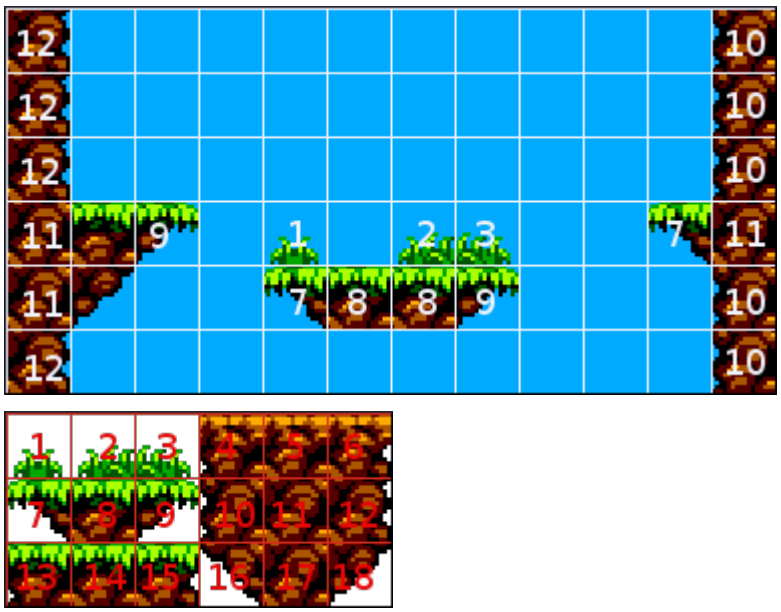
Otro aspecto importante del diseño del juego son los niveles, que estarán compuestos del escenario y los objetos que lo pueblan. Puesto que ya hemos hablado de los objetos, nos centraremos en los escenarios.

La mayor parte de los juegos en 2D usan mapas de tiles, una forma eficiente y rápida de desarrollar escenarios 2D complejos. Veamos un ejemplo de un mapa desarrollado con este método:

Este mapa de 12 x 6 tiles ha sido creado con el siguiente conjunto de tiles, lo que en adelante llamaremos *tileset*:

De momento ya puede verse que una de las ventajas de este sistema es que se pueden crear grandes mapas a partir de imágenes relativamente pequeñas. Los mapas de tiles son en esencia

un vector bidimensional de IDs que nos indican qué parte del tileset dibujar en cada lugar. Para dibujar el mapa, iteraremos a través del número de columnas y filas, usando la ID para seleccionar cada tile y dibujarla en la pantalla. Las tiles que tengan una ID de cero no se dibujarán, serán tiles nulas o tiles «en blanco». A continuación vemos el tileset y el mapa de tiles que se han usado de ejemplo, esta vez con las tiles numeradas:



Indicar una a una las tiles que tiene un mapa es muy lento. Para crear un escenario sencillo de 30 tiles de ancho y 20 de alto, tendríamos que introducir manualmente 600 valores, además de todos los objetos que haya en el nivel con sus atributos. Cuando se cree el editor de niveles podremos indicar gráficamente las tiles y los objetos de cada nivel, pero en este momento del desarrollo del proyecto evidentemente no se había construido, y se necesitaba una forma de agilizar la creación de niveles para poder validar el diseño y la implementación. Para esto se usó el software Tiled, una herramienta desarrollada por Thorbjørn Lindeijer y una amplia comunidad *open source.*

Tiled se define como un «editor de mapas flexible y fácil de usar»[6], y en esencia es eso. Nos permite crear mapas que están compuestos de capas de dos tipos:

- **Capa de tiles** o ***tile layer***: Serán lo que hemos llamado el escenario. Podremos definir tilesets a partir de las imágenes que subamos y usar las tiles para poblar el escenario.
- **Capa de objetos** o ***object layer***: Contendrán objetos del nivel. Para los objetos se utiliza un método genérico en el que podemos definir propiedades y su valor (serán siempre de texto), con lo que podemos crearnos los objetos que vayamos a usar e introducir sus atributos como propiedades.

Los mapas o niveles creados con Tiled se guardarán en archivos XML, que en concreto tendrán el formato TMX (*Tiled Map XML).* Habrá que construir un *parser* para leer estos archivos y convertir la información que contienen en los niveles de nuestro framework. A medida que avanzaba el proyecto, especialmente en la parte del editor, se fue modificando el formato de los archivos y aunque la estructura general conserva ciertas similitudes con TMX, son drásticamente distintos, por lo que no se comentará aquí el formato TMX.

[6] http://www.mapeditor.org/

La clase que representará los niveles en nuestro framework se llamará Level (Figura 8.13), y también contendrá capas que podrán ser de tipo TileLayer u ObjectLayer, derivadas de la clase abstracta Layer (Figura 8.14). Cada nivel se encargará de actualizar y dibujar sus capas, para lo que tendrá los métodos update y render, que llamarán a sendos métodos de la clase Layer.

Además de un vector de Layer, la clase Level tendrá como atributos la ID de la pista de música (ver epígrafe 6.3.5) y un puntero al objeto Player del nivel[7].

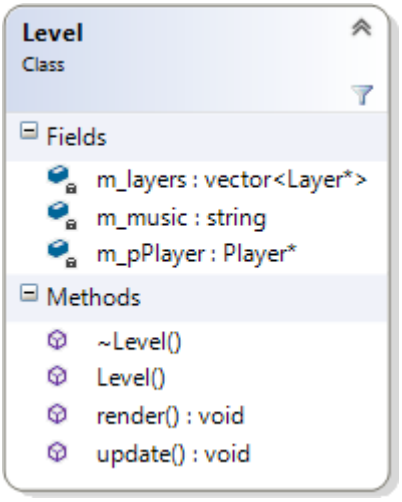


Figura 8.13 – Miembros principales de la clase Level.

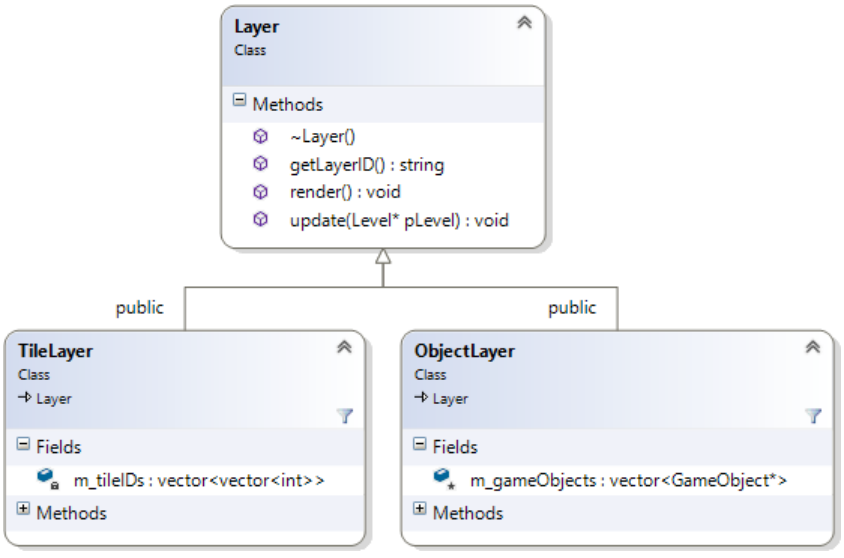


Figura 8.14 – Clase Layer y derivadas.

La clase abstracta Layer tendrá los métodos virtuales puros update, render y getLayerID. Los primeros como de costumbre actualizarán y dibujarán la capa, y el último nos servirá para saber si es una capa de tiles u objetos, de manera análoga a GameObject::type.

El miembro principal de TileLayer será m_tileIDs, un vector bidimensional de enteros para almacenar la ID de cada tile del escenario. Por su parte, GameObjects tendrá un vector de Ga-

[7]Lo normal es que cada nivel tenga un player; si no, el puntero será nulo.

meObject* al que hemos llamado m_gameObjects, con todos los objetos de la capa (que, si no hay más de una capa de objetos, serán todos los objetos del nivel).

Además, para saber a qué tiles se refieren las IDs de la capa de tiles, necesitamos una estructura para almacenar tilesets (Figura 8.15). Nos vale un struct simple con los siguientes campos:

name Nombre del tileset.

firstGridID Primera ID general del tileset. Si hemos añadido un tileset con n tiles, la firstGridID del siguiente que añadamos será *n+1.*

textureID ID de la imagen con el conjunto de tiles.

tileWidth, tileHeight Ancho de cada tile en píxeles.

spacing Espacio entre las tiles de la imagen.

margin Margen de la imagen. Debe ser el mismo por los cuatro laterales.

width, height Ancho y alto del tileset medido en tiles, o número de columnas y filas.

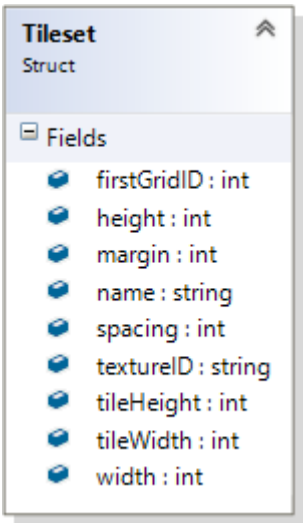


Figura 8.15 – Estructura Tileset.

Se añadirá a la clase principal Game un contenedor de tilesets. Podría haberse añadido como un atributo de Level, pero nos interesa que los tilesets sean comunes a todos los niveles.

Hemos considerado el mapa de tiles y los objetos del nivel, pero no hemos dicho nada de la imagen de fondo, en adelante *background*. Los backgrounds son imágenes que actuarán como fondo del escenario, y se dibujarán debajo (es decir, antes) del resto de elementos del nivel, exceptuando otros backgrounds a los que les asignemos mayor profundidad. Los objetos no interactúan con ellos y su función principal es la de mejorar la apariencia del nivel, dando la impresión de que el mundo del juego es mayor o más rico de lo que parecería si sólo vemos las tiles y un fondo grisáceo estático. Se implementarán tres tipos de background:

Static background Aunque se mueva la cámara, el background seguirá mostrándose igual. Lo normal es que la imagen del background sea del mismo tamaño que la ventana del juego, pero no es obligatorio (en caso de no serlo, se verá sólo la parte que quepa en pantalla).

Parallax background Se moverán a menor velocidad que la cámara, dando una sensación de profundidad. La velocidad de movimiento se calculará según el tamaño de la imagen y el de la ventana del juego.

Moving background Se moverá a una velocidad fija, independientemente de que el jugador (y por tanto la cámara) se mueva. La imagen se irá reproduciendo de forma cíclica, por lo debe usarse una imagen preparada para este efecto, es decir, cuyo extremo derecho ligue bien con el extremo izquierdo.

Los niveles podrán tener también capas de background, que serán realmente capas de objetos, ya que en el framework trataremos los background como objetos del juego. Tendremos una clase base Background (derivada de GameObject) y una clase derivada por cada uno de los tipos de background que hemos comentado: MovingBck, ParallaxBck y StatickBck.

Aunque los background podrían incluirse en una ObjectLayer, se ha creado el tipo derivado BckLayer por razones de eficiencia, ya que con cada Background tendremos que comprobar si es de tipo parallax para calcular la velocidad a la que se mueve. Esta comprobación no cuesta «nada» hacerla en una BckLayer con uno, dos o tres backgrounds, pero hacerla en la ObjectLayer para todos los objetos del juego podría suponer una carga.

En la Figura 8.16 vemos la relación entre las nuevas clases introducidas para la gestión de backgrounds y las ya existentes.

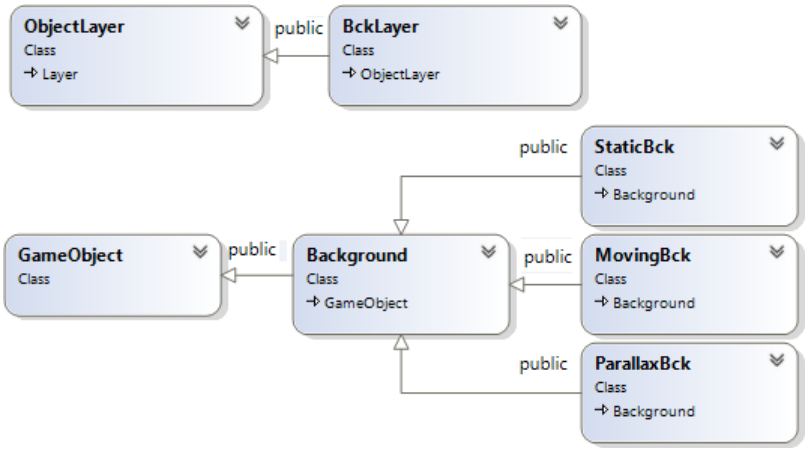


Figura 8.16 – Clases para la gestión de backgrounds.

En relación al código, lo más interesante de lo comentado en esta sección se encuentra en los métodos update y render de las clases ObjectLayer y TileLayer, los cuales se explicarán en las secciones destinadas a la actualización y el dibujado del bucle principal (8.10 y 8.11).

8.6. Data-Driven Design

Para evitar tener que estar cambiando el código que se compila cada vez que queramos probar un nuevo juego, lo que haremos será *parsear* ficheros de datos que contendrán la configuración, los niveles y los objetos de nuestro juego. De esta forma podremos mantener las estructuras del framework genéricas, pues pueden ser completamente diferentes sólo cargando un archivo de

datos distinto. Mantener las clases genéricas y cargar datos externos para determinar su estado es llamado *Data-Driven Design*.

Se ha elegido el formato XML para los ficheros de datos, porque es fácil de parsear, existen bibliotecas en C++ muy ligeras para la tarea y el software de construcción de mapas Tiled, que se utilizó en etapas tempranas del proyecto, usa dicho formato. Para tratar con los ficheros XML se utilizará la biblioteca de código abierto TinyXML2, escrita por Lee Thomason y disponible bajo la licencia zlib.

Además de la posibilidad de extraer datos de ficheros externos, necesitamos poder crear los objetos del juego dinámicamente. Para ello utilizaremos el patrón de diseño *Factory Method*[8]. Una factoría de objetos es una clase encargada de la creación de objetos; en esencia, le decimos a la factoría qué objeto queremos crear, la factoría crea una instancia de ese objeto y la devuelve.

La factoría contendrá un contenedor asociativo de pares *clave-valor*, siendo la clave una ristra con el tipo del objeto (ID) y el valor del «Creador» del objeto, una pequeña clase cuyo único propósito es la creación de un objeto específico. Además necesitamos un método para registrar tipos y otro para crear los objetos de tipos previamente registrados. Podemos ver el diagrama de clases de la factoría en la Figura 8.17. Se compone de dos elementos principales:

- La clase GameObjectFactory, con los métodos registerType y create para registrar tipos y crear objetos de esos tipos, y el atributo m_creators para guardar una relación entre la ID de cada objeto y su Creator. La clase GameObjectFactory será un singleton.

- BaseCreator es una clase abstracta con el método virtual puro createGameObject. Nos tenemos que asegurar de que tenemos un Creator para cada una de los objetos del juego que vayamos a crear (de ahí los puntos suspensivos del diagrama), que en la práctica será, si tenemos un objeto de tipo «T», crear una clase TCreator con un método createGameObject que cree una instancia de de tipo «T» y devuelva un puntero a la misma.

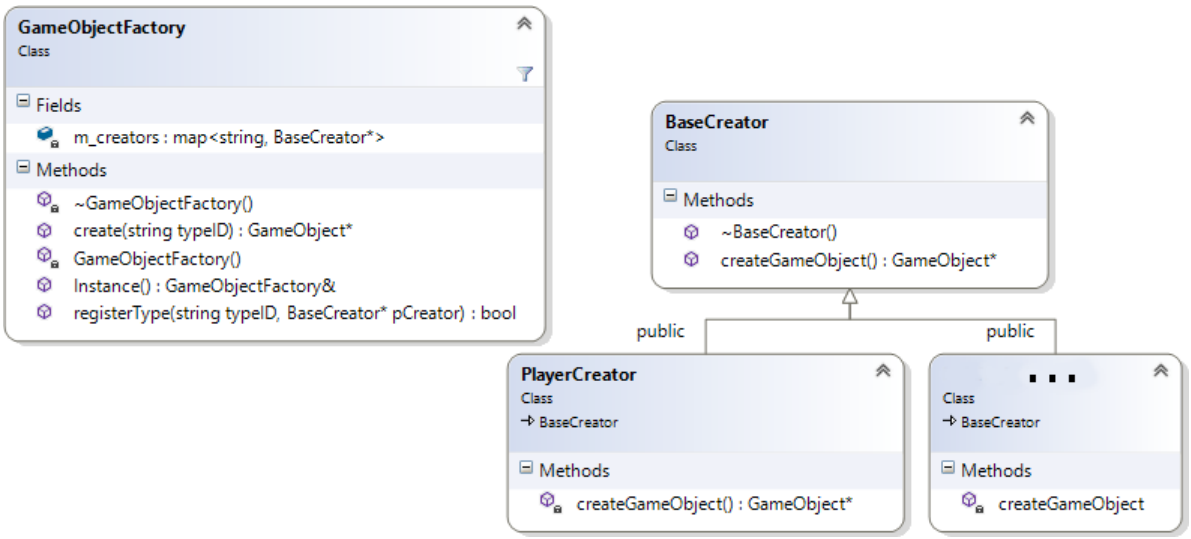


Figura 8.17 – Diagrama de la factoría de objetos.

[8]En la bibliografía consultada también se ha encontrado como *Distributed Factory*.

De esta manera podremos registrar los tipos que vayamos a usar, para a continuación recorrer el fichero de datos XML extrayendo los objetos que componen el juego y pasándole su ID a GameObjectFactory::create, que buscará en su mapa m_creators y, si encuentra el tipo, llamará a BaseCreator::createGameObject, que al ser virtual llamará al createGameObject de la clase derivada correspondiente y devolverá un puntero *GameObject a la instancia creada.

Implementación de la factoría

A continuación se muestra el código del método registerType:

```
bool GameObjectFactory::registerType(std::string typeID, BaseCreator*
    pCreator)
{
  auto it = m_creators.find(typeID);
  if (it != m_creators.end())
  {
    delete pCreator;
    return false;
  }

  m_creators[typeID] = pCreator;
  return true;
}
```

La función toma la ID que queremos asociar al nuevo tipo de objeto que vamos a registar (como una string), y el Creator de la clase. La función intenta encontrar el tipo usando la función std::map::find. Si se encuentra quiere decir que el tipo ya está registrado, por lo que se borra el puntero pasado y se devuelve false. Si el tipo aún no se ha registrado, se asigna su Creator al mapa con la clave pasada y se devuelve true. Como puede verse, es una función sencilla, cuyo único propósito es añadir nuevos tipos al mapa.

Veamos ahora el código del método create, encargado de crear un GameObject del tipo pasado (como string) y devolver un puntero al mismo:

```
GameObject* GameObjectFactory::create(std::string typeID)
{
  auto it = m_creators.find(typeID);
  if (it == m_creators.end())
  {
    std::cout << "could not find type: " << typeID << "\n";
    return nullptr;
  }

  BaseCreator* pCreator = (*it).second;
  return pCreator->createGameObject();
}
```

Lo primero es comprobar si el tipo pasado existe. Si no se encuentra se genera un mensaje de error y se devuelve nullptr; si por el contrario todo va bien y se encuentra, se utiliza el metodo createGameObject de la clase Creator del tipo en cuestión, que creará una nueva instancia y devolverá un puntero a ella. Para aclararlo, veamos la implementación del Creator de uno de los GameObject, por ejemplo de Player:

```
class PlayerCreator : public BaseCreator
{
  GameObject* createGameObject() const
  {
    return new Player();
  }
};
```

La clase, a la que se ha llamado PlayerCreator siguiendo el convenio adoptado para los nombres, no tiene mucha complicación. Hereda de BaseCreator y tiene un único método que devuelve un puntero a una nueva instancia de Player.

8.6.1. Ficheros XML

Por cada juego tendremos un fichero XML global donde indicaremos los niveles de que se compone, los recursos (imágenes, música y efectos de sonido) que se utilizarán y otra información. Además habrá un fichero XML adicional por cada nivel, con información específica de ese nivel.

Fichero XML global

En el fichero global del juego se encontrará la siguiente información:

- **Resolución** de la ventana donde se ejecutará.
- **Valores iniciales y máximos** de salud, vidas, puntuación y llaves.
- **Niveles** que componen el juego.
- **Imágenes, música y efectos de sonido** que se cargarán.
- **Iconos del HUD**: salud, vidas, puntuación y llaves.
- Imágenes de las **pantallas estáticas**: inicio, pausado, game over y conclusión del juego.
- Información sobre los **tilesets** que se usarán.

Por defecto el framework asume que este fichero tiene el nombre game.s2p y se encuentra en el mismo directorio que el ejecutable del juego; si se quiere cambiar, puede usarse el método Game::setGameFile(string file). Veamos un ejemplo de fichero general del juego para explicar su estructura:

```
<game>
    <window width="640" height="480"/>
    <initvalues health="3" lives="3" score="0" keys="0"/>
    <maxvalues health="5" lives="99" score="100" keys="99"/>
    <levels>
        <level name="Level 1" file="levels/Level 1.xml"/>
        <level name="Level 2" file="levels/Level 2.xml"/>
    </levels>
    <textures>
        <texture id="1up" source="assets/1up.png"/>
        <texture id="coinGold" source="assets/coinGold.png"/>
    ...
    </textures>
```

```
    <sounds>
        <sound id="jump" source="assets/sfx_movement_jump.wav"/>
        <sound id="lifeUp" source="assets/sfx_sounds_powerup.wav"/>
    ...
    </sounds>
    <music>
        <msc id="announce" source="assets/DST_Announce.ogg"/>
        <msc id="away" source="assets/DST-Away.ogg"/>
    </music>
    <hud health="healthBan" lives="1up" score="coinBronze" keys="keyBronze"/>
    <screens start="screenStart" pause="screenPause" gameover="screenGameOver" end="sreenEnd"/>
    <tilesets>
        <tileset name="Land" firstgid="1" tilewidth="32" tileheight="32" spacing="0" margin="0"
            textureID="tilesetLand" width="13" height="11"/>
        <tileset name="Clouds" firstgid="144" tilewidth="32" tileheight="32" spacing="0" margin
            ="0" textureID="tilesetClouds" width="4" height="7"/>
    </tilesets>
</game>
```

Se irán explicando uno a uno los elementos y sus atributos:

window Resolución de la ventana. Los atributos width y height nos darán el ancho y el alto.

initvalues Los atributos health, lives, score y keys indican los valores mínimos de salud, vidas, puntuación y llaves.

maxvalues Los atributos health, lives, score y keys indican los valores máximos de salud, vidas, puntuación y llaves.

levels De aquí se extraen los niveles que componen el juego. El orden de los hijos level determinará el orden en que se suceden los niveles en el juego.

- **level** El atributo name es el nombre o ID del nivel, y file es la ruta del archivo.

textures Imágenes que se cargarán como texturas al iniciar el juego. Se cargarán en un gestor de texturas que se explicará en la Sección 8.11.

- **texture** El atributo id es el nombre o ID que le daremos a la textura, y source la ruta del archivo.

sounds Efectos de sonido que se cargarán al iniciar el juego.

- **sound** El atributo id es el nombre del efecto de sonido, y source la ruta del archivo.

music Pistas de música que se cargarán al iniciar el juego.

- **msc** El atributo id es el nombre de la pista de música y source la ruta del archivo.

hud Iconos del HUD. Los atributos health, lives, score y keys hacen referencia a la ID de las texturas que representarán la salud, las vidas, la puntuación y las llaves.

screens Pantallas estáticas del juego. Los atributos start, pause, gameover y end son la ID de las texturas para las pantallas de inicio, pausado, game over y final.

tilesets Tilesets que se crearán al iniciar el juego. Cada hijo tileset contendrá información sobre la imagen que contiene las tiles.

tileset Cada uno de los atributos se corresponde con los campos del struct Tileset, explicado en la Sección 8.5.

Con esto tenemos todos los recursos que se van a cargar y toda la información general del juego, es decir, común a todos sus niveles.

Ficheros XML de niveles

Como se comentó antes, cada nivel contará con un archivo independiente con información sobre ese nivel, en concreto la siguiente:

- **Escenario** del nivel: tamaño, tiles que lo componen y background.
- **Objetos** del nivel.
- **Música** que sonará de fondo.

Para comentar la forma en que está estructurada esta información mostraremos un archivo de ejemplo de un nivel:

```
<map width="20" height="15" tilewidth="32" tileheight="32" musicID="announce">
    <background name="Background Layer">
        <object type="StaticBck" x="0" y="0" textureID="bckBlue" ...>
            <animations>
                <animation name="idle" row="0" nFrames="1" frameTime="0"/>
            </animations>
        </object>
    </background>
    <layer name="Transparent layer" width="20" height="15">
        <data>
            <tile gid="0"/>
      <tile gid="80"/>
            <tile gid="42"/>
             ...
        </data>
    </layer>
    <layer name="Solid layer" width="20" height="15">
        <properties>
            <property name="collidable" value="true"/>
        </properties>
        <data>
      ...
        </data>
    </layer>
    <objectgroup name="Object Layer">
        <object type="Player" x="78" y="242" textureID="monkeySheet" ...>
            <animations>
                <animation name="death" row="3" nFrames="6" frameTime="250"/>
                <animation name="idle" row="0" nFrames="1" frameTime="0"/>
                <animation name="jump" row="2" nFrames="1" frameTime="0"/>
                <animation name="walk" row="1" nFrames="4" frameTime="90"/>
            </animations>
        </object>
        <object type="LeftRight" x="320" y="129" textureID="enemSnail" ...>
            <animations>
                <animation name="idle" row="0" nFrames="1" frameTime="0"/>
                <animation name="walk" row="0" nFrames="2" frameTime="200"/>
            </animations>
        </object>
    </objectgroup>
</map>
```

El archivo XML de un nivel, al igual que los niveles en sí (ver epígrafe 8.5), estará compuesta por cada una de sus capas. El orden en que aparezcan en el fichero será en el que sean dibujadas, significando que la última será la que se dibuje encima de las demás. El orden más lógico por tanto sería: capa de background, capa de tiles no colisionables o transparentes, capa de tiles colisionables o sólidas, y capa de objetos. Puede haber más de una capa de cada tipo, aunque con estas cuatro debería ser suficiente; de hecho, realmente ya hay dos capas de objetos (la de objetos en sí y la de background) y dos capas de tiles (la transparente y la sólida). A continuación se comentarán los elementos y atributos que nos dan esta información:

map Los atributos width y height nos dan el ancho y alto del nivel en tiles. tilewidth y tileheight nos informan del tamaño de las tiles y musicID es la pieza musical que sonará de fondo, que habrá tenido que cargarse previamente desde el fichero global.

background Capa de background con un único atributo llamado name (nombre de la capa). Cada hijo object será un background. El elemento object se explica más adelante.

layer Capa de tiles con los atributos name (nombre de la capa), width (ancho en tiles) y height (alto en tiles). El elemento layer puede tener dos hijos:

- **data** Nos informa de las tiles que componen la capa. Cada tile será un elemento tile, hijo de data, con un atributo gid que hace referencia a la id global de la tile. Si una capa de tiles tiene un ancho de 20 y un alto de 15, como las que se muestran en este ejemplo, tendrá un total de 300 tiles y, por tanto, el archivo tendrá 300 elementos tile.
- **properties** Elemento opcional para propiedades especiales. En el ejemplo se utiliza para añadir una propiedad (elemento property) que nos indica si la capa está compuesta por tiles colisionables o no.

objectgroup Capa de objetos, con un único atributo que nos indica su nombre. La capa estará compuesta de todos los objetos (excepto los background) que tenga el nivel.

- **object** Contiene la información para crear un objeto del juego. La mayoría de atributos del nodo se corresponden con los atributos que nos harán falta para inicializar el objeto. Además cada objeto tendrá como hijo el nodo animations, que a su vez tendrá tantos hijos animation como animaciones tenga el objeto.
  - **animation** Sus cuatro atributos nos dan información sobre una animación del objeto:

name	Nombre de la animación.
row	Fila de la hoja de sprites con los frames de la animación.
nFrames	Número de frames de la animación.
frameTime	Tiempo que se tarda en pasar de un frame al siguiente, en milisegundos.

Con esta información más la del fichero global, ya tenemos todo lo necesario para que el juego se ejecute. Aunque utilicemos la biblioteca TinyXML2 para leer los archivos XML, sus nodos y sus atributos, igualmente necesitaremos construir un parser para extraer toda la información útil de los archivos y pasársela a las funciones correspondientes del framework.

8.6.2. Parser

Se creará una clase Parser con dos métodos públicos: uno para analizar sintácticamente (o *parsear*) el fichero XML global y otro para parsear el fichero XML de un nivel. Para cumplir su propósito, estos métodos se valdrán de una cantidad considerable de métodos privados que analizarán nodos XML concretos. Los dos métodos públicos son parseGameFile y parseLevel.

parseGameFile Recibe como parámetro el archivo XML global del juego, extrae la información que contiene y modifica los atributos de Game en consecuencia. Llama a los siguientes métodos privados:

- **parseLevels** Analiza el elemento levels.
- **parseTextures** Analiza el elemento textures.
- **parseSounds** Analiza los elementos sounds y music.
- **parseTilesets** Analiza el elemento tilesets.
- **parseValues** Analiza los elementos initvalues y maxvalues.
- **parseScreens** Analiza el elemento screens.
- **parseHUD** Analiza el elemento hud.

parseLevel Tiene como único parámetro un archivo XML de un nivel. Extrae la información del mismo y devuelve un puntero a una instancia de la clase Level. Recorrerá los elementos hijos del nodo raíz del archivo y, según el tipo de capa que sea, llamará a los siguientes métodos privados:

- **parseTileLayer** Analiza un elemento layer y, con la información extraída, se crea una capa de tiles (objeto de tipo TileLayer) y se añade al atributo m_layers del nivel.
- **parseObjLayer** Analiza un elemento objectgroup y añade al nivel una capa de objetos (de tipo ObjectLayer) con la información extraída. Para analizar cada objeto se hace uso del siguiente método privado:
  - **parseObject** Analiza un elemento object, crea un objeto según su contenido y devuelve un puntero a una instancia de tipo GameObject (o nullptr si no se ha podido crear).
- **parseBckLayer** Analiza un elemento background. Esta función opera de manera muy similar a parseObjLayer (al fin y al cabo los Background también derivan de GameObject), pero se ha creado para ahorrarnos algunas comprobaciones.

Como se ha comentado, se utilizará la biblioteca TinyXML2, de ahí que en la declaración de los métodos de la clase (Figura 8.18) aparezcan los tipos XMLElement o XMLError, clases propias de dicha biblioteca. XMLElement es un contenedor para un único elemento XML, incluyendo sus atributos; XMLError define los posibles códigos de error devueltos por muchas de las funciones de TinyXML2.

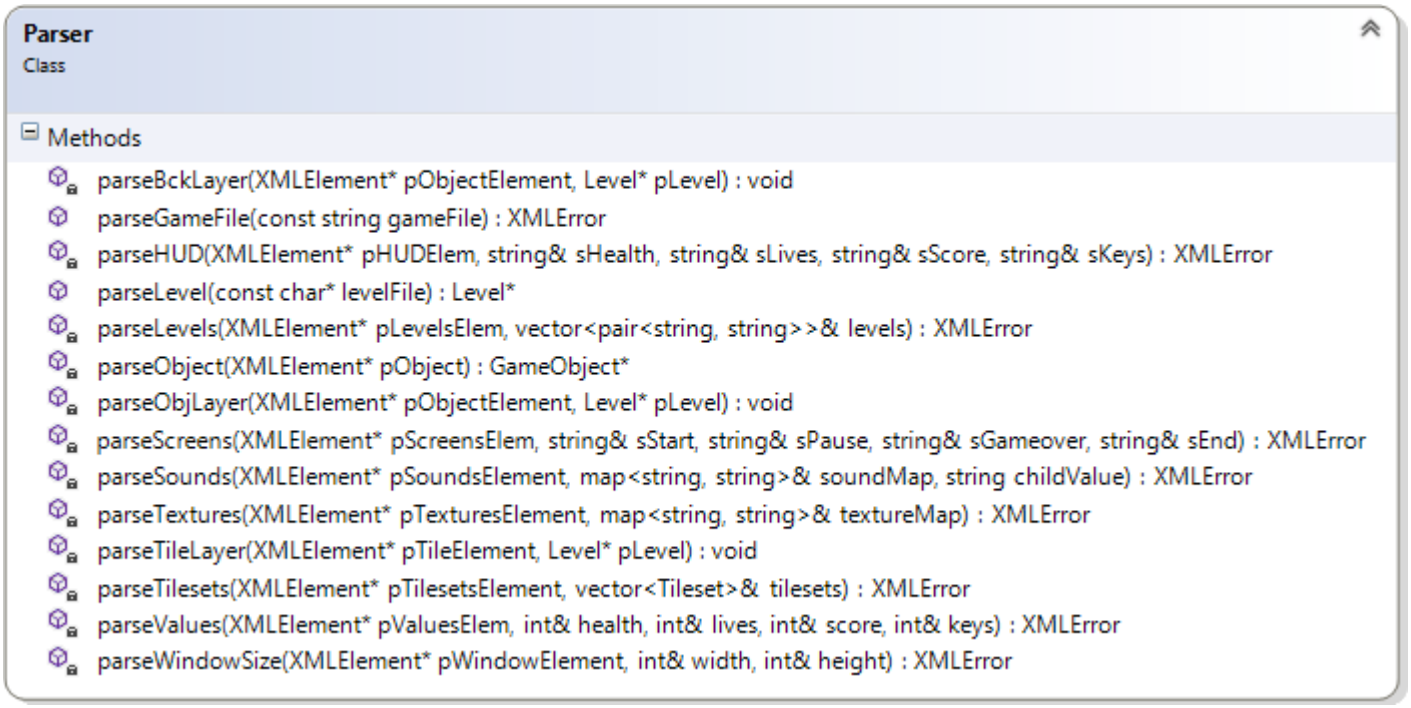


Figura 8.18 – Métodos de la clase Parser.

Analizando el fichero XML global

Comenzaremos explicando cómo se analiza el fichero XML global del juego, a través del código de la función parseGameFile:

```
1  XMLError Parser::parseGameFile(const std::string gameFile)
   {
     XMLDocument xmlDoc;
4    XMLError eResult = xmlDoc.LoadFile(gameFile.c_str());
     XMLCheckPrintResult(eResult);

7    XMLNode * pRoot = xmlDoc.FirstChild();
     if (pRoot == nullptr) XMLCheckPrintResult(XML_ERROR_FILE_READ_ERROR);

10   // Parse level files
     XMLElement *pElement = pRoot->FirstChildElement("levels");
     vector<pair<string, string>> levels;
13   eResult = parseLevels(pElement, levels);
     XMLCheckPrintResult(eResult);
     // Load level files
16   TheGame::Instance().setLevelFiles(levels);
       ...
   }
```

Lo primero es crear un nuevo documento, que se hace en la línea 3 utilizando la clase xmlDoc. A continuación cargamos el archivo gameFile pasado como parámetro (que debería ser el archivo global del juego) en el documento creado, mediante la función XMLDocument::LoadFile. Ésta devuelve un código de error que comprobamos con la macro XMLCheckPrintResult para comprobar que no ha habido ningún error; de haberlo se para la ejecución de parseGameFile y se devuelve el código de error.

Si no ha habido error, ya tenemos el fichero XML cargado en el xmlDoc. En la línea 7 creamos un puntero al nodo raíz del documento. Este nodo será el elemento game, y sus elementos hijo nos darán la información que nos hace falta para configurar el juego (ver epígrafe 8.6.1 para recordar la estructura del fichero XML global). Por tanto lo siguiente será ir buscando los elementos donde tenemos los datos que nos interesan y pasárselos a los métodos privados de Parser correspondientes. Como ejemplo, veremos cómo obtener los niveles de que se compone el juego:

1. Buscar el hijo del nodo raíz con el nombre que corresponda, en este caso levels. Es lo que se hace en la línea 11.

2. Llamar al debido método de Parser, que para extraer los niveles es parseLevels (fila 13). Este método tiene como parámetros un XMLElement y un vector de pares de string donde se almacenarán el nombre de los niveles que compondrán el juego y las rutas de los archivos XML que los definen.

3. Cargar en la clase central Game la nueva información, lo cual se hace en la fila 16 por medio del *setter* llamado setLevelFiles, que almacenará los niveles en el atributo m_levelFiles de Game.

El resto del método consiste en repetir este proceso para los otros elementos del archivo: textures, sounds, music, tilesets, initvalues, maxvalues, screens y hud. En el caso de textures, sounds y music, en el punto 3 no utilizaremos un *setter* de Game, sino que directamente cargaremos las imágenes o los sonidos en el gestor que corresponda (el gestor de sonidos se explica en la Sección 8.7 y el de imágenes en la Sección 8.11).

Se ha hecho mención a dos miembros de la clase Game, setLevelFiles y m_levelFiles, que no fueron nombrados en la Sección 8.2 donde se introdujo la clase. En dicha introducción se presentaron los métodos y atributos más importantes o que hacían entender mejor la estructura del framework, dejando fuera una gran cantidad de miembros que por su carácter auxiliar no aportaban mucho a la explicación, principalmente atributos que guardan información general del juego (niveles, valores iniciales y máximos, imágenes de las pantallas del juego, etc.) y los *setters* y *getters* para establecer y obtener el valor de estos atributos.

Analizando los ficheros XML de niveles

Para analizar un fichero XML con los datos de un nivel del juego, se utilizará el método Parser::parseLevel. El inicio del código de la función es el siguiente:

```
Level* LevelParser::parseLevel(const char* levelFile)
{
  XMLDocument xmlDoc;
  xmlDoc.LoadFile(levelFile) != XML_SUCCESS)
  {
    std::cout << "Error: unable to load " << levelFile << std::endl;
    return nullptr;
  }

  Level* pLevel = new Level();
```

Se crea una estructura XMLDocument donde cargamos el archivo del nivel que se quiere cargar. Si hay algún error, imprimimos un mensaje de error que informe del nivel que lo ha ocasionado; si no hay ninguno, creamos un nuevo objeto de tipo Level que rellenaremos con la información dada por el fichero XML. Lo siguiente es extraer dicha información:

```
XMLNode* pRoot = xmlDoc.RootElement();

pRoot->ToElement()->QueryAttribute("tilewidth", &m_tileSize);
pRoot->ToElement()->QueryAttribute("width", &m_width);
pRoot->ToElement()->QueryAttribute("height", &m_height);
pLevel->m_width = m_width;
pLevel->m_height = m_height;
pLevel->m_tileSize = m_tileSize;
...
```

Creamos un puntero al nodo raíz y obtenemos sus atributos (ver apartado 8.6.1). Para ello usamos la función QueryAttribute, a la que le pasamos el nombre de un atributo y la variable donde queremos que almacene su valor; en este fragmento de código estamos obteniendo los atributos width, height y tilewidth, que nos darán el ancho y alto del nivel en tiles, y el tamaño de dichas tiles. Una vez obtenidos los valores, los establecemos como los valores de los atributos del nivel que estamos creando (Parser se ha declarado como una friend class de Level, por eso se hace directamente sin el uso de métodos).

Una vez analizados los atributos del nodo raíz del documento, pasamos a analizar cada uno de sus hijos, que se corresponden con cada una de las capas del nivel:

```
for (XMLElement* e = pRoot->FirstChildElement(); e != NULL; e = e->
    NextSiblingElement())
{
  if (e->Value() == std::string("objectgroup"))
  {
    parseObjLayer(e, pLevel);
  }
  else if (e->Value() == std::string("layer"))
  {
    parseTileLayer(e, pLevel);
  }
  else if (e->Value() == std::string("background"))
  {
    parseBckLayer(e, pLevel);
  }
}

  return pLevel;
}
```

El bucle for recorre todos los hijos del nodo raíz pRoot y, para cada uno, comprueba si representa una capa de objetos (objectgroup), de tiles (layer) o de background (background), llamando al método privado que corresponda en cada caso: parseObjLayer, parseTileLayer o parseBckLayer respectivamente. Por tanto, al salir del for, el nuevo nivel que hemos creado tendrá su atributo m_layers (ver Figura 8.13) relleno con las capas contenidas en el fichero XML. En la última línea del método se devuelve el puntero al nuevo objeto Level creado.

Como se comentó antes, el método parseObjLayer hace uso de parseObject, que es la función que en última instancia crea un objeto a partir de un elemento XML. Esta parte quizá sea la más

interesante de la implementación del parser, ya que permite ver cómo se hace uso de la factoría para crear objetos dinámicamente a partir de información contenida en un fichero. Pasemos a explicar las partes más relevantes del código de parseObject:

Lo primero que se hará es declarar una serie de variables donde almacenaremos los valores extraídos de los atributos XML:

```
GameObject* LevelParser::parseObject(XMLElement* pObject)
{
  int x{ 0 }, y{ 0 }, value{ 0 }, value2{ 0 }, value3{ 0 }, frameWidth{ 0 },
    frameHeight{ 0 }, xSpeed{ 0 }, ySpeed{ 0 };
  std::string type, textureID, soundID, sound2ID;
  std::map<std::string, Animation> animations;
  SDL_Rect collider;
```

Para entender por qué se han declarado estas variables, observemos el siguiente fragmento de un archivo XML de un nivel, correspondiente a un objeto presente en la capa de objetos:

```
<object type="LeftRight" x="320" y="129" textureID="enemSnail" frameWidth="32
    " frameHeight="32" xBB="3" yBB="4" widthBB="27" heightBB="27" xSpeed="-1"
    ySpeed="5" value="1" value2="1" value3="0" soundID="DthImpact15" sound2ID=
    "dmgImp1">
     <animations>
         <animation name="idle" row="0" nFrames="1" frameTime="0"/>
         <animation name="walk" row="0" nFrames="2" frameTime="200"/>
     </animations>
</object>
```

Tenemos que extraer y almacenar todos los valores de los atributos XML, es decir, los valores numéricos o de texto que se encuentran entrecomillados; estos valores son los que en última instancia nos darán los nueve atributos principales de GameObject mostrados en la Figura 8.5. De las variables declaradas al comienzo de la función, algunas se corresponden directamente con los atributos XML, en concreto las variables de tipo int y string. Las otras dos, animations y collider, son estructuras que almacenarán varios de los atributos:

- collider es de tipo SDL_Rect, una estructura de SDL con cuatro campos enteros que definen un rectángulo: x, y, w y h. Lo usaremos para representar la caja de colisión del objeto, dada por los atributos xBB, yBB, widthBB y heightBB (se eligieron las letras «BB» al final del nombre de los atributos por *bounding box*).

- animations es un mapa asociativo de pares clave-valor donde guardaremos la información del elemento XML animations, dada por los atributos de cada uno de sus hijos animation. Como veremos en breve, se corresponderá directamente con el atributo m_animations de GameObject.

Continuando con el código de la función, para obtener los atributos numéricos usaremos la función QueryAttribute mentada previamente:

```
  pObject->QueryAttribute("x", &x);
  pObject->QueryAttribute("y", &y);
  pObject->QueryAttribute("frameWidth", &frameWidth);
  pObject->QueryAttribute("frameHeight", &frameHeight);
  pObject->QueryAttribute("xBB", &collider.x);
  pObject->QueryAttribute("yBB", &collider.y);
```

```
pObject->QueryAttribute("widthBB", &collider.w);
pObject->QueryAttribute("heightBB", &collider.h);
pObject->QueryAttribute("xSpeed", &xSpeed);
pObject->QueryAttribute("ySpeed", &ySpeed);
pObject->QueryAttribute("value", &value);
pObject->QueryAttribute("value2", &value2);
pObject->QueryAttribute("value3", &value3);
```

Es importante recordar que pObject es un puntero a un objeto de la clase XMLElement de la biblioteca TinyXML2, que representa en este caso a un elemento object del fichero XML de un nivel.

Con los atributos numéricos obtenidos, pasamos a los atributos de texto:

```
const char* check = pObject->Attribute("type");
if (check == nullptr) check = "";
type = check;
check = pObject->Attribute("textureID");
if (check == nullptr) check = "";
textureID = check;
check = pObject->Attribute("soundID");
if (check == nullptr) check = "";
soundID = check;
check = pObject->Attribute("sound2ID");
if (check == nullptr) check = "";
sound2ID = check;
```

Aquí no se ha hecho uso de QueryAttribute, puesto que está sobrecargada para los tipos primitivos, y en este caso se quiere extraer directamente el texto. Se utilizará en su lugar la función Attribute, que al contrario que QueryAttribute sólo tiene un parámetro de entrada y devuelve un const char* con el valor del atributo, o null si no existe. Es necesario declararse una variable char* auxiliar (a la que se ha llamado check), para realizar una comprobación adicional antes de asignar el valor a las variables de tipo string que hemos declarado, pues si el atributo no existiera y se intentara inicializar una string con null, se lanzaría un error.

Ya únicamente nos falta obtener las animaciones del objeto. Para ello tenemos que analizar el elemento XML animations:

```
XMLElement * pAnimations = pObject->FirstChildElement("animations");
for (XMLElement * pAnim = pAnimations->FirstChildElement(); pAnim != NULL;
    pAnim = pAnim->NextSiblingElement())
{
```

El puntero pAnimations apunta a una estructura XMLElement con la información del nodo animations. El bucle for tendrá tantas iteraciones como hijos de animations (o lo que es lo mismo, elementos animation) haya. En cada iteración se creará una nueva animación (clase Animation, acudir al apartado 8.3.1 para ver los detalles) y se añadirá al mapa asociativo animations declarado al inicio de la función; veamos paso a paso el código que realiza estas acciones:

```
    std::vector<SDL_Rect> frames;
    int row{ 0 };
    int nFrames{ 1 };
    int frameTime{ 0 };

    pAnim->QueryAttribute("row", &row);
    pAnim->QueryAttribute("nFrames", &nFrames);
```

```
pAnim->QueryAttribute("frameTime", &frameTime);
```

Se declaran una serie de variables para almacenar el resto de valores que definirán la animación actual. Si volvemos a mirar for veremos que pAnim irá tomando el valor de cada elemento animation, por lo que llamando a su método QueryAttribute obtendremos sus atributos y los guardaremos en las variables row, nFrames y frameTime.

En el caso de frames, es un vector de SDL_Rect que contendrá los frames de la animación, y que ya podemos deducir con los datos que tenemos:

```
for (int i = 0; i < nFrames; i++)
{
  SDL_Rect frame;
  frame.x = i*frameWidth;
  frame.y = row*frameHeight;
  frame.w = frameWidth;
  frame.h = frameHeight;
  frames.push_back(frame);
}
```

Sabiendo el número de frames, el ancho y el alto, podemos ir calculando las dimensiones de cada frame y añadiéndolo al vector. Ya únicamente nos resta extraer el nombre de la animación, crear la instancia de la clase Animation y añadirla a nuestro mapa animations:

```
  const char* name = pAnim->Attribute("name");
  if (name != nullptr)
  {
    Animation animation(textureID, row, frames, frameTime, collider);
    animations[name] = animation;
  }
}
```

Si el atributo name existe, creamos una variable Animation pasándole al constructor los argumentos correspondientes, y añadimos la animación al mapa asociativo con su nombre como clave. Por último se cierra la llave que finaliza el for para parsear las animaciones.

Por fin podemos pasar a la creación del objeto, fin último de la función parseObject que estamos desglosando. Para ello, lo primero es utilizar la factoría de objetos para crear un objeto del tipo que corresponda:

```
GameObject* pGameObject = TheGameObjectFactory::Instance().create(type);
```

En la variable type habíamos guardado el tipo del objeto, que pasado a la función create de GameObjectFactory nos devolverá un puntero a una nueva instancia de esa clase (o null si esa clase no ha sido registrada en la factoría). Si el objeto se ha creado correctamente, llamamos a la función load para darle los valores adecuados a sus atributos:

```
if (pGameObject != nullptr)
{
  pGameObject->load(unique_ptr<LoaderParams>(new LoaderParams(x, y, xSpeed,
      ySpeed, animations,
    value, soundID, sound2ID, value2, value3)));
}
else
{
```

```
        cout << "Warning: Parser couldn't create object of type " << type << endl
            ;
    }
    return pGameObject;
}
```

Por medio de la clase auxiliar LoaderParams, le pasamos a load los atributos principales del objeto, extraídos del fichero XML: la posición (x e y), la velocidad en los ejes x e y (xSpeed e ySpeed), las animaciones (animations), los valores numéricos especiales (value, value2 y value3) y la ID de los sonidos (soundID y sound2ID).

Si el objeto no se ha podido crear (pGameObject == nullptr), reportamos un error con información sobre el tipo que ha causado problemas. Por último, la función devuelve un puntero a la instancia recién de la clase heredera de GameObject creada (o nullptr).

Esto concluye la explicación de la función parseObject. El método parseObjLayer llamará a esta función tantas veces como elementos XML object haya en el fichero XML del nivel, añadiendo cada GameObject* devuelto al vector m_gameObjects de la capa de objetos del nivel (ver Figura 8.14).

8.7. Gestión de sonido

Para cargar y reproducir archivos de sonido, ya sean efectos o música, crearemos un gestor de sonido al que llamaremos SoundManager, que se implementará como un singleton. El gestor funcionará como intermediario entre el framework y la extensión SDL_mixer, que nos facilita la gestión de sonido de SDL (ver epígrafe 8.1.5). La clase se compondrá de los siguientes miembros:

m_sfxs Contenedor asociativo de pares *clave-valor* para almacenar los efectos de sonido que se vayan cargando. La clave de cada elemento será la ID del sonido, y el valor será un puntero a una estructura de tipo Mix_Chunk, la que utiliza SDL.

m_music Igual que el anterior, pero con pistas de música en lugar de sfxs. Por tanto el valor de cada par será de tipo Mix_Music*.

load Indicaremos el archivo de sonido a cargar, el tipo (efecto de sonido o música) y la ID con que se guardará en el mapa correspondiente.

playSound Pasaremos la ID del efecto de sonido que queremos reproducir, que debe haberse cargado previamente. Además tendrá como parámetro un entero que indica el número de veces que debe reproducirse.

playMusic Pasaremos la ID de la pista de música a reproducir (que debe haberse cargado antes) y un entero que indica cuántas veces repetirla.

stopMusic Se parará la pista de música que esté sonando.

SoundManager() En el constructor se inicializará la extensión SDL_mixer.

~SoundManager() En el destructor se liberarán todos los efectos de sonido y pistas de música que se hayan cargado.

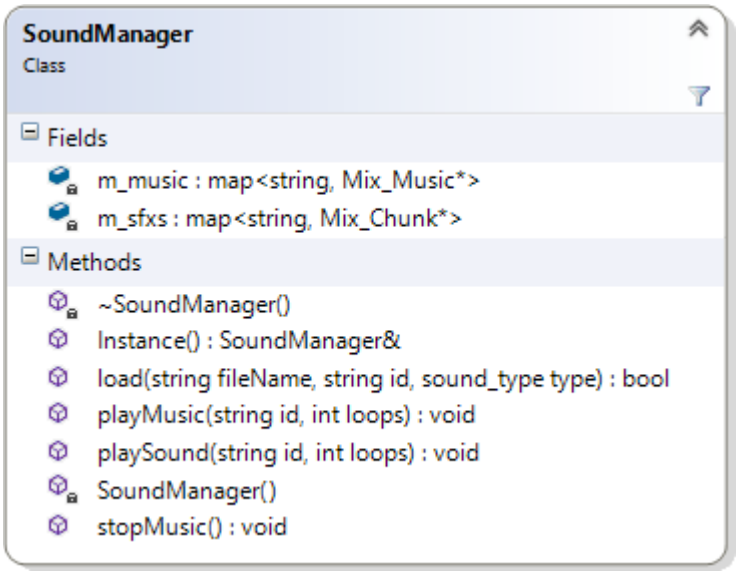


Figura 8.19 – Clase para gestionar el sonido

Implementación del gestor de sonido

El código del constructor de la clase es el siguiente:

```
SoundManager::SoundManager()
{
  if (Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT, 2, 1024) ==
      -1)
  {
    std::cout << "Mix Error: " << Mix_GetError() << std::endl;
  }
}
```

La función Mix_OpenAudio es la encargada de inicializar la extensión SDL_mixer, como se explicó en el apartado dedicado al audio de SDL (8.1.5). En el constructor realizaremos por tanto una llamada a esta función con los valores indicados en dicha subsección. Si la inicialización no se ha podido llevar a cabo (la función devuelve -1), informamos del error específico usando la función Mix_GetError de SDL_mixer.

Para cargar efectos de sonido y pistas de música utilizaremos la misma función, SoundManager::load. Según el valor de su argumento type sabremos si se quiere cargar uno u otro. type es de tipo sound_type, un enum con dos valores posibles: SOUND_MUSIC y SOUND_SFX. El código de la función comienza así:

```
bool SoundManager::load(string fileName, string id, sound_type type)
{
  if (type == SOUND_MUSIC)
  {
    Mix_Music* pMusic = Mix_LoadMUS(fileName.c_str());

    if (pMusic == 0)
    {
      std::cout << "Could not load music. Error: " << Mix_GetError() << std::
          endl;
      return false;
```

```
    }

    m_music[id] = pMusic;
    return true;
  }
```

De inicio se comprueba si queremos cargar una pista de música. De ser así, se utiliza la función Mix_LoadMUS de SDL_Mixer, que toma como parámetro un archivo de sonido (le pasamos el argumento fileName), crea una instancia de la clase Mix_Music y nos devuelve un puntero a ésta. Si pMusic apunta a un valor válido, lo añadimos al mapa m_music con la id pasada como parámetro y devolvemos true; si no, reportamos el error y devolvemos false. Veamos la segunda parte de la función, dedicada a los efectos de sonido:

```
  else if (type == SOUND_SFX)
  {
    Mix_Chunk* pChunk = Mix_LoadWAV(fileName.c_str());

    if (pChunk == 0)
    {
      std::cout << "Could not load SFX. Error: " << Mix_GetError() << std::
         endl;
      return false;
    }

    m_sfxs[id] = pChunk;
    return true;
  }
  return false;
}
```

Como puede verse, se hace lo mismo que con la música, pero se usa la función Mix_LoadWAV, que devuelve un Mix_Chunk* (en lugar de un Mix_music*). Además, debemos añadir el efecto de sonido al contenedor correspondiente, que es m_sfxs.

Antes de ver el código de métodos playMusic y playSound, hablaremos de las funciones Mix_PlayMusic y Mix_PlayChannel de SDL_Mixer, de las que hacen uso. Empecemos con la primera:

```
int Mix_PlayMusic(Mix_Music *music, int loops)
```

music Puntero al Mix_Music a reproducir.

loops Número de veces que se quiere reproducir la pista. -1 para reproducirla en bucle indefinidamente.

Por otro lado, la especificación de Mix_PlayChannel es:

```
int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops)
```

channel En qué canal reproducir el sonido, o -1 para utilizar el primer canal libre.

chunk Puntero al Mix_Chunk a reproducir.

loops Número de repeticiones, -1 para repeticiones infinitas. Si pasamos 1 la muestra se reproducirá dos veces (1 repetición).

Conociendo estas funciones y los parámetros de los métodos playMusic y playSound, su código apenas necesita explicación:

```
void SoundManager::playMusic(string id, int loops)
{
  Mix_PlayMusic(m_music[id], loops);
}

void SoundManager::playSound(string id, int loops)
{
  Mix_PlayChannel(-1, m_sfxs[id], loops);
}
```

Utilizamos la ID de la música o el efecto de sonido que se quiere reproducir para buscar el Mix_Music* o Mix_Chunk* en el mapa correspondiente, y lo pasamos a la función de SDL_Mixer correspondiente, especificando también el número de repeticiones.

El código del destructor de SoundManager se mostrará en la Sección 8.12, donde se explica cómo se sale del juego y se liberan los recursos.

8.8. Inicialización del juego

Aquí se explicará cómo se inicia el juego y se cargan los recursos, de lo cual se encarga el método init de la clase central Game. Los pasos en que se divide el método son los siguientes:

- Inicializar los módulos y extensiones de SDL que se van a utilizar. Para ello se ha creado el método privado initSDL.
- Si no ha habido problemas con el punto anterior, se hará uso del subsistema de vídeo de SDL para inicializar los atributos m_pWindow y m_pRenderer (ver Figura 8.3), que serán la ventana en que se ejecutará el juego y el renderer encargado de dibujar en ella.
- Concluidos los preliminares relacionados con SDL, pasaremos a la carga de recursos por medio de una función que llamaremos initResources.
- Si la carga de recursos ha ido bien, pasaremos a finalizar la función, inicializando algunos atributos de Game y devolviendo true para indicar que la inicialización se ha completado con éxito.

Se profundizará en estos puntos a continuación, apoyándonos en la solución implementada. En la Figura 8.20 se muestra un esquema con los pasos en que se divide el método.

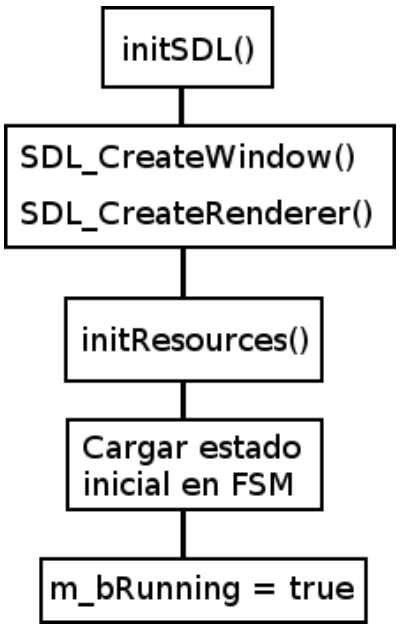


Figura 8.20 – Estructura del método Game::init.

Implementación del método de inicio

Antes de entrar en detalle en la definición del método Game::init, veamos su prototipo y el significado de sus parámetros:

```
bool Game::init(int xpos, int ypos, int width, int height, const char* title,
    bool fullscreen)
```

xpos, ypos Posición en la pantalla que ocupará la esquina superior izquierda de la ventana del juego.

width, height Ancho y alto de la ventana.

title Texto que se mostrará en el título de la ventana.

fullscreen Booleano para indicar si queremos o no que la ventana ocupe toda la pantalla.

La definición completa del método es la siguiente:

```
bool Game::init(int xpos, int ypos, int width, int height, const char* title,
    bool fullscreen)
{
  if (initSDL())
  {
    int flags = 0;
    if (fullscreen) flags = SDL_WINDOW_FULLSCREEN;
    m_pWindow = SDL_CreateWindow(title, xpos, ypos, width, height, flags);

    if (m_pWindow != nullptr)
    {
      m_pRenderer = SDL_CreateRenderer(m_pWindow, -1,
          SDL_RENDERER_PRESENTVSYNC);
```

```
        if (m_pRenderer != nullptr)
14      {
          if (initResources())
          {
17          m_gameWidth = width;
            m_gameHeight = height;
            m_pGameStateMachine->changeState(new StartState());
20          m_bRunning = true;
            return true;
          }
23      }
      }
    }
26  cout << "Couldn't initialize game. SDL error: " << SDL_GetError() << endl;
    return false;
}
```

Nada más comenzar (línea 2) se llama al método initSDL, que intentará inicializar SDL, SDL_-Image y SDL_ttf, y devolverá true si tiene éxito o false en caso contrario.

Si initSDL ha concluido con éxito, pasamos a inicializar m_pWindow, la ventana en que se ejecutará el juego. Como se explicó en el apartado 8.1.2, para ello se hace uso de la función SDL_CreateWindow, a la que se llama en la línea 7. Los argumentos utilizados son exactamente los mismos que recibe init, a excepción de la nueva variable flag, que tomará el valor de la constante SDL_WINDOW_FULLSCREEN si fullscreen es verdadero o cero en caso contrario.

Si la ventana SDL se ha creado correctamente (m_pWindow != null) procedemos a inicializar m_pRenderer, el renderer que dibujará en la ventana del juego, puesto que queremos hacer uso de la API de aceleración por hardware de SDL 2. Se llama a la función SDL_CreateRenderer (línea 11) pasándole los siguientes argumentos:

m_pWindow Ventana en la que se mostrará lo que se dibuje.

-1 Para indicar que se inicialice el primer driver de renderizado que admita las flags pasadas, sin especificar ninguno en concreto.

SDL_PRESENTVSYNC Para sincronizar la tasa de actualizado del renderer con la de la pantalla. No es necesario pasar también la flag SDL_RENDERER_ACCELERATED ya que por defecto se dará prioridad a esta opción.

Si se han inicializado con éxito SDL, m_pWindow y m_pRenderer, ya estamos en disposición de cargar los recursos que necesitará el juego mediante el método privado initResources (línea 15). Puesto que esta función se compone de una secuencia de llamadas a funciones ya tratadas previamente, se ha optado por explicarla por medio de comentarios incluidos en el propio código:

```
bool Game::initResources()
{
  // Registrar en la factoría los objetos herederos de GameObject
  TheGameObjectFactory::Instance().registerType("Player", new PlayerCreator()
     );
  TheGameObjectFactory::Instance().registerType("ScoreItem", new
     ScoreItemCreator());
```

```
  TheGameObjectFactory::Instance().registerType("CheckPoint", new
      CheckPointCreator());
  // ... Resto de tipos ...

  // Inicializar la FSM, atributo de Game
  m_pGameStateMachine = new GameStateMachine();

  // Parsear el fichero XML global del juego. Esta es la parte central de la
      carga de recursos.
  EditorParser parser;
  if (parser.parseGameFile(m_gameFile) != tinyxml2::XML_SUCCESS)
  {
    return false;
  }
  return true;
}
```

Volviendo al método init, y suponiendo que la carga de recursos haya sido satisfactoria (initResources() == true), sólo quedan los últimos detalles para finalizar la función: se guarda el valor del ancho y el alto de la ventana en los atributos m_width y m_height (líneas 17 y 18), se carga en la FSM el estado inicial del juego que por defecto será StartState (línea 19), y se pone el atributo m_bRunning a true para indicar que el juego ya está corriendo (línea 20). Por último, devolvemos true para indicar que la inicialización del juego se ha llevado a cabo sin problemas.

8.9. Game loop: User input

La gestión de entrada es la primera de las tres funciones del game loop, que en el caso de nuestro framework toma la forma del método Game::handleEvents. No obstante, no haremos uso del subsistema de eventos de SDL directamente en la clase Game, sino que se creará un «gestor de eventos», una clase que gestionará la entrada de dispositivos (en concreto de teclado y ratón) a la que llamaremos InputHanlder, la cual será un singleton con los siguientes miembros:

m_keystates Estructura que guardará el estado del teclado.

m_mouseButtonStates Vector con el estado de los botones del ratón.

m_mousePosition Posición del ratón en la ventana.

update Método central del gestor de eventos. Revisará la cola de eventos mediante SDL_PollEvent y, según el tipo del evento, llamará al método correspondiente, que modificará el estado de los atributos m_keyStates, m_mouseButtonStates o m_mousePosition en consecuencia. Podría hacerse toda la gestión directamente en el método update, pero de esta forma quedará más limpio y legible. Estas pequeñas funciones para gestionar eventos concretos son onKeyUp, onKeyDown, onMouseButtonDown, onMouseButtonUp y onMouseMove.

getMouseButtonState Método para obtener el estado de los botones del ratón.

getMousePosition Método para obtener la posición del puntero del ratón.

isKeyDown Método para saber si una tecla concreta del teclado está siendo pulsada.

El método Game::handleEvents simplemente llamará a InputHandler::update, que actualizará el estado de sus atributos si se ha producido algún evento. En la Figura 8.22 puede verse un esquema del proceso.

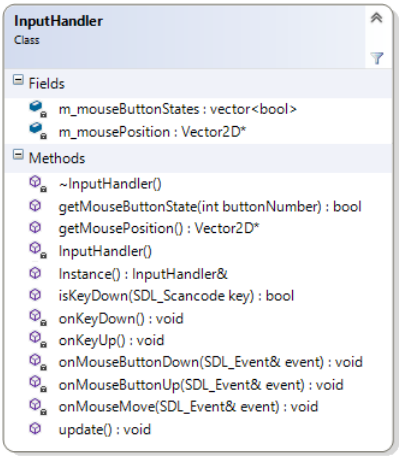


Figura 8.21 – Clase InputHandler.

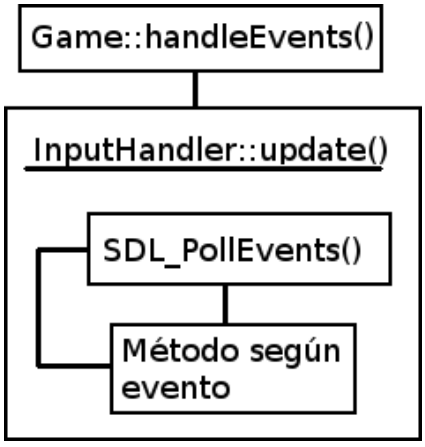


Figura 8.22 – Esquema de la gestión de eventos

Implementación del gestor de entrada

Veamos la implementación del método update, que como se ha dicho es el principal de InputHandler:

```
void InputHandler::update()
{
  SDL_Event event;

  while (SDL_PollEvent(&event))
  {
    switch (event.type)
    {
    case SDL_QUIT:
      TheGame::Instance().quit();
      break;
    case SDL_MOUSEBUTTONDOWN:
      onMouseButtonDown(event);
      break;
    case SDL_MOUSEBUTTONUP:
      onMouseButtonUp(event);
      break;
    case SDL_KEYDOWN:
      onKeyDown();
      break;
    case SDL_KEYUP:
      onKeyUp();
      break;
    default:
      break;
```

```
    }
  }
}
```

Como vemos, se hace uso de la estructura SDL_event y de la función SDL_PollEvent para capturar los eventos, tal y como se explicó en el apartado 8.1.3. Dependiendo del tipo de evento, dado por event.type, se llamará a un método u otro de InputHandler, exceptuando el caso SDL_QUIT (salida forzada del juego, por ejemplo si el usuario cierra la ventana o presiona ALT+F4) en el que se llama al método Game::quit.

Veamos algunos de los métodos de InputHandler, empezando por onMouseButtonDown. Antes de pasar a su definición, es conveniente saber que SDL utiliza la siguiente numeración para los botones del ratón: 0 para el izquierdo, 1 para el central (rueda) y 2 para el derecho. Utilizaremos esta misma numeración para los índices del vector m_mouseButtonStates, y para hacerlo más intuitivo se definirá el siguiente enum:

```
enum mouse_buttons
{
  LEFT = 0,
  MIDDLE = 1,
  RIGHT = 2
};
```

Sabiendo esto, pasemos a ver el código de onMouseButtonDown:

```
void InputHandler::onMouseButtonDown(SDL_Event& event)
{
  if (event.button.button == SDL_BUTTON_LEFT)
  {
    m_mouseButtonStates[LEFT] = true;
  }
  if (event.button.button == SDL_BUTTON_MIDDLE)
  {
    m_mouseButtonStates[MIDDLE] = true;
  }
  if (event.button.button == SDL_BUTTON_RIGHT)
  {
    m_mouseButtonStates[RIGHT] = true;
  }
}
```

Según el botón que se haya pulsado (dado por el campo event.button.button), modificamos el atributo m_mouseButtonStates, poniendo a true el botón correspondiente para indicar que se está presionando. El método onMouseButtonUp sólo se diferencia en que, en lugar de igualar a true, igualamos a false para indicar que el botón ya no está siendo presionado.

Con esto tendríamos actualizados los botones del ratón, pero falta actualizar la posición en pantalla del puntero, de lo que se encarga el método onMouseMove:

```
void InputHandler::onMouseMove(SDL_Event& event)
{
  m_mousePosition->x(event.motion.x);
  m_mousePosition->y(event.motion.y);
}
```

Se toman los valores de las posiciones x e y del puntero y se asignan a los campos correspondientes del atributo m_mousePosition. Con esto finaliza la actualización del ratón, por lo que pasaremos al teclado, en concreto al método que se llama cuando se detecta la pulsación de una tecla, llamado onKeyDown:

```
void InputHandler::onKeyDown()
{
  m_keystates = SDL_GetKeyboardState(NULL);
}
```

Como vemos, es un método sencillo con una única instrucción compuesta de una llamada a SDL_GetKeyboardState . Esta función de SDL devuelve un puntero a un *array* con el estado de las teclas (si pasamos un argumento de tipo int* distinto de NULL, lo rellena con la longitud del array devuelto). Un valor de 1 significa que la tecla ha sido pulsada y un valor de 0 indica que no. Los índices de este array se obtienen usando valores de tipo SDL_Scancode[9]. Para entenderlo mejor, veamos el método público isKeyDown, usado para conocer el estado de una tecla:

```
bool InputHandler::isKeyDown(SDL_Scancode key)
{
  if (m_keystates != 0)
  {
    return m_keystates[key];
  }
  else
  {
    return false;
  }
}
```

key será un SDL_Scancode con el valor correspodiente a la tecla de la que queremos saber el estado, por ejemplo SDL_SCANCODE_R para la letra R, SDL_SCANCODE_ESCAPE para la tecla ESCAPE, etc. El atributo m_keyStates, que se habrá actualizado en la última llamada a SDL_GetKeyboardState, apuntará al array con el estado de las teclas, por lo que el método isKeyDown devolverá verdadero (1) si la tecla está pulsada y falso (0) si no.

No se ha mencionado el método onKeyUp puesto que es idéntico a onKeyDown. Se han mantenido ambos por si en el futuro se quiere incluir alguna modificación que necesite diferenciarlos.

Disponemos de tres métodos públicos para conocer el estado de los periféricos: isKeyDown, getMouseButtonState y getMousePosition. El primero ya se ha comentado, por lo que pasaremos a los otros dos:

```
Vector2D* getMousePosition()
{
  return m_mousePosition;
}

bool getMouseButtonState(int buttonNumber)
{
  return m_mouseButtonStates[buttonNumber];
}
```

[9]https://wiki.libsdl.org/SDL_Scancode

Apenas necesitan explicación: getMousePosition devuelve el atributo m_mousePosition y getMouseButtonState devuelve el estado del botón que se consulte (LEFT, MIDDLE o RIGHT).

Con esto queda explicado el gestor de eventos. Como se dijo, el método Game::handleEvents se compone simplemente de una llamada al método InputHandler::update:

```
void Game::handleEvents()
{
  TheInputHandler::Instance().update();
}
```

Posteriormente, las clases que necesiten comprobar el estado de los dispositivos de entrada realizarán llamadas a los métodos correspondientes de InputHandler, normalmente en el método update de dicha clase. Por ejemplo, supongamos que en Game::update se quiere saber si se ha pulsado la tecla F, si se ha pulsado el botón izquierdo del ratón y la posición del puntero; una forma de hacerlo es la siguiente:

```
#include "InputHandler.h"
...
void Game::update()
{
  bool f_pressed = TheInputHandler::Instance().isKeyDown(SDL_SCANCODE_F));
  bool lmouse_pressed = TheInputHandler::Instance().getMouseButtonState(LEFT)
     );
  int mouseX = TheInputHandler::Instance().getMousePosition()->x();
  int mouseY = TheInputHandler::Instance().getMousePosition()->y();
}
```

8.10. Game loop: Update

Una vez procesados los eventos de entrada, es hora de actualizar el estado del juego, de lo que se encargará el método update de Game. Al igual que Game, gran parte de las clases que se han presentado tienen un método update que se encarga de actualizarlas. Llamando al update de Game, se iniciará una reacción en cadena de llamadas a los update de cada clase, que terminará con la actualización de los objetos del juego. La secuencia de llamadas (resumida en la Figura 8.23) será la siguiente:

- El update de Game llamará al update de GameStateMachine. Recordemos que la FSM es un atributo de Game.
- La FSM llamará al método update del estado actual. Para seguir con la secuencia de llamadas supondremos que el estado actual es PlayState, el más complejo y común.
- La clase PlayState tendrá como atributo el nivel que se está jugando, de tipo Level. Se llamará al update de Level.
- El nivel actualizará todas sus capas. En el caso de las capas de tiles no ocurrirá nada, pero en las capas de objetos se recorrerá el vector de objetos llamando al método update de cada GameObject.

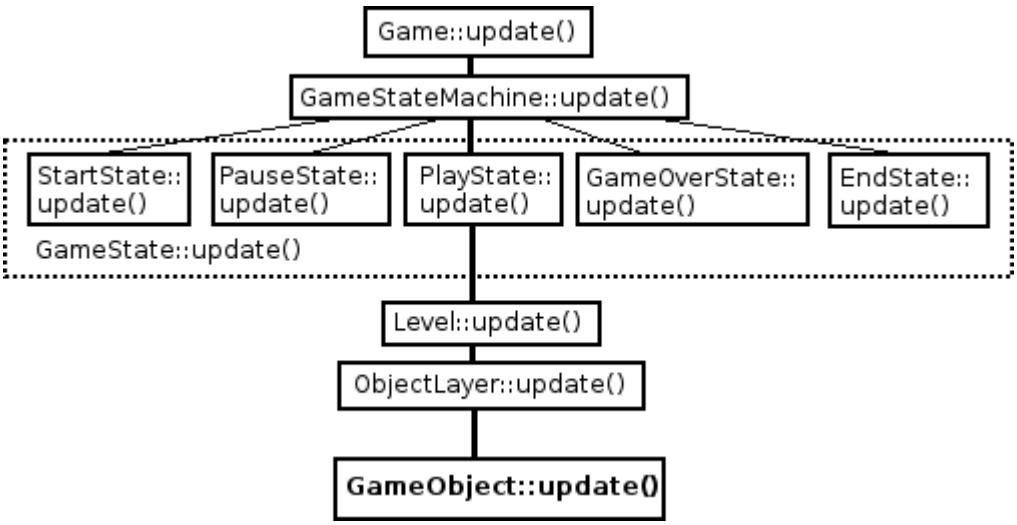


Figura 8.23 – Esquema de la secuencia de actualización del juego.

De esta forma, con una llamada al update de la clase central Game ya se actualizarán todos los objetos del juego, cada uno con su función update específica gracias al polimorfismo. Si en lugar de PlayState el estado actual fuera otro, sería más sencillo, ya que son pantallas estáticas y en el método update sólo tendríamos que comprobar si se ha pulsado la tecla correspondiente para cambiar de estado, sin actualizar ningún nivel ni, por tanto, ningún objeto del juego.

Implementación de la secuencia de actualización

Este proceso de actualización del juego es el más complejo del framework con diferencia, por lo que explicar el código de todos los métodos implicados no es la opción más sensata. En su lugar se mostrará el código de la susodicha secuencia de llamadas update, resumiendo en los casos en que sea más complejo. De esta forma podrá verse la cohesión entre las distintas clases del framework. El proceso de actualización empieza con una llamada al método Game::update:

```
void Game::update()
{
  m_pGameStateMachine->update();
}
```

Se realiza una llamada al método update del atributo m_pGameStateMachine, la FSM con los estados del juego. Veamos pues dicho método:

```
void GameStateMachine::update()
{
  if (!m_gameStates.empty())
  {
    m_gameStates.back()->update();
  }
}
```

Se comprueba si hay algún estado en la FSM y, de ser así, se llama al método update del estado actual (el último del vector m_gameStates). En este momento pueden darse dos opciones: que el estado actual sea «estático» (StartState, PauseState, GameOverState o EndState) o que se

trate del estado PlayState. Empezaremos por la primera opción por ser la más sencilla, y como ejemplo tomaremos el estado StartState:

```
void StartState::update()
{
  if (TheInputHandler::Instance().isKeyDown(SDL_SCANCODE_ESCAPE))
  {
    TheGame::Instance().quit();
  }
  if (!TheInputHandler::Instance().isKeyDown(SDL_SCANCODE_SPACE))
  {
    m_bPressed = false;
  }
  else if (TheInputHandler::Instance().isKeyDown(SDL_SCANCODE_SPACE) && !
     m_bPressed)
  {
    TheGame::Instance().getStateMachine()->changeState(new PlayState(0));
  }
  TheGame::Instance().getStateMachine()->dequeState();
}
```

En primer lugar se comprueba si se ha pulsado la tecla ESCAPE, en cuyo caso saldremos del juego. A continuación se comprueba si se ha pulsado SPACE y, de ser así, se realiza la transición al siguiente estado, que en este caso sería PlayState con el argumento 0 correspondiente al primer nivel, ya que estamos en el estado inicial.

La doble comprobación y el uso de la variable auxiliar m_bPressed tienen por objetivo evitar la propagación de la pulsación de una tecla de un estado a otro: por ejemplo, si estamos en el estado GameOverState y pulsamos SPACE, pasaremos a StartState, y no queremos que se propague la pulsación de SPACE, ya que de ser así pasaríamos directamente a PlayState sin ver la pantalla de inicio.

Por último llamamos al método dequeState de la FSM, pues se ha considerado una buena práctica para evitar problemas, aunque a priori este estado no debería darlos al no tener objetos que se actualicen independientemente.

El resto de estados estáticos funcionan de manera análoga, cambiando las teclas que se comprueban y el estado al que pasan según corresponda (ver Sección 8.4). Aquí terminaría la cadena de actualizaciones para uno de estos estados, pero no para PlayState, que es mucho más complejo:

```
void PlayState::update()
{
  if (TheGame::Instance().resurrected()){...}
  else if (TheGame::Instance().gameOver()){...}
  else if (TheGame::Instance().levelCompleted()){...}
  else
  {
    if (TheInputHandler::Instance().isKeyDown(SDL_SCANCODE_ESCAPE))
    {
      TheGame::Instance().quit();
    }
    if (!TheInputHandler::Instance().isKeyDown(SDL_SCANCODE_RETURN))
    {
      m_bPausePressed = false;
    }
```

```
    if (TheInputHandler::Instance().isKeyDown(SDL_SCANCODE_RETURN) && !
        m_bPausePressed)
    {
      m_bPausePressed = true;
      TheGame::Instance().getStateMachine()->pushState(new PauseState());
    }
    else
    {
      m_pLevel->update();
    }
  }
}
```

Al principio del método se realizan una serie de comprobaciones por medio de llamadas a métodos de Game[10] que nos dan información muy específica: resurrected nos señala si el jugador ha muerto con vidas restantes, gameover si ha muerto sin vidas, y levelComplete indica si el jugador ha llegado al objeto de fin de nivel del nivel actual. Si se cumplen estas condiciones habrá que llevar a cabo una serie de acciones, incluidas más comprobaciones (por ejemplo no es lo mismo completar un nivel cualquiera que el último del juego). Los bloques de código se han sustituido por «...» por simplificar la función; para verlos puede acudirse al código fuente, el cual cuenta con abundantes comentarios.

Después de las acciones del párrafo anterior, se comprueban la tecla ESCAPE (salir del juego) y ENTER (pasar al estado PauseState). Si ninguna se ha pulsado, se llama al método update del atributo PlayState::m_pLevel, el cual es de tipo Level* y apunta al nivel actual. El método Level::update se define como sigue:

```
void Level::update()
{
  for (int i = 0; i < m_layers.size(); i++)
  {
    m_layers[i]->update(this);
  }
}
```

Se recorren las capas del nivel y se llama al método update de cada una de ellas. Recordemos que las capas podían ser de tiles (TileLayer) o de objetos (ObjectLayer). Como ya se adelantó, en TileLayer::update no ocurrirá nada, aunque se mantiene el método por si se quiere añadir alguna funcionalidad (por ejemplo que el escenario se mueva). Por el contrario, ObjectLayer::update es un método importante donde, entre otras cosas, se lleva a cabo el actualizado de todos los objetos del juego:

```
void ObjectLayer::update(Level* pLevel)
{
  if (!m_gameObjects.empty())
  {
    for (auto it = m_gameObjects.begin(); it != m_gameObjects.end();)
    {
      if ((*it)->dead())
      {
        delete *it;
```

[10]Estos y otros métodos no se explicaron al presentar la clase Game debido a que su trascendencia en el documento es casi nula.

```
            it = m_gameObjects.erase(it);
11        }
          else
          {
14          ...
            (*it)->update();
            ++it;
17        }
        }
      }
20  }
```

Comenzamos comprobando si hay algún objeto en la capa, es decir, si m_gameObjects (ver Figura 8.14) no está vacío. Si hay objetos, iteraremos a través de ellos recorriendo el vector m_gameObjects de principio a fin (línea 5).

Para cada objeto, comprobaremos si ha «muerto» (línea 7) y, de ser así, lo borramos de memoria y del vector m_gameObjects. Un objeto muerto no es únicamente un enemigo al que hemos eliminado, puede ser un ítem que hemos recogido, una puerta que hemos abierto, etc. Si por el contrario el objeto está «vivo» o activo, se realizarán una serie de comprobaciones para saber si hay que actualizarlo (ya que no actualizaremos los objetos a la derecha de la cámara hasta que aparezcan por primera vez) y, de ser así, lo actualizaremos (línea 15) y avanzaremos el iterador una posición para evaluar el siguiente objeto.

Con esta llamada a GameObject::update concluye la secuencia de llamadas mostrada en la Figura 8.23. GameObject::update es un método virtual puro y, por tanto, debe implementarse para cada una de las clases derivadas de GameObject. La implementación variará mucho de una clase a otra, tanto como difieran las clases entre sí: en los ítems apenas habrá que hacer nada, mientras que en el jugador habrá que implementar el movimiento, las colisiones, el control, etc.

El usuario que vaya a utilizar el framework podrá implementar el método update de cada clase de manera que el comportamiento del juego se adecue a sus necesidades y expectativas. Al igual que con el resto de elementos del framework, todas las clases derivadas de GameObject tienen un comportamiento por defecto para el que se ha tenido que definir su método update, pero por razones de tiempo y espacio no se tratarán aquí. Si el lector está interesado en mirar el código de alguno, se recomienda el de la clase Player, con mucha diferencia el más interesante y complicado.

A continuación se comentará el enfoque (sin entrar a comentar el código) que se ha seguido con dos tareas que se realizan durante la etapa de actualización y que son especialmente importantes por su complejidad y la influencia que tienen en el resultado final: el movimiento y la detección de colisiones.

8.10.1. Movimiento

Para simular el movimiento de un objeto se realizarán dos acciones: modificar su posición según su velocidad, y cambiar (si fuera necesario) el sprite que se muestra. Para ello se seguirán los siguientes pasos en el método update de cada clase derivada de GameObject:

- Calcular el vector de velocidad del objeto. En base a las velocidades iniciales (m_xSpeed y m_ySpeed) y al comportamiento específico del tipo objeto, se calculará el vector de

velocidad actual, necesario para calcular la nueva posición del objeto. Por ejemplo, el enemigo LeftRight invertirá la componente x del vector de velocidad al chocar con un obstáculo o encontrar un precipicio, el jugador cambiará la componente y al saltar, etc.

- Se calculará la nueva posición, sumándole a la posición previa el vector de velocidad, y se comprobará si es una posición válida. Para saber si la nueva posición candidata es válida, habrá que llevar a cabo detección de colisiones, que se explica en el siguiente epígrafe.
- Calcular el sprite a mostrar. Para ello se calculará la animación actual y el frame de dicha animación que correspondería mostrar según el tiempo transcurrido y el frameTime, tiempo de espera entre frames de la animación.

La complejidad del cálculo del movimiento variará según el tipo de GameObject. En casos como los Item, que sólo tienen una animación y no se mueven, sólo hay que calcular qué frame mostrar. Lo opuesto ocurre con Player, donde antes de calcular el movimiento y la animación, habrá que comprobar la entrada de usuario para modificar la velocidad en consecuencia.

Estas modificaciones actualizan la posición y el sprite de la estructura GameObject, pero la sensación de movimiento no se completará hasta el último paso del game loop, cuando el objeto sea dibujado en pantalla.

8.10.2. Colisiones

Aunque hayamos calculado la nueva posición de un objeto en base a su velocidad, no siempre será posible moverlo ahí. Antes de modificar su posición definitivamente, tenemos que comprobar si habrá conflicto con los elementos sólidos del nivel, es decir, controlar las colisiones. La inmensa mayoría de las colisiones en los juegos que crearemos se producen con el escenario, en concreto con las tiles sólidas de él, pero también tendremos que tener en cuenta colisiones con algunos objetos sólidos, como las puertas cerradas.

Añadiremos a la clase GameObject dos funciones, que se llamarán checkCollideTile y checkCollideObject, para calcular colisiones con tiles y objetos. Las llamadas a estas funciones se realizarán en el método de actualización de cada objeto, después de calcular su nueva posición candidata.

Además de lo comentado en los párrafos previos, con la clase Player hay que gestionar colisiones especiales que no modifican el movimiento, como cuando recogemos un ítem, o que lo modifican de una manera especial, como cuando saltamos sobre un enemigo y rebotamos. Para controlar estas colisiones, que son siempre con objetos (nunca con tiles), haremos algunas modificaciones en el framework:

- Añadir un nuevo método virtual a la clase base GameObject, llamado collision.
- Añadir una clase para gestionar las colisiones, llamada CollisionManager. El gestor tendrá un método ckeckCollision que comprobará si hay colisión entre el jugador y un conjunto de objetos del juego, llamando al método collision de cada objeto con el que ocurra.
- Añadir a la clase ObjectLayer un atributo de tipo CollisionManager. Ahora en el método update de ObjectLayer, antes del bucle que recorre todos los objetos actualizándolos, se llamará a checkCollision para detectar colisiones entre Player y el resto de objetos de la capa (contenidos en el atributo ObjectLayer::m_gameObjects).

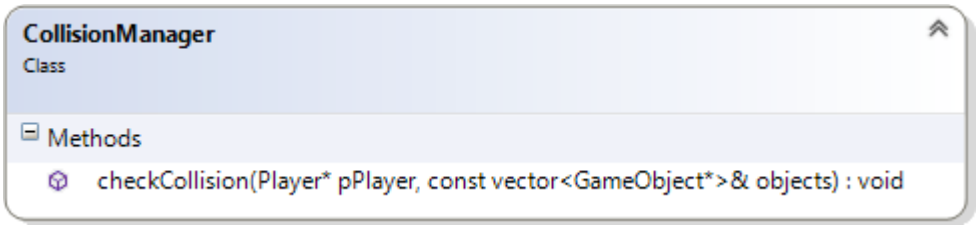


Figura 8.24 – Clase CollisionManager con su único método.

Una opción en principio más simple hubiera sido añadir a ObjectLayer un método checkCollision y llamarlo directamente, sin mediación del gestor. Se prefirió crear la clase CollisionManager por hacer el código más modular y facilitar la adición de funcionalidad relacionada con las colisiones.

8.11. Game loop: Render

Cuando ya se han leído los dispositivos de entrada y se ha actualizado la lógica del juego, se pasa a la fase de *render,* en la que se muestran en pantalla los resultados. Como se adelantó en la Sección 8.2, de esto se encargará el método render de la clase Game.

La estructura de la fase de *render* es muy similar a la de la fase de *update*: el método Game::render iniciará una secuencia de llamadas a los métodos de dibujado de las distintas clases del código, que en este caso terminará con las llamadas a las funciones correspondientes de un gestor de texturas que crearemos. Este gestor de texturas, al que llamaremos TextureManager, hará uso de la API de renderizado de SDL y, al igual que sus análogos el gestor de sonido (SoundManager) y el gestor de dispositivos de entrada (InputHandler), se implementará con el patrón singleton.

Los miembros de la clase TextureManager nos permitirán cargar y crear texturas (de tipo SDL_Texture) desde imágenes de archivo, dibujarlas y disponer de una lista de las que hayamos cargado, de manera que podamos usarlas cuando y donde las necesitemos. La Figura 8.25 contiene el diagrama de TextureManager con estos miembros, que se explican a continuación:

load Función encargada de cargar una textura en el manager. Se le pasará la ruta del archivo con la imagen que queremos usar, la ID con la que queremos referirnos a la textura, y el renderer que se va a usar.

m_textureMap Mapa asociativo con las texturas cargadas en el manager. La clave de cada par *clave-valor* es la ID para referirnos a la textura, y el valor un puntero a la estructura de tipo SDL_Texture que la contiene.

draw Se usa para dibujar una textura. Tomará como parámetros la ID de la textura que se quiere dibujar, la posición x e y que ocupará en la ventana, el renderer y un parámetro de tipo SDL_RendererFlip con el que indicaremos si invertir la textura.

drawFrame Especialización de draw diseñada para dibujar los frames de una animación. Tendrá algunos parámetros adicionales: currentFrame será un rectángulo que nos indicará que porción de la textura dibujar (o qué frame de la sprite sheet coger), con angle indicaremos si rotar la textura y con alpha si queremos aplicarle transparencia.

drawTile Diseñada para dibujar tiles a partir de la textura de un tileset, que será a la que nos refiramos con la ID que le pasemos. Además de los parámetros de draw, tendrá los siguientes: ancho y alto de las tiles, margen y espacio entre las tiles del tileset, fila y columna de la tile que queremos dibujar (currentRow para la fila y currentFrame para la columna).

~TextureMap En el destructor se recorrerá m_textureMap, liberando la memoria ocupada por cada textura por medio de la función SDL_DestroyTexture.

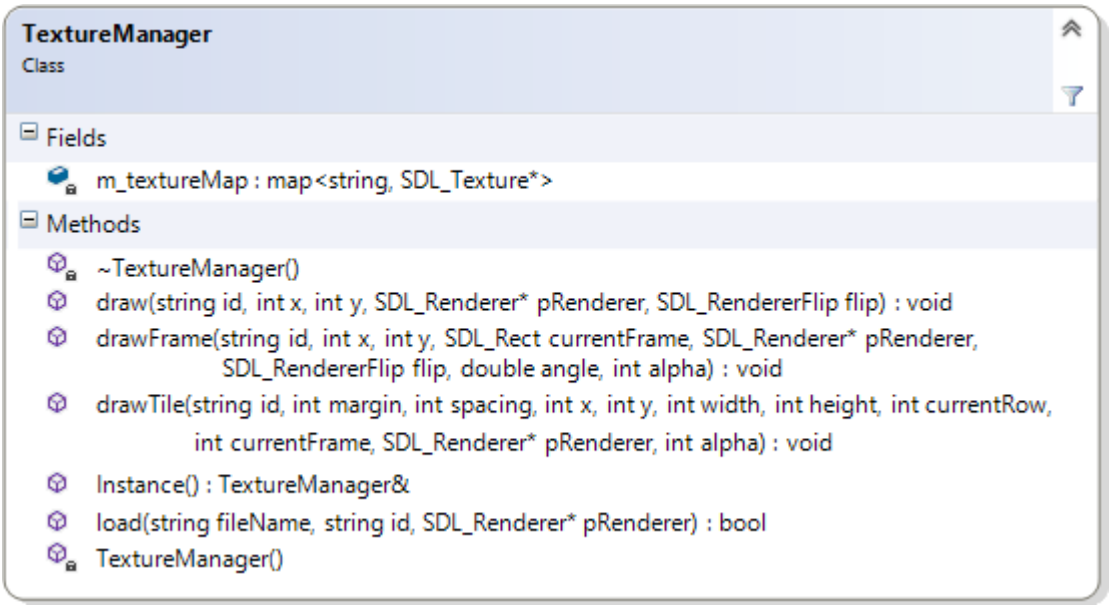


Figura 8.25 – Miembros de la clase TextureManager.

El renderer que se utilice en los métodos, tanto para la creación de texturas como para las distintas variaciones del método draw, será el atributo m_pRenderer de la clase central Game. Podría haberse eliminado el parámetro y usar este renderer directamente aprovechando que Game es un singleton y es accesible desde TextureManager, pero de esta forma mantenemos el módulo de gestión de texturas independiente, por si en el futuro quisiera usarse de otra forma.

Habiendo presentado la clase TextureManager, volvemos a la secuencia de llamadas iniciada por Game::render, que será la siguiente:

- Game::render llama a GameStateMachine::render, que llamará a GameState::render, es decir, al método de dibujado del estado actual. Si el estado actual es distinto de PlayState, sólo habrá que dibujar una pantalla estática, por lo que se hará una llamada al método draw de TextureManager y terminará la secuencia de llamadas.
- Si el estado actual es PlayState, se llamará al método render del nivel actual.
- Level::render llamará al método render de cada una de sus capas. Si la capa es de objetos la forma de proceder será distinta a si es de tiles:
  - ObjectLayer::render recorrerá todos los objetos del nivel, llamando al método draw de los objetos que se encuentren en el área visible del juego. En GameObject::draw se llamará al método drawFrame del gestor de texturas con los parámetros necesarios.

- TileLayer::render iterará a través del número de filas y columnas de la capa de tiles, dibujando sólo las tiles dentro del área visible. Para ello se llamará al método drawTile del gestor de texturas.

En la Figura 8.26 puede verse un esquema que ayudará a entender el proceso.

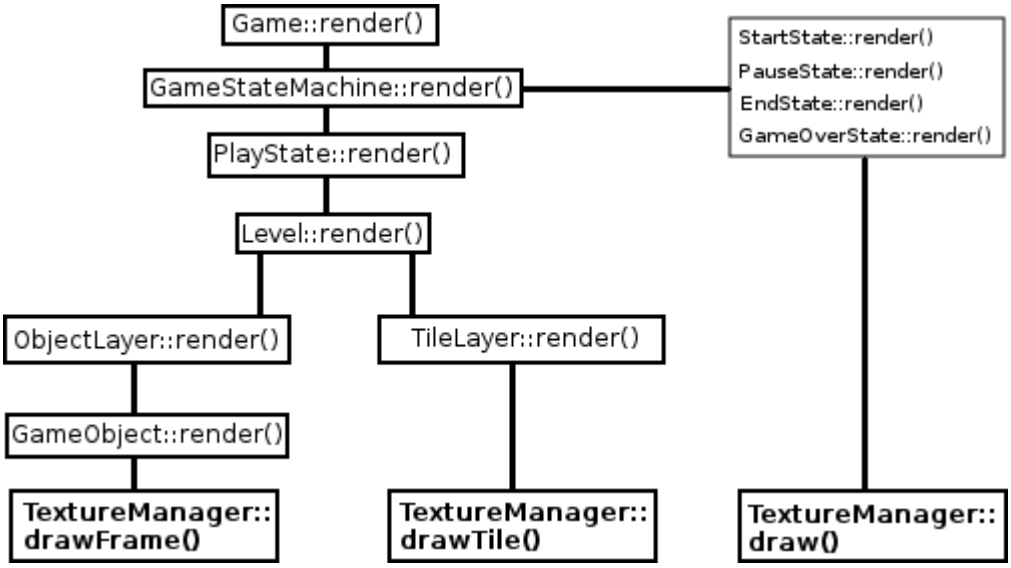


Figura 8.26 – Esquema del proceso de dibujado en pantalla.

Implementación del gestor de texturas

Las dos acciones principales para las que usaremos la clase TextureManager son la carga de texturas para su posterior uso y el dibujado de estas texturas.

En SDL, las texturas tienen su propio tipo de datos, llamado SDL_Texture. Para tratar con texturas es necesario un SDL_Renderer que las dibuje en pantalla. Podemos crear una SDL_Texture haciendo uso de la siguiente función:

```
SDL_Texture* SDL_CreateTextureFromSurface(SDL_Renderer* renderer,
                                          SDL_Surface*  surface)
```

La función devuelve un puntero a una estructura SDL_Texture creada a partir de una superficie SDL, de tipo SDL_Surface, en la que previamente habremos cargado la imagen que queremos dibujar. SDL_Surface es una estructura que contiene los píxeles de una imagen junto con todos los datos necesarios para renderizarla.

Las superficies SDL pueden utilizarse para dibujar en pantalla (sin hacer uso de las texturas), pero usan renderizado por software, es decir, hacen uso de la CPU. Puesto que nosotros queremos hacer uso de la API de renderizado hardware de SDL 2, sólo utilizaremos las superficies como paso intermedio para obtener texturas. A continuación veremos cómo se realiza este proceso ayudándonos del código del método load del gestor de texturas, cuyo prototipo es el siguiente:

```
bool TextureManager::load(std::string fileName, std::string id, SDL_Renderer*
    pRenderer);
```

fileName Ruta del archivo con la imagen que queremos cargar.

id Nombre con el que identificaremos a la textura creada a partir de fileName.

pRenderer SDL_Renderer que se utilizará. Siempre pasaremos el atributo m_pRenderer de Game, pero se deja el prototipo abierto al uso de otro renderer.

Así, una vez cargada una textura en el manager, ésta se introducirá en el mapa m_textureMap con la id pasada como clave, permitiéndonos referirnos a ella más adelante por medio de dicha id. Veamos la definición del método load paso a paso:

```
bool TextureManager::load(std::string fileName, std::string id, SDL_Renderer*
    pRenderer)
{
  SDL_Surface* pTempSurface = IMG_Load(fileName.c_str());
  if (pTempSurface == nullptr)
  {
    printf("IMG Error: %s\n", IMG_GetError());
    return false;
  }
```

Lo primero que hacemos es cargar la imagen pasada (fileName) en una SDL_Surface. Para ello hemos usado la función IMG_Load de la extensión SDL_Image. También puede utilizarse la función SDL_LoadBMP de la biblioteca básica de SDL, pero como su nombre indica está limitada a archivos de imagen BMP. Comprobamos si la superficie SDL se ha creado correctamente y, de no ser así, reportamos el error y salimos de la función devolviendo false para indicar que se ha fracasado.

Si no ha habido problema creando la estructura SDL_Surface, pasamos a crear la textura:

```
  SDL_Texture* pTexture = SDL_CreateTextureFromSurface(pRenderer,
     pTempSurface);
  SDL_FreeSurface(pTempSurface);
```

Utilizamos la función SDL_CreateTextureFromSurface, previamente mencionada, para crear una estructura SDL_Texture a la que apuntará la variable pTexture. Puesto que ya no necesitamos más la superficie SDL, la liberamos por medio de la función SDL_FreeSurface. A continuación comprobaremos que la textura SDL se creó bien para proceder a incluirla en el mapa:

```
  if (pTexture != nullptr)
  {
    if (m_textureMap.find(id) != m_textureMap.end())
    {
      SDL_DestroyTexture(m_textureMap[id]);
    }
    m_textureMap[id] = pTexture;
    return true;
  }
```

Si la textura SDL ha sido creada con éxito (pTexture es distinto de nulo), antes de incluirla en el mapa (atributo m_textureMap) comprobamos si ya existía una con esa ID y, de ser así, la destruimos. Posteriormente, añadimos a m_textureMap la nueva textura usando como clave la ID pasada. Es muy importante destruir la textura en caso de que ya exista una con esa ID, ya que de no hacerlo y simplemente sobrescribir la textura anterior, ésta quedaría ocupando

memoria que no podemos liberar, ya que m_textureMap[id] haría referencia a la nueva. Una vez cargada la textura en el mapa, devolvemos true para indicar que el método load tuvo éxito.

Por último, si pTexture es igual a nullptr, significa que no se ha podido crear y cargar la textura correctamente, por lo que reportamos el error ocurrido y devolvemos false:

```
  cout << "Unable to load '" << id << "'. IMG Error: " << IMG_GetError() <<
      endl;
  return false;
}
```

Con esto concluye la carga de una textura en TextureManager. Posteriormente se podrán dibujar las texturas cargadas, llamando a los métodos de dibujado del gestor con la ID correspondiente. Como se comentó previamente, hay varios de estos métodos, por si quiere dibujarse una textura completa (draw), un fragmento de una textura (drawFrame) o una tile de un tileset (drawTile), pero todos ellos se basan en la siguiente función de SDL:

```
int SDL_RenderCopyEx(SDL_Renderer*          renderer,
                     SDL_Texture*           texture,
                     const SDL_Rect*        srcrect,
                     const SDL_Rect*        dstrect,
                     const double           angle,
                     const SDL_Point*       center,
                     const SDL_RendererFlip flip)
```

renderer Renderer que se va a usar.

texture Textura de la que se tomará el rectángulo fuente.

srcrect Rectángulo fuente, que delimita la porción de la textura que se va a dibujar.

dstrect Rectángulo destino, es decir, porción de la ventana donde se va a dibujar.

angle Ángulo en grados que indicará la rotación aplicada a dstrect.

center Puntero a un punto SDL alrededor del cual se rotará dstrect

flip Un valor de tipo SDL_RendererFlip que indicará si invertir la textura horizontalmente (SDL_FLIP_HORIZONTAL), verticalmente (SDL_FLIP_VERTICAL) o no invertirla (SDL_FLIP_NONE).

Todos los métodos de dibujado de TextureManager son similares, y se basan en obtener los rectángulos fuente y destino a partir de los argumentos que se les pasen. Se ha elegido el método drawFrame para comentar su código, ya que además hace uso del ángulo y la componente alfa. Veamos su prototipo y el significado de sus argumentos:

```
void drawFrame(std::string id, int x, int y, SDL_Rect currentFrame,
    SDL_Renderer* pRenderer,
  SDL_RendererFlip flip = SDL_FLIP_NONE, double angle = 0., int alpha = 255);
```

id ID de la textura previamente cargada que se quiere dibujar.

x Componente x del rectángulo destino.

y Componente y del rectángulo destino.

currentFrame Rectángulo fuente (de tipo SDL_Rect), que contiene el fragmento de la textura que se quiere dibujar.

pRenderer Renderer que se va a utilizar. En este proyecto siempre se utilizará el de la clase Game.

flip Indica si invertir horizontalmente, verticalmente o no invertir la textura. Por defecto no se invertirá (SDL_FLIP_NONE).

angle Ángulo que se rotará la textura. Por defecto sin rotación (0).

alpha Valor de 0 a 255 que indicará lo transparente u opaca que se dibujará la textura. Por defecto será totalmente opaca (255).

Como vemos, algunos de los parámetros podrán pasarse directamente a la función SDL_RenderCopyEx, pero otros necesitarán tratarse previamente. Pasemos a ver la definición de drawFrame:

```
1  void TextureManager::drawFrame(std::string id, int x, int y,  SDL_Rect
       currentFrame, SDL_Renderer* pRenderer, SDL_RendererFlip flip, double angle
       , int alpha)
   {
     SDL_Rect dstRect = { x, y, currentFrame.w, currentFrame.h};
4    SDL_SetTextureAlphaMod(m_textureMap[id], alpha);
     if (SDL_RenderCopyEx(pRenderer, m_textureMap[id], &currentFrame, &dstRect,
        angle, 0, flip) < 0)
     {
7      printf("Couldn't draw frame of texture '%s'. SDL_Error: %s\n", id.c_str()
          , SDL_GetError());
     }
     SDL_SetTextureAlphaMod(m_textureMap[id], SDL_ALPHA_OPAQUE);
10 }
```

Ya tenemos el rectángulo fuente en el formato necesario, dado por currentFrame, por lo que lo primero que se hará es calcular el rectángulo destino (línea 3). Los parámetros x e y nos dan el origen, y el ancho y el alto coincidirán con los del rectángulo fuente, ya que queremos dibujar el fragmento de textura al completo.

A continuación indicaremos el componente alfa de la textura mediante la función SDL_SetTextureAlphaMod, a la que le pasaremos la textura y el valor de alfa.

Ya con el rectángulo destino calculado y el componente alfa establecido, pasamos a llamar a SDL_RenderCopyEx con los argumentos correspondientes (línea 5). La función devolverá cero si tiene éxito o un código de error negativo en caso contrario, por lo que comprobamos si el resultado es menor que cero y, de ser así, reportamos el error.

Finalmente, volvemos a hacer opaca la textura (línea 9), pues queremos que la componente alfa de la textura se modifique sólo en este dibujado, no permanentemente.

Implementación de la secuencia de dibujado

Al igual que ocurría con el proceso de *update*, el proceso de *render* comienza con una llamada al método de la clase Game:

```
void Game::render()
{
  SDL_RenderClear(m_pRenderer);
  m_pGameStateMachine->render();
  SDL_RenderPresent(m_pRenderer);
}
```

La llamada a GameStateMachine::render es la que desencadenará la secuencia que terminará con el dibujado de las capas de tiles y objetos del nivel actual. Pero antes fijémonos en las dos funciones de SDL que la envuelven:

- SDL_RenderClear, limpiará o borrará el objetivo del renderer (la ventana del juego) con el color de dibujado que se haya especificado[11].
- SDL_RenderPresent, actualizará la pantalla con cualquier renderizado llevado a cabo desde la llamada anterior a esta misma función.

Es importante que el dibujado de texturas se realice entre estas dos llamadas, pues si no se llama a SDL_RenderClear estaremos dibujando sobre lo anteriormente dibujado, y si no se llama a SDL_RenderPresent no se verán en la ventana las texturas que se hayan dibujado. Esto último es así porque las funciones de renderizado de SDL operan sobre un *buffer* trasero; es decir, llamar a una función como SDL_RenderCopyEx no plasma directamente una textura en pantalla, sino que actualiza dicho buffer. De esa manera se puede ir componiendo la escena entera en el buffer trasero y pasarlo a la pantalla como una imagen completa, lo cual es precisamente lo que hace SDL_RenderPresent.

Ahora sí, veamos la definición del método render de la FSM:

```
void GameStateMachine::render()
{
  if (!m_gameStates.empty())
  {
    m_gameStates.back()->render();
  }
}
```

Se llama al método render del estado actual, que será mucho más sencillo para los estados compuestos de pantallas estáticas que para PlayState, al igual que sucedía con update. Empecemos estudiando el render del estado estático StartState:

```
void StartState::render()
{
  TheTextureManager::Instance().draw(m_imageID, 0, 0, TheGame::Instance().
      getRenderer());
}
```

Puesto que sólo tiene que dibujarse una imagen constantemente (la imagen de inicio del juego), basta con una llamada al método draw del gestor de texturas indicando la imagen (m_imageID es un atributo de los estados estáticos con la ID de la imagen que muestran), el origen (0, 0) y el renderer (que será el de Game). Con esto se terminaría la secuencia de dibujado (ver

[11] Para cambiar el color de dibujado del renderer, usar SDL_SetRenderDrawColor.

Figura 8.26), se volvería a Game::render y se llamaría a SDL_RenderPresent, mostrándose la pantalla de inicio.

El render de PlayState es algo más complejo:

```
void PlayState::render()
{
  if (m_pLevel != 0)
  {
    m_pLevel->render();
  }
  drawHUD();
}
```

Se llamará al render del nivel actual, y posteriormente se llama al método privado drawHUD, que se encarga de dibujar el HUD con la información sobre el estado del jugador. La llamada a drawHUD es la última que se realiza porque queremos que el HUD se dibuje sobre todo lo demás. La definición de Level::render es la siguiente:

```
void Level::render()
{
  for (int i = 0; i < m_layers.size(); i++)
  {
    m_layers[i]->render();
  }
}
```

Se recorren las capas del nivel llamando al método render de cada una de ellas, que será distinto si se trata de una capa de tiles (TileLayer) o de objetos (ObjectLayer). Comenzaremos por el dibujado de la capa de tiles:

```
void TileLayer::render()
{
  for (int i = 0; i < m_numRows; i++)
  {
    for (int j = 0; j < m_numColumns; j++)
    {
      int id = m_tileIDs[i][j];
      if ( (id == 0) || (tile_outside_camera) )
      {
        continue;
      }
      Tileset tileset = getTilesetByID(id);
      if (tileset.firstGridID)
      {
        TheTextureManager::Instance().drawTile(...);
      }
    }
  }
}
```

Empezamos con dos bucles anidados para recorrer todas y cada una de las tiles de la capa (los atributos m_numRows y m_numColumns indican el número de filas y columnas). El vector m_tilesIDs[i][j] contiene en cada iteración el valor numérico que identifique a la tile actual, es decir, su ID.

Obtenemos la ID de la tile (línea 7) y comprobamos si es necesario dibujarla, para lo que deben darse dos condiciones: que exista una tile en esa posición (ID distinta de cero) y que esté dentro del área visible (se han sustituido las verdaderas comprobaciones matemáticas por tile_outside_camera por simplicidad). De no cumplirse alguna, continuamos iterando sin dibujarla.

Si hay que dibujar la tile, pasamos a obtener el tileset al que pertenece mediante el método privado TileLayer::getTilesetByID, que a partir de la ID de una tile nos devuelve el tileset que la contiene. A continuación comprobamos que el tileset es válido (si su primera ID global es mayor que cero) y, de ser así, llamamos al método drawTile del gestor de texturas (línea 15). Sumando la información del tileset a la que ya conocíamos (tamaño de tile, posición de la cámara, tile actual, etc.) es posible calcular los valores correspondientes que pasarle a este método (ver prototipo en Figura 8.25) .

Con esto ya se habrán dibujado las tiles que componen el escenario. A continuación se mostrará el código del dibujado de la capa de objetos:

```
void ObjectLayer::render()
{
  GameObject* pPlayer = nullptr;
  for (auto object : m_gameObjects)
  {
    if (object_inside_camera)
    {
      if (object->type() == "Player")
      {
        pPlayer = object;
      }
      else
      {
        object->draw();
      }
    }
  }
  if (pPlayer != nullptr)
  {
    pPlayer->draw();
  }
}
```

Al inicio nos creamos una variable pPlayer de tipo GameObject*, cuyo propósito conoceremos en unos instantes. A continuación recorremos el vector m_gameObjects que contiene punteros a todos los GameObject de la capa. Para cada objeto hacemos una serie de comprobaciones:

- Primero comprobamos que el objeto se encuentra dentro del área visible (línea 6), o lo que es lo mismo, dentro de la cámara. Se han sustituido las verdaderas comprobaciones por object_inside_camera para simplificar la explicación. Si el objeto no está dentro del área visible, no es necesario dibujarlo, por lo que no hacemos nada e iteramos hacia el siguiente objeto.
- Si el objeto está dentro del área visible, comprobamos si es el jugador (de tipo Player) y, de ser así, hacemos que el puntero pPlayer definido al inicio de la función apunte al objeto (línea 10). Esto tiene como objetivo dibujar el jugador después (y por tanto encima) de todos los objetos. Si el objeto no es de tipo Player, llamamos ya a su método de dibujado (línea 14), que en el caso de GameObject hemos llamado draw en lugar de render.

Por último, al salir del for, dibujamos el jugador (línea 20) si existe, pues puede haber capas de objetos sin jugador, como por ejemplo ocurre en las capas de backgrounds.

El método GameObject::draw es virtual puro, por lo que habrá que implementarlo para cada objeto derivado. Sin embargo, al contrario de lo que ocurría con update, el proceso de dibujado es muy similar para todos los objetos, pues ya sabremos la animación, el frame y la posición actual del objeto (calculados debidamente en update) y sólo restará llamar al método TextureManager::drawFrame con los argumentos adecuados.

Por lo comentado en el párrafo previo, se ha decidido implementar draw en la clase base para que las clases derivadas puedan simplemente llamar a GameObject::draw. El método se define como sigue:

```
void GameObject::draw()
{
  TheTextureManager::Instance().drawFrame(getTextureID(),
    m_position.x() - TheCamera::Instance().getPosition().x(),
    m_position.y() - TheCamera::Instance().getPosition().y(),
    m_currentAnim.getCurrentFrame(),
    TheGame::Instance().getRenderer(),
    m_sdlFlip);
}
```

El **primer parámetro** es la ID de la textura que se va a dibujar. GameObject::getTextureID es un método que nos da la ID de la textura de la animación actual del objeto.

El **segundo y tercer parámetro** son las coordenadas de la posición del objeto en la ventana, corregidas con la posición de la cámara puesto que debemos proporcionar valores relativos (a la ventana) y no absolutos. Hasta ahora se ha hecho alguna mención a la cámara (llamándola también «área visible») pero no se había mostrado antes en el código. Si se ha dejado la clase Camera al margen de este documento es porque no es necesario saber los detalles de su diseño e implementación para entender su función, que es seguir a un objetivo (normalmente el jugador) y permitirnos conocer en todo momento el área del mapa que está viendo el jugador. Como puede intuirse por la forma de llamar a sus métodos, se ha implementado como un singleton.

El **cuarto parámetro** es el frame de la animación que toca dibujar, que se obtenemos por medio del método Animation::getCurrentFrame. El atributo GameObject::m_currentAnim contiene la animación actual.

Como **quinto parámetro** pasamos el renderer de Game y como **sexto parámetro** el atributo GameObject::m_sdlFlip de tipo SDL_RendererFlip, que indicará si se debe invertir la textura o no.

Ahora podemos hacer referencia a este método desde las clases derivadas. Por ejemplo, el método draw de la clase Enemy será tan simple como:

```
void Enemy::draw()
{
  GameObject::draw();
}
```

Y lo mismo ocurre con la clase Item. Para las clases Background y Player el método varía brevemente, pero el fundamento es el mismo.

8.12. Salir del juego

En esta fase se liberarán los recursos que se hayan adquirido a lo largo de la ejecución del juego:

- En el destructor de SoundManager se recorrerá el mapa de efectos de sonido llamando a la función Mix_FreeChunk (función de SDL_Mixer para liberar la memoria de un sfx) con cada uno de ellos. Posteriormente se seguirá el mismo procedimiento con el mapa de pistas de música, usando en este caso la función Mix_FreeMusic, para finalmente cerrar la extensión SDL_mixer por medio de Mix_CloseAudio:

```
SoundManager::~SoundManager()
{
  for (auto sfx : m_sfxs)
  {
    Mix_FreeChunk(sfx.second);
  }
  m_sfxs.clear();

  for (auto music : m_music)
  {
    Mix_FreeMusic(music.second);
  }
  m_music.clear();

  Mix_CloseAudio();
}
```

- El destructor de TextureManager recorrerá el mapa de texturas llamando a la función SDL_DestroyTexture con cada una de ellas:

```
TextureManager::~TextureManager()
{
  for (auto texture : m_textureMap)
  {
    SDL_DestroyTexture(texture.second);
  }
  m_textureMap.clear();
}
```

- En el destructor de Game se destruirá la ventana SDL, el renderer SDL, la FSM y por último se cerrará SDL mediante SDL_Quit:

```
Game::~Game()
{
  SDL_DestroyWindow(m_pWindow);
  m_pWindow = nullptr;

  SDL_DestroyRenderer(m_pRenderer);
  m_pRenderer = nullptr;

  delete m_pGameStateMachine;
  m_pGameStateMachine = nullptr;

  SDL_Quit();
}
```

8.13. Gestión del tiempo

Hasta ahora se han explicado las distintas etapas del game loop (*user input*, *update* y *render*), pero no cómo se ha implementado la relación entre esas partes; en ello se centrará esta sección, particularmente en la solución adoptada para la **gestión del tiempo**. Veamos el siguiente código:

```
while (TheGame::Instance().running())
{
  TheGame::Instance().handleEvents();
  TheGame::Instance().update();
  TheGame::Instance().render();
}
```

Éste es nuestro game loop en su forma más básica. El problema es que este bucle simplemente se ejecuta, no hay ningún tipo de gestión del tiempo. En un hardware más lento el juego se ejecuta más lento, y en uno más rápido se ejecuta más rápido. Es inadmisible que la velocidad del juego sea distinta dependiendo de la plataforma, pues eso influye directamente en la jugabilidad del mismo. Hace tiempo esto no suponía un gran problema, pues la velocidad del hardware al que nuestro juego iba destinado era conocida en la mayoría de los casos, pero hoy en día hay tantas plataformas hardware que se hace necesario controlar el tiempo de algún modo.

Una solución sencilla al «problema» de la gestión del tiempo es actualizar y mostrar el juego a una velocidad constante, por ejemplo 50 veces por segundo:

```
const int FPS = 50;
const Uint32 dt = 1000. / FPS;
Uint32 nextGameTick = SDL_GetTicks();
int sleepTime = 0;

while (TheGame::Instance().running())
{
  TheGame::Instance().handleEvents();
  TheGame::Instance().update();
  TheGame::Instance().render();

  nextGameTick += dt;
  sleepTime = nextGameTick - SDL_GetTicks();
  if (sleepTime >= 0) SDL_Delay(sleepTime);
}
```

Si queremos una frecuencia de 50 FPS, cada frame deberá completarse en $1/50 = 0{,}02$ segundos. A este tiempo le llamaremos *tiempo delta*. En cada iteración del game loop llamaremos a Game::update y Game::render, esperando el tiempo sobrante. En el código de arriba se inicializa el tiempo delta en milisegundos ($1000/FPS$) debido a que las funciones SDL_GetTicks y SDL_Delay, pertenecientes al subsistema de timers de SDL explicado en el apartado 8.1.4, trabajan en esta unidad.

Esta solución tiene la gran ventaja de ser simple y funciona para cualquier hardware que soporte los FPS definidos. Sin embargo, presenta problemas en un hardware lento que no pueda soportar ese número de actualizaciones y dibujados por segundo, haciendo que el juego vaya más lento. En el peor de los casos existirían partes en las que el juego fuera extremadamente lento y partes en las que fuera normal, lo que acabaría haciéndolo injugable.

Una posible solución para ese problema es mantener el juego actualizándose al mismo ritmo, pero reducir el número de veces que se muestra. Esto puede hacerse a mediante el siguiente bucle:

```
const int updateRate = 50;
const Uint32 dt = 1000. / updateRate;
Uint32 nextGameTick = SDL_GetTicks();

while (TheGame::Instance().running())
{
  skippedFrames = 0;
  while (SDL_GetTicks() > nextGameTick)
  {
    TheGame::Instance().handleEvents();
    TheGame::Instance().update();
    nextGameTick += dt;
  }
  TheGame::Instance().render();
}
```

De esta forma el juego se actualizará a una velocidad constante de 50 veces por segundo, y el dibujado será tan rápido como sea posible; de esta forma, la velocidad y las sensaciones jugables del juego serán las mismas a 24, 30 o 50 FPS. Si el dibujado se produce a mayor frecuencia que el actualizado, algunos frames subsecuentes serán el mismo, por lo que realmente, a nivel visual nunca existirán más de 50 frames distintos cada segundo.

Puede pensarse que en un hardware más rápido se están «desperdiciando» ciclos de reloj, y es cierto. Esto puede ser un problema para juegos en tres dimensiones en los que la pantalla se llena de balas y efectos especiales, pues la diferencia entre 50 o 300 FPS tiene un efecto en la fluidez visual. Existen opciones para mejorar la última solución propuesta, de manera que el juego sea más atractivo visualmente en un hardware más rápido sin perder su eficiencia en uno más lento. Estas opciones implican la introducción e implementación de conceptos como interpolación o función de predicción, pero no iremos más allá, ya que en los juegos a los que en principio está destinado este framework difícilmente podrá notarse la diferencia con FPS superiores a 30, a lo que hay que añadir que una tasa de frames más alta implica un mayor consumo (lo cual puede suponer un problema en dispositivos dependientes de batería).

Ejemplo de función *main*

Puesto que los métodos de Game para actualizar y renderizar el juego se proporcionan por separado, cada usuario puede implementar la gestión del tiempo de la manera que considere oportuna. No obstante, para hacer más inmediato el uso del framework, se ha incluido una solución por defecto. Esta solución es ligeramente más compleja que la aquí explicada, en tanto que se hace uso de la API de timers de alta resolución de SDL (ver epígrafe 8.1.4) y se pone un límite inferior a los FPS. Para utilizar esta solución puede hacerse uso del método Game::iterate. Así, la función principal de nuestro proyecto podría tener la siguiente forma:

```
#include "Game.h"

int main(int argc, char *argv[])
{
```

```
  if (TheGame::Instance().init(SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED
      , 640, 480))
  {
    while (TheGame::Instance().running())
    {
      TheGame::Instance().iterate();
    }
  }
}
```

Con estas escasas líneas el juego se ejecutaría al completo, siempre y cuando existan el archivo XML global con las opciones y recursos del juego y los archivos XML con la información de los niveles (ver Sección 8.6).

Parte III

Editor de niveles S2PEditor

Capítulo 9

Análisis

En esta segunda parte del desarrollo software del proyecto, se construirá un editor que proporcionará una interfaz gráfica para hacer uso del framework S2Dp. La función de esta herramienta, a la que llamaremos S2PEditor (*SDL 2D Platformers Editor*) es permitir a cualquiera, tenga o no conocimientos de programación, crear juegos de plataformas en 2D sencillos a través de una GUI. El alcance de los juegos creados estará limitado por las opciones por defecto del framework, especificadas en la Sección 6.3.

En ocasiones nos hemos referido y referiremos a la herramienta S2PEditor como «editor de niveles», pues editar niveles será lo que hagamos la mayor parte del tiempo que la usemos, pero esto no quiere decir que su función sea generar niveles de forma aislada. La herramienta nos permitirá generar distintos juegos al completo, para los que además de sus niveles podremos definir opciones como la resolución, las pantallas (*start, pause, game over, end*), los valores iniciales y máximos de los atributos del jugador y el aspecto que éstos toman en el HUD. En la siguiente sección se describirán con mayor detalle, valiéndonos de los casos de uso, las acciones que puede llevar a cabo un usuario con la herramienta.

9.1. Casos de uso

Los casos de uso son una técnica para capturar los requisitos funcionales de un sistema, proporcionando una descripción de las interacciones típicas entre los usuarios y el mismo. Para el formato de redacción de los casos de uso se ha tomado como modelo el presentado en [29], que a su vez se basa en el propuesto por Cockburn[1]. Se usan los siguientes conceptos:

Goal Level Nivel del caso de uso. Cockburn propone un esquema de niveles, en el que los casos de uso principales serían de tipo *sea level*. A su vez, los casos de uso que existen únicamente porque son incluidos por los de tipo *sea level*, diríamos que son de tipo *fish level*.

Main Success Scenario (MSS) Secuencia de pasos numerados que describe el escenario principal en que un caso de uso termina con éxito.

Precondición Describe una condición que debe darse para que pueda iniciarse un caso de uso.

Extensión Variación del MSS.

No se especificarán los actores que toman parte en cada caso de uso, puesto que sólo existe uno y por tanto siempre es el mismo: el usuario que hace uso de la herramienta S2PEditor. Además, se han tomado algunas decisiones para simplificar la redacción:

[1] http://alistair.cockburn.us/

- Se han omitido los casos de uso relativos a eliminar o borrar elementos añadidos anteriormente, pues sus pasos iban a ser «seleccionar elemento» y «borrar elemento», lo cual no aporta apenas información y alarga la sección de manera innecesaria.
- Cada vez que se diga que el usuario *confirma* una acción, también puede cancelarla. Esto podría haber sido una extensión del caso de uso, pero se ha omitido por la misma razón que en el punto anterior.

A continuación se presentan los casos de uso, divididos según afecten a todo el proyecto[2], a un único nivel o al juego final.

9.1.1. Proyectos

Crear nuevo proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario elige la opción de crear un nuevo proyecto.
2. El usuario introduce el nombre del proyecto.
3. El usuario introduce los datos del primer nivel del proyecto.
4. El usuario confirma y el nuevo proyecto es creado junto a su primer nivel.

Cargar proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario elige la opción de cargar un proyecto existente.
2. El usuario selecciona el proyecto a cargar desde un explorador de archivos.
3. Se pregunta al usuario si desea guardar el proyecto actual.
4. Se carga el proyecto seleccionado en 2, pasando a ser el proyecto actual.

Guardar proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario elige la opción de guardar el proyecto actual.
2. El proyecto actual se guarda en el directorio destinado a tal fin.

[2]Hemos llamado un «proyecto» al conjunto de niveles y recursos a partir de los cuales el usuario construirá el juego.

Exportar proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario elige la opción de exportar el proyecto actual.
2. El usuario introduce el nombre que tendrá la carpeta a la que se exportará el proyecto.
3. El usuario elige, a través de un explorador de archivos, el directorio en que se creará la carpeta con el proyecto exportado.
4. En el destino elegido se crea una copia del proyecto actual, con sus recursos y niveles.

Editar texturas del proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario abre el gestor de texturas.
2. El usuario decide si añadir una nueva textura o borrar una existente.
3. El usuario confirma las modificaciones, saliendo del gestor.

Añadir nueva textura

Goal Level: Fish Level

Precondición: Se ha llegado al diálogo para añadir nueva textura a partir del gestor de texturas o alguna de las siguientes opciones: añadir objeto, editar objeto, añadir tileset, configurar opciones del juego, añadir background.

Main Success Scenario:

1. El usuario introduce la ID de la nueva textura.
2. El usuario introduce la ruta de la imagen a partir de la cual se creará la textura, mediante texto o explorador de archivos.
3. El usuario confirma, creándose una nueva textura con la imagen e ID indicadas.

Editar sonidos y música del proyecto

El proceso para gestionar sonidos es el mismo que el descrito para las texturas, con la salvedad de que el gestor de sonidos diferenciará entre efectos de sonido y pistas de música.

Añadir objeto al proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario selecciona el tipo del objeto que desea añadir (ver apartado 6.3.4 para conocer los posibles tipos).
2. El usuario inserta el nombre que se le dará al nuevo objeto.
3. El usuario selecciona la ID de una textura previamente añadida que hará de spritesheet del objeto.
4. El usuario inserta el alto y el ancho del sprite del objeto.
5. El usuario define la caja de colisión del objeto, numérica o gráficamente.
6. El usuario indica los valores numéricos especiales del objeto (velocidad, daño, salud, etc.), que dependerán de su tipo.
7. El usuario selecciona la ID de los efectos de sonido del objeto (sonido de un enemigo al ser dañado, sonido de un ítem al ser recogido, etc.), dependiendo de su tipo.
8. El usuario rellena la información de las animaciones del objeto. Para cada animación: fila de la spritesheet, número de frames y tiempo de espera de un frame al siguiente.
9. El usuario confirma y el nuevo objeto es añadido al proyecto, pudiendo usarse en todos los niveles del mismo.

Extensiones:

3a. El usuario añade una nueva textura al proyecto y la selecciona, retornando al paso 4.

7a. El usuario añade un nuevo efecto de sonido al proyecto y lo selecciona, retornando al paso 8.

Editar un objeto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario selecciona el objeto que desea editar.
2. El usuario selecciona la opción para editar el objeto seleccionado.
3. El usuario indica las modificaciones que desea introducir en el objeto.
4. Si el nombre del objeto ha cambiado, éste se añade al proyecto como un nuevo objeto.

Extensiones:

4a. Si el nombre del objeto no ha cambiado, se pide al usuario confirmación para sobrescribir el objeto.

Añadir tileset al proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario selecciona la opción para insertar un nuevo tileset al proyecto.
2. El usuario introduce el nombre del nuevo tileset.
3. El usuario introduce la ID de la textura fuente del tileset.
4. El usuario introduce el resto de datos del tileset: tamaño de las tiles, espacio entre ellas y margen con respecto al borde de la imagen. Todo en píxeles.
5. El usuario confirma y el tileset se añade al proyecto, permitiendo usar sus tiles en todos los niveles.

Extensiones:

3a. El usuario añade una nueva textura al proyecto, seleccionándola como textura fuente del tileset y retornando al paso 4.

9.1.2. Niveles

Añadir nivel al proyecto

Goal Level: Sea Level

Main Success Scenario:

1. El usuario selecciona la opción para añadir un nuevo nivel al proyecto actual.
2. El usuario introduce el nombre del nuevo nivel.
3. El usuario introduce el resto de datos del nuevo nivel: ancho en tiles, alto en tiles y tamaño de tiles en píxeles.
4. El usuario confirma los datos introducidos y el nuevo nivel se añade al proyecto.

Guardar copia del nivel actual

Goal Level: Sea Level

Main Success Scenario:

1. El usuario selecciona la opción para guardar una copia del nivel actual, es decir, el que se está editando.
2. El usuario introduce el nombre que tendrá el «nuevo» nivel.
3. Se crea un «nuevo» nivel, copia del actual, con el nombre dado en el paso 2.

Probar nivel

Goal Level: Sea Level

Main Success Scenario:

1. El usuario selecciona la opción para probar el nivel actual.
2. El nivel se ejecuta dentro del editor, pudiendo el usuario probarlo de la misma forma que si estuviera jugando a la versión final.
3. El usuario selecciona la opción para parar la ejecución del nivel y volver al modo de edición.

Gestionar backgrounds

Goal Level: Sea Level

Main Success Scenario:

1. El usuario abre el gestor de backgrounds.
2. El usuario decide si añadir un nuevo background, modificar la profundidad de los existentes o eliminar uno de ellos.
3. El usuario confirma las modificaciones, saliendo del gestor.

Añadir nuevo background

Goal Level: Fish Level

Precondición: Debe haberse abierto el gestor de backgrounds.

Main Success Scenario:

1. El usuario elige en el gestor de backgrounds la opción de añadir uno.
2. El usuario indica el tipo de background que desea añadir: estático, móvil o parallax.
3. El usuario selecciona la ID de la textura fuente del background.
4. El usuario confirma la adición del background al nivel.

Extensiones:

3a. El usuario añade una nueva imagen al proyecto y la selecciona.

4a. Si el background es móvil, el usuario elige la velocidad a la que se mueve en los ejes `x` e `y`. Retorna al punto 4.

Modificar profundidad de los background

Goal Level: Fish Level

Precondición: Deben existir en el nivel al menos dos backgrounds.

Main Success Scenario:

1. El jugador selecciona el background del que quiere cambiar la profundidad.
2. El usuario disminuye o aumenta la profundidad del background seleccionado con respecto al resto de backgrounds.

Asignar música al nivel

Goal Level: Sea Level

Precondición: Debe haberse añadido como mínimo una pista de música al proyecto.

Main Success Scenario:

1. El usuario selecciona la opción para asignar al nivel una pista de música.
2. El usuario selecciona la ID de la pista de música que desea asignar al nivel actual.
3. El usuario confirma su selección y la pista es asignada al nivel.

Insertar objeto en el nivel

Goal Level: Sea Level

Precondición: Debe haberse añadido como mínimo un objeto al proyecto.

Main Success Scenario:

1. El usuario selecciona el objeto que quiere insertar en el nivel, de entre los objetos añadidos al proyecto.
2. El usuario selecciona gráficamente el lugar del nivel en el que quiere insertar el objeto.

Insertar tile en el nivel

Goal Level: Sea Level

Precondición: Debe haberse añadido al proyecto como mínimo un tileset.

Main Success Scenario:

1. El usuario selecciona el tileset que contiene la tile que quiere insertar.
2. El usuario selecciona la tile que desea insertar.
3. El usuario indica si la tile va a ser colisionable o no.
4. El usuario indica gráficamente el lugar del nivel en que quiere insertar la tile.

9.1.3. Juego

Generar juego

Goal Level: Sea Level

Main Success Scenario:

1. El usuario configura las opciones del juego.
2. El usuario exporta el juego.

Configurar opciones del juego

Goal Level: Fish Level

Main Success Scenario:

1. El usuario selecciona la opción para editar las opciones del juego.
2. El usuario elige los valores iniciales y máximos para la salud, las vidas, la puntuación y el número de llaves del jugador.
3. El usuario selecciona la ID de las texturas que representarán en el HUD la salud, las vidas, la puntuación y el número de llaves del jugador.
4. El usuario selecciona la ID de las texturas para las pantallas de *start*, *pause*, *game over* y *end*.
5. El usuario indica la resolución de la ventana en que se ejecutará el juego.
6. El usuario indica qué niveles del proyecto compondrán el juego, y cuál será su orden.
7. El usuario confirma y las opciones del juego se sobrescriben.

Extensiones:

3a. El usuario añade una nueva textura al proyecto y la selecciona. Retorna al punto 4.

4a. El usuario añade una nueva textura al proyecto y la selecciona. Retorna al punto 5.

Exportar juego

Goal Level: Fish Level

Main Success Scenario:

1. El usuario selecciona la opción para exportar/crear el juego.
2. El usuario introduce el nombre del juego.
3. El usuario elige el directorio donde se creará la carpeta con el juego final.
4. En el directorio indicado en 3, se creará una carpeta con el nombre indicado en 2, que contendrá el ejecutable del juego.

9.2. Enfoque del desarrollo

Para construir el editor de niveles, es necesario un conjunto de clases que nos proporcione el abanico de componentes (*widgets*) típico de las aplicaciones de escritorio con interfaz gráfica de usuario (GUI): botones, *scrollbars*, diálogos de selección de archivo, *layout* automático, etc. Para alcanzar este objetivo se presentan varias opciones:

Integrar SDL en una biblioteca GUI existente. La idea es que el sistema de renderizado de SDL y el de la biblioteca GUI compartan la misma ventana, que contendría el escenario renderizado por SDL rodeado de los *widgets* de la biblioteca GUI.

Es el enfoque que primero se explorará por considerarlo el ideal, ya que para la visualización del nivel (y de los elementos que se vayan situando en él) se podrían reutilizar las clases creadas con C++ y SDL sin apenas cambios. Además sería posible probar el nivel en la misma ventana donde se está construyendo, lo cual ayudaría a que el editor en su conjunto se mostrara más fluido y compacto. Es una opción compleja, pues implica compatibilizar los sistemas de entrada/salida y (especialmente) renderizado de SDL con los de la biblioteca GUI elegida, además de establecer un sistema de mensajería entre ambas. Algunas pocas bibliotecas GUI permiten la integración directa con SDL pero, como se verá más adelante, se quedan lejos de las más maduras en cuanto a contenido, diseño y documentación.

Se podría argumentar que un inconveniente que no se está teniendo en cuenta en este enfoque y en el siguiente es el incremento del tamaño del proyecto, ya que implican incluir una biblioteca GUI y las que han alcanzado el nivel de madurez deseable son por lo general muy extensas. Estaríamos de acuerdo en que esto es un problema si la biblioteca se fuera a utilizar para renderizar la interfaz *in-game* (menús, vidas, inventario, etc.), ya que podría afectar al rendimiento del juego; sin embargo, la biblioteca GUI sólo se usará en el proyecto del editor de niveles. El juego que se genere al final no tiene ninguna dependencia con la biblioteca GUI, sólo renderiza a través de SDL, por lo que su tamaño y velocidad no se verán afectados en absoluto.

Ventajas:

- Diseñar y probar nivel en la misma ventana.
- Se puede reutilizar el código.
- Disponer de un conjunto de *widgets* testeados.
- Estética.

Inconvenientes:

- Compatibilizar los sistemas de entrada/salida y renderizado de SDL con los de una biblioteca GUI no es trivial.
- Hay que dedicarle tiempo a investigar bibliotecas GUI y, una vez elegida una, aprender a utilizarla.

Usar exclusivamente una biblioteca o framework de C++ orientada a GUI con madurez y largo recorrido, como Qt, GTK+, wxWidgets, etc. Se han descartado Microsoft Foundation

Classes (MFC) y Windows Forms porque uno de los objetivos del PFC es usar tecnologías multiplataforma.

Esta solución presenta la ventaja de que, una vez elegida y aprendidas las bases de la biblioteca GUI, sólo hay que preocuparse de implementar el editor, el cual sería totalmente independiente de SDL. No obstante, existe un inconveniente inmediato: habría que implementar la visualización del nivel con las clases de la biblioteca GUI que elija, sin poder reutilizar directamente el código creado hasta ahora. Además, no se podría probar el nivel en la misma ventana en la que se está editando; cuando el usuario pulse el botón para probar el nivel habría que llamar a la aplicación creada con SDL, la cual generaría una nueva ventana que se cerraría cuando el usuario quiera terminar de probar el nivel y seguir editándolo.

Ventajas:

- Disponer de un conjunto de *widgets* testeados.
- Estética.
- No hay que lidiar con problemas de compatibilidad entre SDL y la biblioteca GUI.

Inconvenientes:

- No se puede reutilizar el código.
- Diseño y testeo del nivel en ventanas separadas.
- Hay que dedicarle tiempo a investigar bibliotecas GUI y, una vez elegida una, aprender a utilizarla.

Implementar una biblioteca GUI propia con C++ y SDL. El primer problema que viene a la mente al pensar en este enfoque es la complejidad técnica asociada a desarrollar una biblioteca GUI, lo cual puede ser un proyecto titánico que bien podría suponer otro PFC. Evidentemente, en este caso se desarrollaría una biblioteca que sólo aportara los *widgets* imprescindibles para programar el editor, pero incluso así conllevaría un gran trabajo si se quiere realizar una implementación decente, para obtener finalmente un resultado menos vistoso y familiar para el usuario que con cualquiera de las dos opciones anteriores. Además, la inclusión de nuevas características en el editor tendría asociado un coste temporal muy alto, ya que habría que diseñarlas e implementarlas, mientras que si estoy usando una biblioteca GUI sólo tendría que consultar la documentación y usar el componente que necesite.

El aspecto final de la interfaz, como se sugiere en el párrafo anterior, es otro inconveniente notable de esta solución, pues habría que dedicarle mucho tiempo para lograr que la estética de todos los *wigdets* sea coherente y agradable a la vista, objetivo que queda fuera de los límites del PFC y de los conocimientos del que suscribe.

Ventajas:

- No hay que elegir ni estudiar nuevas bibliotecas.
- Reutilización de código.

- No hay que lidiar con problemas de compatibilidad entre SDL y la biblioteca GUI.
- Sólo añadiríamos al proyecto lo necesario.
- Diseñar y probar nivel en la misma ventana.

Inconvenientes:

- Implementación de una biblioteca GUI desde cero.
- Aspecto de la interfaz.
- Los añadidos tendrán un coste temporal muy alto.

De los tres enfoques planteados, se ha decidido optar por el primero, pues nos permite disponer de todos los *widgets* de una biblioteca GUI contrastada pudiendo a la vez reutilizar el código y probar el nivel en el mismo espacio en que se está diseñando. En el siguiente capítulo se describirá el proceso seguido para elegir dicha biblioteca.

Capítulo 10

Comparativa de bibliotecas GUI

En este capítulo se describirá el proceso descrito para seleccionar la herramienta con la que se construirá la interfaz gráfica del editor S2PEditor. La mayor parte del capítulo se centrará en evaluar la compatibilidad de estas herramientas con SDL, pues será éste el principal criterio de selección.

10.1. Bibliotecas ligeras

A continuación se muestran una serie de bibliotecas para desarrollar interfaces gráficas de usuario en C++. Se irán presentando en el orden en que se fueron probando, explicando en cada caso la razón por la que fueron descartadas frente a la opción finalmente elegida: el framework Qt.

10.1.1. SDL2-widgets

Fue la primera en ser probada por ser ligera, sencilla y, sobre todo, por estar construida con SDL, con lo que se nos facilitaría mucho el paso de la integración. En su web[31] se comenta que está testeada en Linux (Fedora) y en Windows (usando MinGW), aunque en este último SO no fue el autor quien lo probó.

Desde un principio llama la atención la escasa documentación. Las indicaciones para su instalación en Windows se reducen a un archivo *readme* con el siguiente contenido:

```
STEPS TO BUILD ON WINDOWS WITH MINGW
====================================
You will need MSYS to run configure and make
(1) There are 2 truetype files: DroidSans.ttf and DroidSansMono.ttf
(2) Run command: ./configure
(3) Run make
(4) To run, every executable will need in their own folder both TTFs.
    They also need (either installed or in their folder) all these DLLS:
SDL2.dll
SDL2_ttf.dll
libfreetype-6.dll
libgcc_s_dw2-1.dll
libstdc++-6.dll
```

Por su parte, no existe información sobre cómo utilizarla más allá de tres pequeños ejemplos sin explicar y sin comentar y, como cabe esperar de un proyecto tan modesto y limitado como éste, la comunidad detrás de él es inexistente.

A pesar de los inconvenientes comentados, se compiló la biblioteca y se intentaron ejecutar los ejemplos que se incluían, llegando a un punto en el que había un bug reportado de MinGW relacionado con la biblioteca de C *math.h* y el uso de C++ 11, imprescindible en la versión 2 de SDL-widgets. Se podría intentar con la versión 1, la cual no utiliza C++ 11, pero no funcionaría para el fin de este proyecto ya que utiliza la primera versión de SDL.

Es posible que el bug se pudiera solucionar dedicándole tiempo y estudio, pero aunque se solucionara, la precariedad de la biblioteca unida a la falta de documentación y una comunidad que la apoye, hacen que no haya dudas a la hora de descartarla.

10.1.2. Game Gui

Las razones por las que se tuvo en cuenta esta librería son muy similares a las de la anterior: es ligera y, aunque no está programada sobre SDL, en su página web[32] se indica que pueden ser usadas conjuntamente con facilidad. El autor explica también su motivación para crearla:

> Main reason for the development of the GUI was, that after trying several gui's more or less suited for games (libRocket, Qt, NVWidgets, CEgui, etc), I found that none of them was really light weight, easy to use and fully skinable at the same time. So the solution was to create one which is exactly so.

La duración de un mes del proyecto, unida a que se compone de sólo unas 2000 líneas de código, nos indican que nos encontramos nuevamente ante una biblioteca *amateur* sin aspiraciones a reunir una gran comunidad alrededor de ella, y la ausencia de documentación nos lo confirma; no obstante, ejecutamos un programa de prueba aprovechando que incluye un archivo con una solución de Microsoft Visual Studio, obteniendo el resultado de la Figura 10.1.

El aspecto de los widgets es suficientemente correcto para lo que se busca, pero sus limitaciones y la falta de documentación hacen que se haya decidido no usarla.

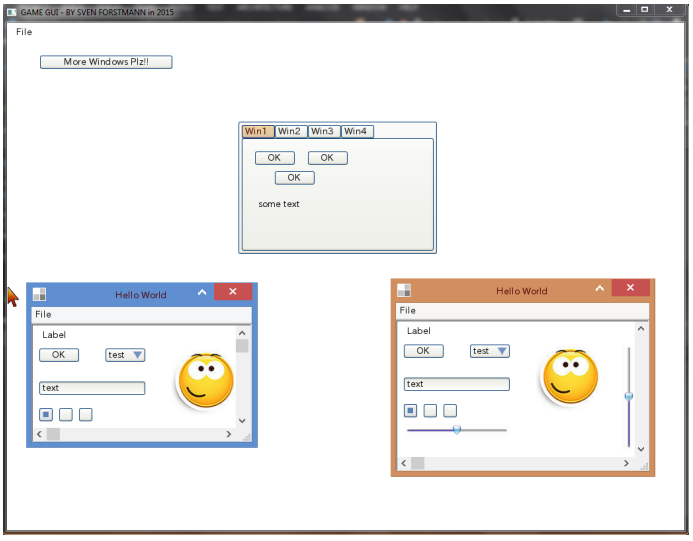


Figura 10.1 – Programa de ejemplo incluido con Game Gui

10.1.3. libRocket

libRocket es una biblioteca para la construcción de interfaces gráficas de usuario en C++, basada en los estándares HTML y CSS. Está licenciada bajo la licencia MIT[33]. Basta hacer un recorrido por su página web para comprender que nos encontramos ante la primera biblioteca «seria» de las que hemos probado. La documentación[34], aunque lejos de la excelencia, parece estar relativamente organizada, y al basarse en HTML/CSS es posible resolver algunas de las dudas que surgen con los widgets consultando estos estándares. Además cuenta con tutoriales, de los cuales algunos se detallarán más adelante.

libRocket ha sido desarrollada para su uso en las siguientes plataformas:

- Windows 32-bit compilando con Microsoft Visual Studio**.**
- MacOSX Intel 32/64bit compilando con GCC 4.
- Linux compilando con GCC 4.

Instalar libRocket en Microsoft Visual Studio

Lo primero es descargarse desde Github la última versión estable de la biblioteca[35]. libRocket se divide en tres módulos: RocketCore, RocketControls y RocketDebugger, cada uno de los cuales se debe incluir y enlazar por separado. No se incluyen los archivos binarios (los archivos .lib y .dll correspondientes), por lo que tenemos que compilarla y generarlos nosotros mismos. Por fortuna, sí que se incluye un fichero «CMakeLists.txt», lo cual nos permitirá crear un proyecto para Microsoft Visual Studio (MVS) con CMake.

libRocket tiene una dependencia con la biblioteca freetype2, la cual tenemos que añadir en el directorio «Dependencies» de la biblioteca. Una vez descargada la versión de freetype2 indicada en los *readme* que se incluyen con libRocket, se construye el proyecto para MVS con ayuda de CMake-GUI, una interfaz gráfica para CMake. Abrimos dicho proyecto y, tras modificar levemente el código para resolver algunos fallos relativos a los *includes*, logramos compilar la primera versión de libRocket.

Junto a la biblioteca se incluyen una serie de ejemplos, siendo el más interesante el llamado *Rocket Invaders*, un clon del popular *Space Invaders* diseñado por Tomohiro Nishikado y lanzado al mercado en 1978. Se procede pues a ejecutar este ejemplo para comprobar que la biblioteca compilada funciona correctamente, y el proceso se completa sin problemas, exceptuando algunas modificaciones menores en el código. En las figuras 10.2 y 10.3 podemos ver el aspecto de *Rocket Invaders* y de algunos widgets (principalmente botones y etiquetas) de la biblioteca libRocket, tanto en el menú principal como en medio de una partida.

Usar libRocket junto a SDL

Ahora que hemos compilado la biblioteca y comprobado que funciona, es el momento de testear su compatibilidad con la versión 2 de SDL. Junto con libRocket se proporcionan ejemplos para integrarla con algunas de las API para renderizado de gráficos con C++ más populares, en concreto DirectX, DirectX10, Ogre3D, SFML, SFML2 y la que nos interesa: SDL2.

El ejemplo de uso conjunto de libRocket y SDL2 se compone de cinco archivos:

Figura 10.2 – Menú principal de *Rocket Invaders*

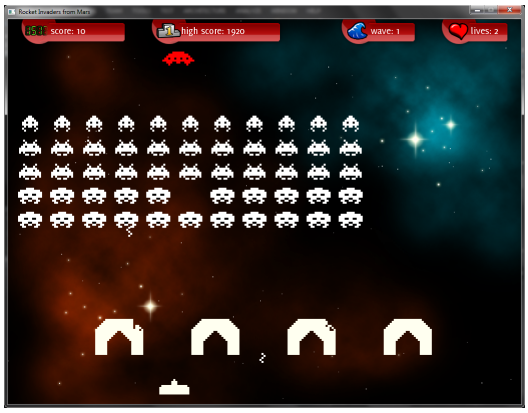


Figura 10.3 – Primer nivel de *Rocket Invaders*

- **SystemInterfaceSDL2.h** y **SystemInterfaceSDL2.cpp:** Se encargan de la compatibilidad entre los sistemas de entrada de libRocket y de SDL2, realizando un mapeo entre los códigos que cada biblioteca utiliza para las pulsaciones de teclado y de ratón.
- **RenderInterfaceSDL2.h** y **RenderInterfaceSDL2.cpp**: Presentan una complejidad mayor que los anteriores, y aportan una clase llamada RocketSDL2Renderer cuya función es permitir inicializar el sistema de renderizado de libRocket a partir de un *renderer* de SDL2 (SDL_Renderer), lo cual nos permite hacer llamadas tanto a libRocket como a SDL2 para que realicen el dibujado en la misma ventana.
- **main.cpp:** Se utiliza SDL para crear una ventana (SDL_Window) y un SDL_Renderer que dibuje en ella, para después crear un RocketSDL2Renderer pasándoselo como parámetro. Posteriormente se inicia el bucle de renderizado, dentro del cual se llaman a las funciones de dibujado de ambas biblioteca en cada iteración.

Es importante indicar que la clase RocketSDL2Renderer realiza el dibujado con OpenGL, por lo que hay que indicarle al SDL_Renderer que también la utilice, lo cual nos impide renderizar por medio de DirectX, que es lo que hace por defecto SDL en Windows por estar más optimizada. Esto no supone un gran problema, ya que se puede utilizar SDL2 + libRocket (con OpenGL) para el editor y exclusivamente SDL2 con DirectX para el juego final.

Al intentar compilar el ejemplo para ver que realmente libRocket y SDL2 comparten la misma ventana sin problemas, vemos que existe una dependencia la biblioteca GLEW[1]. Tras descargarla e incorporarla al proyecto, ejecutamos y vemos la ventana de la Figura 10.4:

El fondo azul se está dibujando a través de SDL2 y el recuadro de texto a través de libRocket, pues es un widget de esta biblioteca. Podemos hacer click sobre la barra *Title* para arrastrar y soltar el recuadro donde queramos dentro de la ventana.

Para que quede más claro que ambas bibliotecas están compartiendo la ventana y se aprecie que el cuadro de texto es transparente, se ha modificado el programa de prueba sustituyendo el fondo azul por una SDL_Texture con la imagen «HelloWorld» que se usó al inicio de este PFC en el ejemplo para decidir entre SDL, SFML y Monogame. El resultado puede verse en la

[1] http://glew.sourceforge.net/

Figura 10.5. El cuadro de texto soporta transparencia y *drag&drop*; se ha desplazado del centro para apreciar que la imagen renderizada por SDL se ve detrás.

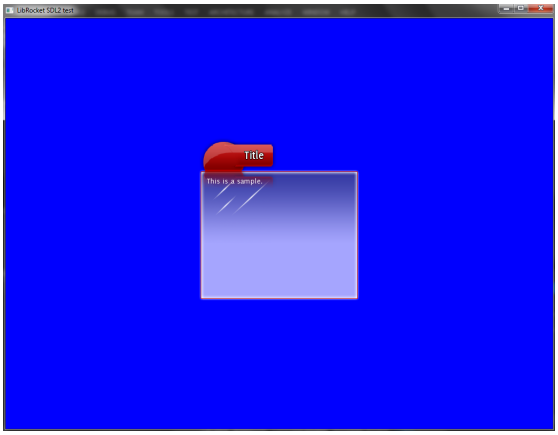


Figura 10.4 – SDL2 y libRocket compartiendo ventana.

Figura 10.5 – SDL2 y libRocket: *Hello World*.

Habiendo confirmado que integrar la biblioteca libRocket en un proyecto que usa SDL es posible y no presenta grandes complicaciones, se procedió a realizar los tutoriales[2] disponibles en la web oficial de libRocket para valorar la facilidad de uso, las posibilidades y la documentación de la biblioteca.

Conclusiones tras los tutoriales

Aunque el avance por los tutoriales es relativamente fluido y muestra la sencillez y rapidez de la biblioteca, se han encontrado una serie de problemas:

- Se hace referencia constantemente a una imagen de la que se extraen las subimágenes que se utilizan para dar forma a los widgets. Esta imagen no está entre los recursos que vienen con la biblioteca, pero una búsqueda por la red revela un post[3] en los foros de libRocket en el que un usuario plantea el problema y uno de los desarrolladores de la biblioteca responde aportando un enlace de descarga con una versión anterior de los tutoriales donde sí se encuentra dicha imagen.
- La documentación, aunque muy superior a la de las bibliotecas anteriores, es confusa, no está actualizada y dista de ser completa.
- En el tercer tutorial nos encontramos un problema cuya solución[4] implica realizar una breve modificación (sólo una instrucción) en el código de libRocket y recompilar el módulo RocketCore.
- En el mismo tutorial nos topamos con otro contratiempo: en lugar de verse la imágenes que hemos asociado a botón para expandir y reducir una fila de una tabla, se ven unos

[2] http://librocket.com/wiki/documentation/tutorials

[3] http://forums.librocket.com/viewtopic.php?f=2&t=782&p=1818&hilit=tutorial#p2078

[4] https://github.com/libRocket/libRocket/issues/113

cuadrados negros. No se ha encontrado información sobre el problema ni en la documentación, ni en los foros, ni en la red en general, por lo que se ha optado por preguntar en el foro oficial[5]. En el momento de escribir estas líneas, aún no se ha obtenido respuesta.

Este incovenientes podrían no suponer un impedimento para nuestro objetivo (de hecho algunos están resueltos); no obstante, las dificultades encontradas siguiendo tres tutoriales sencillos nos hacen pensar que nos encontraríamos con muchas más programando la GUI el editor, y el tiempo que habría que dedicarle a solventar dichas dificultades sería excesivo, teniendo en cuenta la calidad de la documentación y la cantidad de información en general. Estas razones hacen que nos decantemos por descartar también libRocket.

Impresiones finales sobre libRocket

libRocket se ha mostrado como una buena biblioteca, sólida y con cierta madurez, pero desactualizada e incompleta en su documentación. Después de ver lo sencilla que fue su integración con SDL2, existían esperanzas de que pudiera ser la elegida, pero como se comenta al final del epígrafe anterior, los problemas con los que nos hemos encontrado siguiendo ejemplos sencillos y guiados podrían ser un preludio de lo que nos esperaría construyendo el editor de niveles.

Otra desventaja es que cada vez que queramos añadir un nuevo widget a la interfaz, por sencillo que sea, tenemos que diseñarlo gráficamente; esto es, crearnos una imagen (en formato TGA) y asociarla al componente (concretamente a su *decorator*), ya que no tienen un aspecto por defecto. En los tutoriales no hemos tenido este problema porque hemos obtenido una imagen que incorpora todos los elementos gráficos que necesitábamos, pero con el editor de niveles no tendríamos la misma suerte.

Por otra parte, salta a la vista que es una biblioteca pensada para renderizar la interfaz dentro del juego (menú, inventario, número de vidas, tabla de puntuaciones...), por lo que no dispone de algunos de los componentes más útiles que harían falta en un editor, como un explorador de archivos para cargar los recursos, o barras de menú y herramientas.

No todo son inconvenientes: las sensaciones utilizando la biblioteca son agradables, es rápida y ligera, y a la mencionada compatibilidad con SDL2 habría que añadir que no hay que recompilar el código cada vez que se cambia la GUI, ya que se define en un archivo externo RML (*Rocket Markup Language*) & RCSS (*Rocket Cascading Style Sheet*) de la misma forma que ocurre en las web con HTML & CSS.

En definitiva, libRocket no va a ser utilizada para construir GUI del editor de niveles; sin embargo, ha sido un descubrimiento interesante, y se podría tener en consideración en futuros proyectos, sobretodo si se llegara a participar en el desarrollo de un juego con una carga elevada de interfaz gráfica de usuario *in-game*, algo que no ocurre en un juego de plataformas básico como el que nos ocupa, para cuya interfaz *in-game* SDL se muestra más que suficiente.

10.1.4. Conclusiones

Las bibliotecas probadas hasta este momento han sido descartadas principalmente por sus limitaciones y la calidad de la documentación, aunque sería injusto dejar esta afirmación sin

[5] http://forums.librocket.com/viewtopic.php?f=2&t=5240

matizar, ya que libRocket sin duda se encuentra muchos pasos por delante de Game Gui y SDL2-widgets.

Además de las mencionadas SDL2-widgets, Game Gui y libRocket, se han considerado otras herramientas que finalmente no han tenido su epígrafe propio en la memoria por haber sido descartadas antes de probarlas, a saber: MyGUI, Awesomium, Cegui, AntTweakBar, Guichan, GWEN, OtterUI, FLTK y Turbo Badger. El motivo de su exclusión no es ni mucho menos la certeza de que sean inferiores a las que se han probado, sino su similitud con éstas en características y diseño, lo que nos lleva a pensar que nos encontraremos con similares ventajas e inconvenientes. Esto unido a que el tiempo de que se dispone es limitado y a que el objetivo del PFC no es realizar una comparativa entre bibliotecas para el desarrollo de interfaces de usuario en C++, ha hecho que nos decidamos por pasar a una herramienta madura, robusta, con rendimiento nativo, de largo recorrido y ampliamente utilizada en proyectos de cualquier magnitud: el framework de desarrollo multiplataforma Qt.

10.2. Qt

Es razonable preguntarse por qué no se eligió Qt desde el principio, así que se dará respuesta a esta cuestión en los siguientes párrafos.

Desde que surgió la necesidad de usar una biblioteca para desarrollar GUIs se pensó en Qt, pero en contraposición con las bibliotecas ligeras de las que se ha hablado hasta ahora, Qt es una herramienta extensa y compleja. Su condición de framework y la gran cantidad de código de que se compone (código desconocido para nosotros) hacen además más difícil monitorizar el flujo del programa, y puede existir la sensación de que se pierde un poco el control total de lo que está ocurriendo. Esto no supondría un gran problema si fuera a desarrollarse un programa con GUI al uso que sólo utilizara el sistema de pintado en pantalla de Qt, pero dada la naturaleza de nuestro problema es especialmente relevante, pues implica lidiar con varios bucles *inputHandling-update-render* interactuando entre sí, por lo que integrar SDL en Qt se prevé una tarea compleja.

La extensión en contenido de Qt tiene asociada una extensión en tamaño. Realizando una comparación rápida, vemos que la versión instalada (la que se recomienda por defecto en la web para la versión 5.5 de Qt) ocupa más de 2 GB en disco, quedando muy lejos de los 27 MB de Game Gui, los 23 MB de libRocket o los 418 KB de SDL2-widgets. Esto afecta negativamente a la portabilidad del entorno de trabajo y a la velocidad de compilación, y supone el añadido de una enorme cantidad de características y código de los que no se va a hacer uso.

Estas fueron las razones que nos llevaron a explorar primero algunas bibliotecas sencillas. Tras haberlo hecho y habernos encontrado con los problemas explicados en la Sección 10.1, retornamos a la idea original de utilizar Qt.

Qt es un framework para desarrollar aplicaciones multiplataforma, con una amplia gama de plataformas soportadas entre las que se incluyen Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry o Sailfish OS.

Qt no es un lenguaje de programación por sí mismo; es un framework escrito en C++. Un preprocesador, el MOC (Meta-Object Compiler)[6] es usado para extender el lenguaje C++ con

[6]http://doc.qt.io/qt-5/moc.html

características como el mecanismo de señales y slots con el que se comunican los objetos de la interfaz. Antes de la compilación, el MOC parsea los archivos escritos en Qt-extended C++ y genera ficheros estándar compilables en C++. De esta forma el framework y las aplicaciones o bibliotecas que lo utilicen pueden compilarse con cualquier compilador estándar de C++ como Clang, GCC, ICC, MinGW o Microsoft Visual Studio, que es el que utilizaremos.

Qt se encuentra disponible bajo varias licencias, tanto comerciales como de código abierto. En el apartado de descargas de la web se nos realiza un cuestionario y según nuestros intereses y el fin para el que vayamos a utilizar el framework se nos recomienda una u otra. Tras indicar que es para uso académico y que estamos dispuestos cumplir las obligaciones de la licencia LGPL[7], se nos recomienda descargarnos la versión Qt Open Source.

El framework viene con su propio Entorno de Desarrollo Integrado (IDE), llamado Qt Creator. Una alternativa es hacer uso del Visual Studio Add-in, un añadido a Microsoft Visual Studio[8] que integra las herramientas de desarrollo de Qt en el IDE de Microsoft, permitiendo a los desarrolladores usar el entorno de desarrollo estándar sin tener que preocuparse por procesos de instalación y preparación del entorno relacionados con Qt. Nos decantamos por esta última opción, pues aunque el Qt Creator parece un buen IDE, ya estamos familiarizados con Microsoft Visual Studio. Además de la posibilidad de abstraernos del preprocesador de Qt, el add-in nos permite utilizar el Qt Designer, una herramienta para diseñar visualmente GUIs a partir de componentes de Qt.

10.2.1. Visual Studio Add-in

Tras instalar Qt y el Visual Studio Add-in 1.2 sin contratiempos, vemos que en el asistente de MVS para la creación de nuevos proyectos, se ha añadido un nuevo grupo de plantillas bajo el nombre de Qt5 Projects (Figura 10.6).

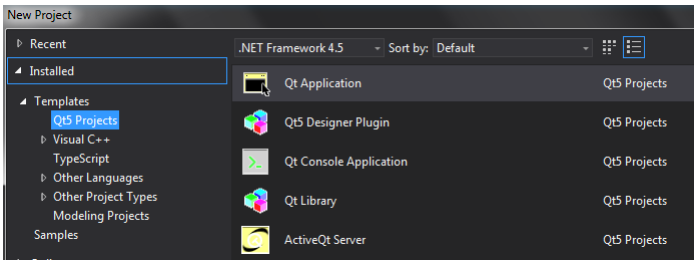


Figura 10.6 – Crear un Proyecto Qt5 con la ayuda del asistente.

Elegimos una Qt Application con las opciones por defecto y al compilar y ejecutar vemos la ventana que se aprecia en el centro de la Figura 10.7. En la misma imagen se puede apreciar en el Solution Explorer un archivo resaltado llamado «vsaddin.ui» (tendrá el nombre que se le haya dado al proyecto seguido de la extensión .ui). Los archivos con esta extensión son archivos de Qt Designer (Qt UI Files) que representan el árbol de widgets en formato XML.

Si hacemos doble click en el archivo «vsaddin.ui» se nos abrirá con el Qt Designer, permitiéndonos diseñar nuestra aplicación mediante una interfaz gráfica de usuario (Figura 10.8).

[7]http://www.qt.io/qt-licensing-terms/

[8]http://doc.qt.io/vs-addin/

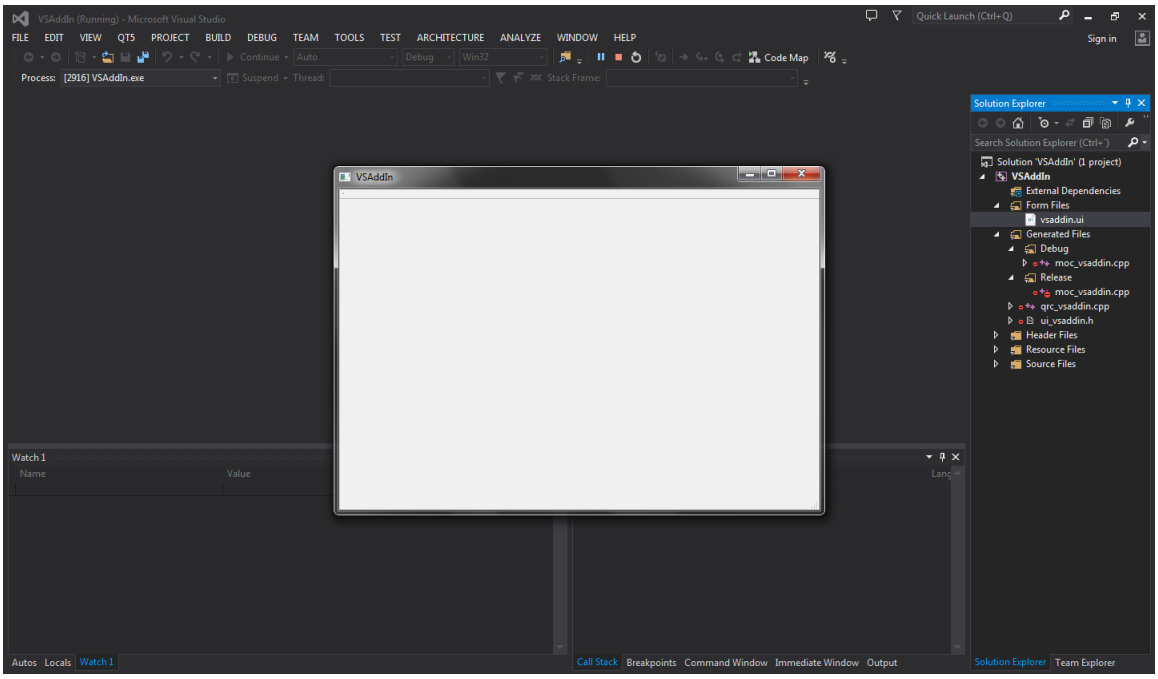


Figura 10.7 – Qt Application por defecto.

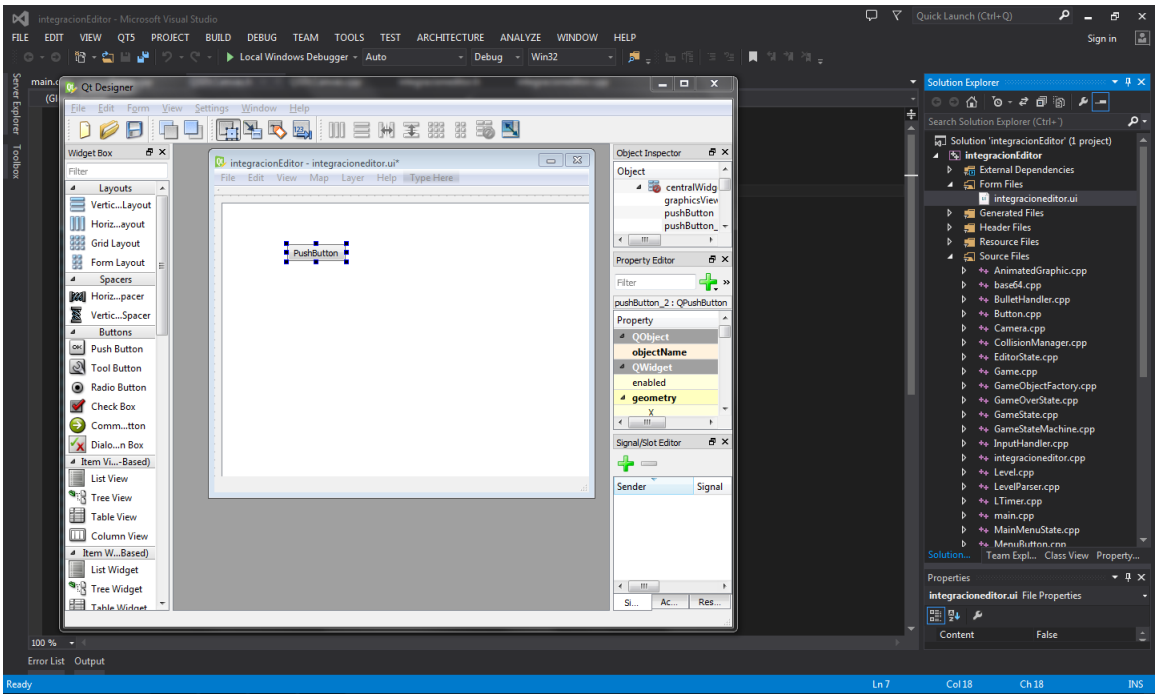


Figura 10.8 – Qt Designer integrado con Microsoft Visual Studio.

10.2.2. Usar Qt junto a SDL

Nuestro objetivo ahora es integrar nuestra biblioteca para la creación de juegos de plataformas dentro del entorno de Qt, para lo cual lo primero es estudiar cómo lograr que los sistemas de renderizado de SDL y Qt compartan la misma ventana, de manera que podamos disfrutar de la potencialidad de Qt para desarrollar GUIs a la vez que seguimos usando las clases y funciones de nuestro framework S2Dp (las cuales renderizan con SDL) para pintar en la ventana central del editor, donde se mostrará el escenario y los objetos situados en él y se podrá probar el nivel sin necesidad de generar otra ventana temporal.

Al ser un objetivo tan específico el que se persigue es difícil encontrar bibliografía de calidad y actualizada. En el repositorio de proyectos de SDL[9] hay un ejemplo programado por Sam Lantinga (el padre de SDL) de integración de SDL con GTK+ (*The GIMP Toolkit)*, un *toolkit* para desarrollar GUIs de características y popularidad similares a Qt. Analizando este ejemplo podría extraerse la forma de integrar SDL con Qt, o si esto no fuera posible nos podríamos plantear utilizar GTK+. La solución que utiliza Lantinga para que SDL use la misma ventana que GTK+ es la siguiente, extraída directamente del código:

```
/* Hack to get SDL to use GTK window */
{
  char SDL_windowhack[32];
  sprintf(SDL_windowhack, "SDL_WINDOWID=%ld", GDK_WINDOW_XWINDOW(mainwin->
     window));
  putenv(SDL_windowhack);
}
```

Desafortunadamente es un ejemplo para SDL1, como nos indica el uso de la variable de entorno SDL_WINDOWID, obsoleta en SDL2. Este arreglo del que Sam Lantinga hace uso es conocido como «SDL_WINDOWID hack». En la ya difunta wiki de SDL 1.2 hay un pequeño FAQ dedicado a usar SDL con herramientas GUI existentes[10], y se le dedican unas líneas a este *hack* en las que se explica que se hace a SDL usar una ventana existente en lugar de crear una nueva. Como vemos en el extracto de código, se basa en llamar a putenv("SDL_WINDOWID=XXX"), siendo XXX la ID de una ventana existente, que en nuestro caso sería la que está usando Qt. En SDL1 no existe la posibilidad de manejar varias ventanas, por lo que tener una variable de entorno que nos indique qué ventana está manejando SDL tiene sentido y no se presta a confusión; sin embargo, con SDL2 y su capacidad para manejar varias ventanas, SDL_WINDOWID desaparece dando paso a la clase SDL_Window, de la cual tendremos que crearnos una instancia por cada ventana que se esté manejando. Por tanto, el *hack* SDL_WINDOWID no nos es útil; no obstante, aún existe la posibilidad de crear una ventana a partir de una existente, por medio de la función que sigue:

```
SDL_Window* SDL_CreateWindowFrom(const void* data)
```

El parámetro data normalmente será un cast de una ventana nativa a void*. Haciendo uso de esta función es posible capturar la identidad de la ventana que está usando Qt y crear una SDL_Window a partir de ahí. SFML, una de las bibliotecas que consideramos usar al principio del PFC, cuenta en su web con un ejemplo de integración con Qt siguiendo este enfoque[36]. Dada su similitud con SDL (es común referirse a SFML como una «SDL orientada a objetos»), el ejemplo

[9] https://www.libsdl.org/projects/

[10] http://sdl.beuc.net/sdl.wiki/FAQ_GUI

resulta ser un buen primer contacto para usar conjuntamente SDL y Qt, aunque evidentemente habrá que modificarlo en profundidad para que llegue a funcionar. En el siguiente capítulo, dedicado al diseño y la implementación de S2PEditor, podrá verse la solución finalmente implementada para combinar SDL y Qt.

Capítulo 11

Diseño e Implementación

11.1. Interfaz Gráfica de Usuario

Antes de pasar a diseñar e implementar el código del editor, se creará un boceto de la interfaz gráfica de usuario (GUI), el cual podemos ver en la Figura 11.1. A partir de los casos de uso presentados en la Sección 9.1, podemos extraer las acciones que la interfaz deberá permitir. Estas acciones se encontrarán disponibles en la barra de menú y, al igual que se hizo con los casos de uso, se dividirán en tres grupos según afecten a todo el proyecto, al nivel actual o al juego final. Las acciones anidadas en cada entrada del menú serán las siguientes:

- **Project:** New Project, Open Project, Save Project, Export Project, Textures, Sounds, Exit.
- **Level:** New Level, Save as, Run Level, Stop Level, Backgrounds, Music.
- **Game:** Game Settings, Create Game.

Como es usual en este tipo de GUIs para programas de escritorio, estas acciones podrán realizarse también a través de los botones de la barra de tareas.

Para la gestión de objetos y tilesets se añadirán dos ventanas independientes, pues son acciones más complejas y, por la frecuencia con que accederemos a ellas, conviene tenerlas a mano en todo momento. Estas ventanas permitirán *grosso modo* lo siguiente:

Ventana de objetos Tendrá un botón (con su correspondiente icono) por cada uno de los doce tipos de objetos que podemos crear (ver epígrafe 6.3.4). Además incorporará una lista *drop-down* para seleccionar los objetos que hayamos creado, y dos botones para editarlos o borrarlos.

Ventana de tilesets Mediante un sistema de pestañas podremos seleccionar o borrar los distintos tilesets que hayamos añadido. La parte central de la ventana contendrá la imagen del tileset, desde la que podremos seleccionar las distintas tiles. Tendrá cuatro botones con las siguientes funciones: añadir un nuevo tileset, aumentar zoom, disminuir zoom e indicar si las tiles son sólidas o transparentes.

En la Figura 11.1 podemos ver todos los elementos mencionados rodeando la parte principal del editor: el mapa donde se editará el nivel actual. Este mapa se generará con SDL, al contrario que el resto de elementos de la interfaz, generados con Qt.

Cada uno de los botones de la barra de herramientas y de las ventanas de objetos y tilesets se han numerado; debajo del boceto se incluye una leyenda con la función de cada uno. Además puede verse en el extremo inferior derecho una tercera ventana dedicada a los niveles (oculta por defecto) que simplemente contendrá una lista con los niveles del proyecto, a partir de la cual podremos cargar o borrar cada uno de ellos.

En el Manual de Usuario (Capítulo A) se profundizará en la GUI final y sus opciones, mostrándose y explicándose las distintas ventanas de diálogo que aparecen al seleccionar cada una de las opciones.

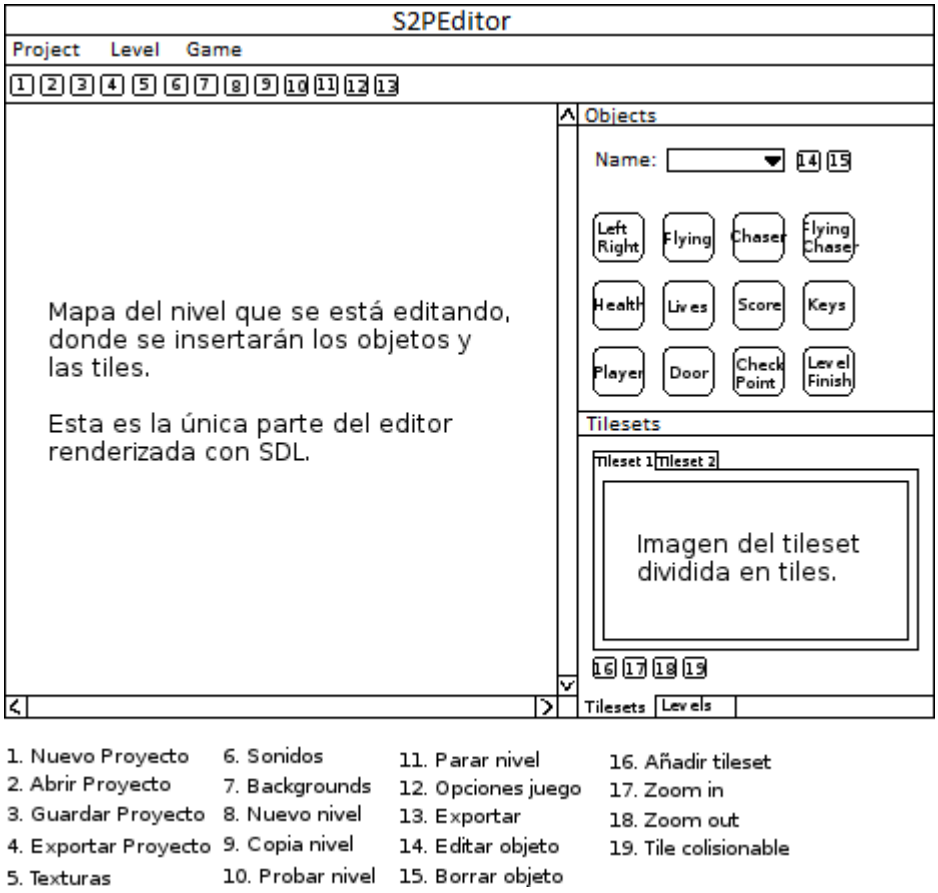


Figura 11.1 – Boceto de la GUI de S2PEditor.

11.2. Estructura de la solución adoptada

Crearemos un proyecto Qt con la plantilla *Qt Application*, como vimos en la Figura 10.6. Se creará una clase con el nombre que le hayamos dado al proyecto (en nuestro caso S2PEditor), derivada de QMainWindow, que nos proporciona un entorno para construir aplicaciones de interfaces de usuario. Qt dispone de QMainWindow y sus clases relacionadas[1] para la gestión de la ventana principal. QMainWindow tiene su propia disposición o *layout* a la que se pueden añadir QToolBars, QDockWidtets, una QMenuBar y una QStatusBar. En la Figura 11.2, extraída de la página oficial de Qt, puede apreciarse un esquema de esta disposición que resultará familiar a cualquiera que haya utilizado un software de escritorio con una GUI al uso..

El layout tiene una parte central que puede ser ocupada por cualquier tipo de widget; a este widget lo llamaremos widget central. El QMenuBar proporciona una barra de menú horizontal consistente en una lista de ítems con sus correspondientes acciones. Por su parte, la barra

[1] http://doc.qt.io/qt-5/widget-classes.html#main-window-and-related-classes

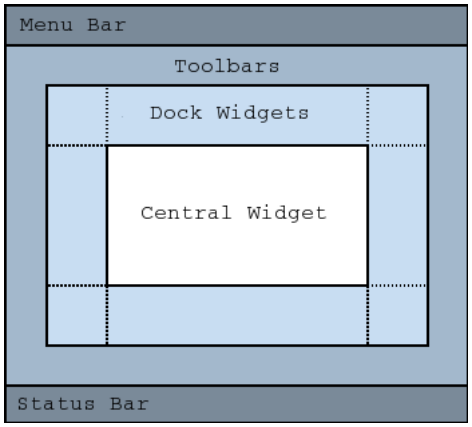


Figura 11.2 – Disposición de los componentes de QMainWindow.

de herramientas de tipo QToolBar nos da un panel móvil con botones que se puede situar a cualquiera de los cuatro lados del widget central, adoptando una disposición horizontal o vertical según corresponda.

Los *dock widgets*, de tipo QDockWidget, son widgets que pueden acoplarse a la ventana principal o flotar como ventanas independientes, además de anidarse con otros widgets del mismo tipo mediante un sistema de pestañas. También se les conoce como paletas de herramientas (*tool palettes*) o ventanas de utilidad (*utility windows*).

En algunos casos utilizaremos los tipos estándar proporcionados por Qt, pero en la mayoría de ocasiones la funcionalidad básica no será suficiente para satisfacer las necesidades de nuestro programa, por lo que se tomarán como base los tipos estándar de Qt para crear clases derivadas que modificarán y expandirán las características originales, a las que daremos el nombre de *custom widgets* o widgets personalizados.

Tomando como referencia la Sección 11.1 anterior, donde proponíamos un boceto de la interfaz, podemos ya asignar un tipo de datos a cada uno de los componentes más relevantes de nuestra GUI:

Ventana principal Widget personalizado S2PEditor, derivado de QMainWindow.

Barras de menú y herramientas La barra de menú y la de herramientas serán widgets de tipo QMenuBar y QToolBar, ambos básicos de Qt.

Widget central El widget central será de tipo QCanvasScrollArea, widget personalizado derivado de QScrollArea. Una *scroll area* se usa para mostrar parte del contenido de un widget hijo, pudiendo movernos por el resto de su superficie a través de barras de desplazamiento. Por tanto, QCanvasScrollArea únicamente es un contenedor, siendo mucho más importante su contenido: el widget personalizado QSDLCanvas, derivado de QWidget, clase base de todos los objetos de interfaz de usuario. En el QSDLCanvas, que utilizará las funciones de dibujado de SDL, será donde veamos, editemos y probemos el nivel actual.

Dock widgets las ventanas acoplables para gestionar los objetos, los tilesets y los niveles, tomarán la forma de los widgets personalizados QObjectsDock, QTilesetsDock y QLevelsDock, respectivamente. Todos ellos serán derivados de QDockWidget.

Ventanas de diálogo La mayoría de acciones accesibles desde el menú, barra de herramientas y *dock widgets*, generarán la aparición de un diálogo, una ventana que aparecerá sobre las demás y que generalmente se usa para tareas a corto plazo y comunicación con el usuario. Cada uno de ellos será un widget personalizado derivado de QDialog, clase básica de Qt para ventanas de diálogo.

Los elementos de la interfaz se comunicarán con el framework S2DP desarrollado en la primera parte del proyecto través de una clase a la que hemos llamado EditorController, puesto que actúa como el controlador que media entre la GUI y el framework. No hay restricciones para utilizar las clases del framework directamente desde los widgets, pero se ha preferido seguir este enfoque, de manera que si se quisiera realizar otra interfaz usando Qt u otra herramienta, la mayoría de los métodos de EditorController no requerirían ser modificados.

En la Figura 11.3 se muestra un esquema de la comunicación entre la GUI, el controlador y S2DP, con los nombres de las clases más relevantes que toman parte en el proceso.

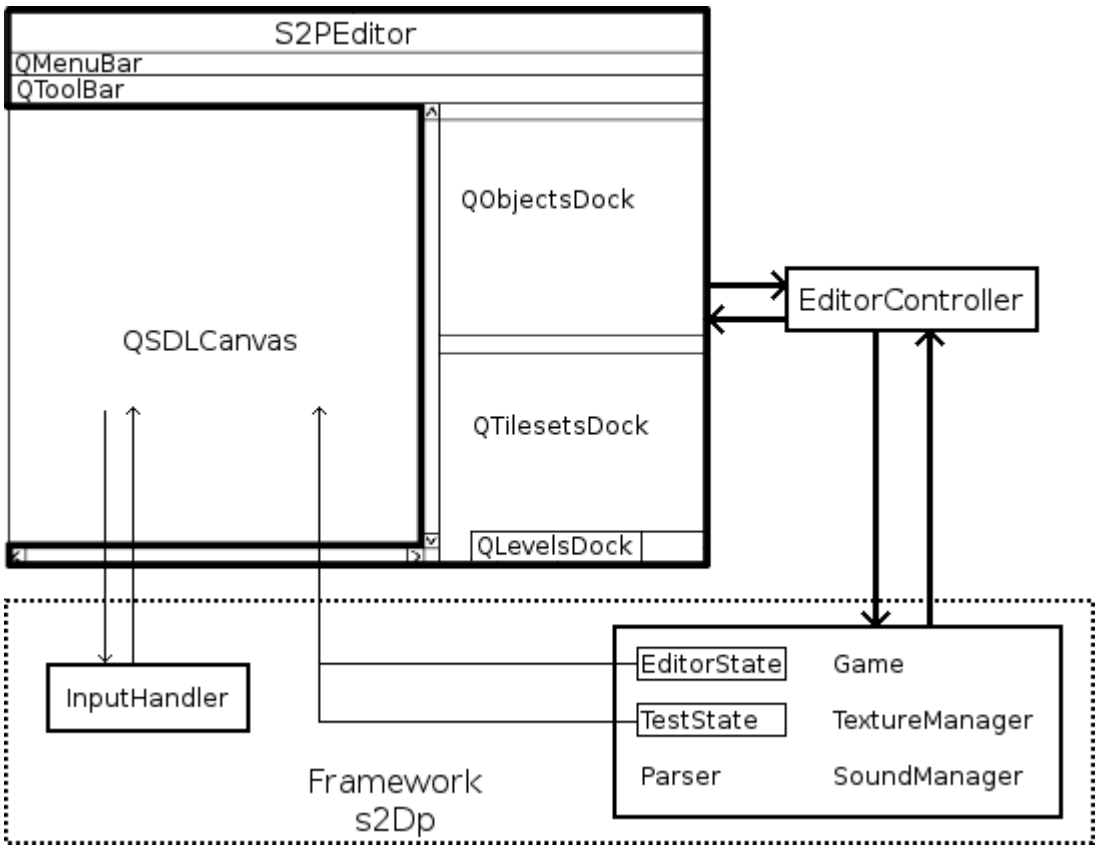


Figura 11.3 – Esquema del diseño de S2PEDITOR.

Las únicas clases del esquema que no se han presentado son EditorState y TestState. Estas clases son nuevos estados del juego (los estados se explican en la Sección 8.4) añadidos al framework S2DP, necesarios para cubrir las necesidades de un editor, distintas a las de un juego. El framework dibujará en QSDLCanvas de manera ininterrumpida, por lo que si se realiza alguna acción en la GUI que modifique el estado del nivel, ésta tendrá su reflejo inmediato en el área de edición.

En el esquema puede verse además que QSDLCanvas hace uso del gestor de eventos de S2DP (InputHandler) directamente, resultando en una de las escasas excepciones en que un componente

de la GUI accede al framework sin pasar por el controlador. La razón es que queremos mantener el controlador lo más independiente posible de la biblioteca GUI, y para realizar la gestión de entrada de teclado hay que hacer un mapeo entre los tipos de datos que identifican las teclas en Qt y en SDL, lo cual es una tarea específica de Qt que con otra biblioteca sería distinta.

En la siguiente tabla se enumeran todas las nuevas clases creadas para este segundo desarrollo del PFC; a las ya mencionadas se añaden el resto de *custom widgets* creados, en su mayoría diálogos. Para cada clase se indica su nombre, su clase base y una muy breve definición.

Clase	**Clase base**	**Definición**
S2PEditor	QMainWindow	Panel principal de la interfaz.
QSDLCanvas	QWidget	Área donde se ve el nivel que se está editando.
QLevelsDock	QDockWidget	Ventana acoplable para gestionar los niveles del proyecto.
QObjectsDock	QDockWidget	Ventana acoplable para gestionar los objetos del proyecto.
QTilesetsDock	QDockWidget	Ventana acoplable para gestionar los tilesets del proyecto.
QAddBckDialog	QDialog	Diálogo para añadir un background al nivel.
QAddSoundDialog	QDialog	Diálogo para añadir un sonido al proyecto.
QBBDialog	QDialog	Diálogo para especificar la hitbox de un sprite.
QBckDialog	QDialog	Diálogo que actúa como gestor de backgrounds.
QGameSettingsDialog	QDialog	Diálogo para configurar las opciones del juego final.
QNewLevelDialog	QDialog	Diálogo para añadir un nuevo nivel al proyecto.
QNewObjectDialog	QDialog	Diálogo para añadir un nuevo objeto al proyecto.
QNewProjectDialog	QDialog	Diálogo para crear un nuevo proyecto.
QNewSpriteDialog	QDialog	Diálogo para añadir una nueva imagen al proyecto.
QNewTilesetDialog	QDialog	Diálogo para añadir un nuevo tileset al proyecto.
QSoundsDialog	QDialog	Diálogo que actúa como gestor de sonidos.
QTexturesDialog	QDialog	Diálogo que actúa como gestor de texturas.
QBBLabel	QLabel	Etiqueta para indicar gráficamente la hitbox de un sprite.
QTilesetLabel	QLabel	Etiqueta con la imagen del tileset actual, a través de la que se eligen las tiles. Permite zoom.
QCanvasScrollArea	QScrollArea	Área que contiene el QSDLCanvas con el nivel actual, permitiendo desplazarse por el mapa.

Clase	Clase base	Definición
QTilesetScrollArea	QScrollArea	Área que contiene la QTilesetLabel con el tileset actual, permitiendo desplazarse por el mismo.
EditorState	GameState	Estado de S2DP en el que nos encontramos mientras editamos el nivel.
TestState	PlayState	Estado de S2DP al que se pasa al probar el nivel actual.

En el resto del capítulo se ampliará la información sobre las clases que se han considerado más importantes, que coinciden en gran parte con los widgets personalizados vistos en el esquema de la Figura 11.3.

11.3. Ventana principal

Como se ha comentado, la clase S2PEditor derivará de QMainWindow y será la ventana principal de la aplicación, de ahí que le hayamos dado el mismo nombre que al editor en sí. Tanto las distintas ventanas acoplables como el área de edición del nivel formarán parte de la ventana principal o, dicho de otra forma, tendrán a S2PEditor como su clase padre.

A la barra de menú (QMenuBar) se pueden añadir distintos ítems o entradas de tipo QMenu. Cada uno de estos ítems podrá tener asociada una serie de acciones, de tipo QAction. Además, esas mismas acciones podrán ser añadidas fácilmente a la barra de tareas. En la tabla que sigue podemos ver el nombre de cada QAction creada y del QMenu al que está asociada:

Entrada QMenu	QAction	Descripción
menu_Project	actionNewProject	Crear nuevo proyecto.
menu_Project	actionOpen	Abrir proyecto existente.
menu_Project	actionSaveProject	Guardar el proyecto actual.
menu_Project	actionExportProject	Exportar el proyecto actual.
menu_Project	actionTextures	Abrir gestor de texturas.
menu_Project	actionSounds	Abrir gestor de sonidos.
menu_Project	actionExit	Salir del programa.
menu_Level	actionAddLevel	Añadir un nivel al proyecto.
menu_Level	actionSaveAs	Guardar copia del nivel actual.
menu_Level	actionRun	Probar nivel actual.
menu_Level	actionStop	Parar nivel actual.
menu_Level	actionBackgrounds	Abrir gestor de backgrounds.
menu_Level	actionMusic	Indicar música del nivel.
menu_Game	actionGameSettings	Abrir opciones del juego.
menu_Game	actionCreateGame	Exportar juego final.
menu_Insert	actionSelectTile	Cambiar modo de inserción a «Tiles».
menu_Insert	actionSelectObject	Cambiar modo de inserción a «Objetos».

En programación GUI, al modificar un objeto, a menudo queremos que otro objeto sea notificado. De manera más general, queremos que objetos de cualquier tipo sean capaces de comunicarse con cualquier otro. Algunos *toolkits* antiguos utilizan *callbacks* (punteros a funciones) para lograr esto; sin embargo, Qt utiliza un sistema de señales y *slots* que se ha convertido en una de sus características más distintivas.

Una señal es emitida cuando ocurre un evento en particular, y un slot es una función a la que se llama como respuesta a una señal en concreto. Los widgets de Qt tienen tanto señales como slots predefinidos, a los que podemos añadir los nuestros propios creando widgets que derivados de los básicos.

Aunque existen varias maneras de hacerlo manualmente, nos centraremos ahora en una forma automática[2] de conectar señales y slots, siempre y cuando estos últimos sean nombrados adecuadamente. Si se desea que esto ocurra, debe declararse e implementarse un slot con un nombre que siga el siguiente convenio:

```
void on_<object name>_<signal name>(<signal parameters>);
```

Así, si queremos definir un método que se ejecute al activarse la acción para crear un nuevo proyecto, su nombre será el siguiente:

```
void on_actionNewProject_triggered();
```

, ya que actionNewProject es el nombre de la acción y triggered es la señal emitida cuando un usuario la activa. El siguiente paso sería escribir la definición o cuerpo del método, que en este caso particular se compondrá de las instrucciones necesarias para crear y abrir un nuevo proyecto. El proceso será el mismo para todas las QAction de la tabla previa, resultando en un nuevo método (que a su vez es un slot) añadido a S2PEditor por cada acción.

Para cargar el resto de elementos de la ventana principal (ventanas acoplables y área de edición del nivel), se han incluido los siguientes métodos privados:

```
bool initEditor();
void initObjectsDock();
void initTilesetsDock();
void initLevelsDock();
```

initEditor establece y carga el widget central y se encarga de las comprobaciones pertinentes para cargar el último proyecto utilizado, o crear uno nuevo si éste no se encontrara.

initObjectsDock crea una nueva instancia de la clase QObjectsDock y la añade a la ventana principal. initTilesetsDock e initLevelsDock hacen lo mismo con instancias de las clases TilesetsDock y LevelsDock, respectivamente.

11.4. Añadidos al framework s2Dp

Si queremos poder utilizar el framework s2Dp para editar niveles, se requieren una serie de añadidos. El más importante es el nuevo estado en el que nos encontraremos cuando estemos editando un nivel, así como otro estado para cuando lo estemos probando. Además, tendremos

[2]Ésta es una de las facilidades que aporta la clase QMetaObject, que contiene meta-información acerca de los objetos de Qt.

que aumentar la funcionalidad del parser, de manera de que ahora además de leer archivos pueda escribirlos. Empezaremos por esto último.

11.4.1. Parser

En la versión original del parser de S2Dp, sólo eran necesarios métodos de lectura para analizar sintácticamente los archivos XML con la información del juego y sus niveles, de lo que se encargaban Parser::parseLevel y Parser::parseGameFile, explicados en el apartado 8.6.2. Con la creación del editor, necesitamos añadir a la clase Parser métodos con la función opuesta: escribir los archivos XML que contendrán la información del juego.

Los niveles que el usuario cree con la GUI se escribirán en ficheros XML de niveles (ver epígrafe 8.6.1) mediante el método writeLevel, cuyo prototipo es el siguiente:

```
bool writeLevel(const Level &level, std::string name);
```

A partir de una instancia de la clase Level, se escribe un archivo con el nombre dado por el parámetro name, que incluirá la ruta de destino deseada.

Para escribir el archivo XML con la información global del juego (ver epígrafe 8.6.1) se utilizará el método writeGameFile:

```
tinyxml2::XMLError writeGameFile(const std::string settingsFile,
  const std::map<std::string, std::string> &textures,
  const std::vector<Tileset> &tilesets,
  const std::vector<std::pair<std::string, std::string>> &levels,
  const std::map<std::string, std::string> &sfxs,
  const std::map<std::string, std::string> &music);
```

El primer parámetro es el destino del archivo de salida, y los siguientes son referencias a estructuras de datos con la información requerida: texturas, tilesets, niveles, efectos de sonido y música.

Con esto ya cubrimos la escritura de los archivos del juego, pero necesitamos además métodos para guardar y cargar el estado del proyecto del usuario, de manera que al cerrar y abrir S2PEditor pueda continuar editando el juego por donde lo había dejado. Para escribir el estado del proyecto crearemos writeProjectFile:

```
bool writeProjectFile(const std::string fileName, const std::map<std::string,
    GameObjCfg> &objConfigs,
  const std::map<std::string, std::string> &projectTextures,
  const std::vector<Tileset> &tilesets,
  const std::vector<std::pair<std::string, std::string>> &levels,
  const std::map<std::string, std::string> &sfxs,
  const std::map<std::string, std::string> &music);
```

El archivo con la información del proyecto tendrá una estructura muy similar al global del juego. Esto se refleja en que cinco de los seis parámetros coinciden con los de writeGameFile, aunque en este caso se refieren a las texturas, tilesets, niveles, efectos de sonido y música añadidos al proyecto, que no tienen por qué coincidir con los del juego final. La principal diferencia es que en lugar de tener name con el nombre del archivo destino[3], tenemos objConfigs.

[3]El ficheros de proyecto tendrá un directorio y un nombre predeterminado, con extensión S2PE.

El parámetro objConfigs es un mapa asociativo con todas las *configuraciones de objeto* añadidas al proyecto. Llamaremos configuración de objeto (a la que nos referiremos en ocasiones como «objeto» por simplicidad) al conjunto de instancias de una clase derivada de GameObject con un comportamiento y apariencia concretos. Por ejemplo, el usuario puede elegir mediante la GUI añadir un enemigo de tipo LeftRight que tenga la sprite sheet de una araña, se mueva con una velocidad de tres píxeles por paso, tenga dos puntos de vida, etc., y darle el nombre «Spider1». Diríamos entonces que se ha añadido al proyecto la configuración de objeto «Spider1».

También necesitamos un método para cargar el estado de un proyecto guardado previamente, al que llamaremos parseProjectFile:

```
tinyxml2::XMLError parseProjectFile(const std::string file,
  std::vector<std::pair<std::string, std::string>> &editorLevels,
  std::map<std::string, std::string> &textures,
  std::vector<Tileset> &tilesets,
  std::map<std::string, GameObjCfg> &objConfigs,
  std::map<std::string, std::string> &sounds,
  std::map<std::string, std::string> &music);
```

Este método funciona de forma inversa al previamente comentado, analizando sintácticamente el archivo file y guardando su información en las estructuras de datos correspondientes de nuestro código, de ahí que sean parámetros de entrada/salida.

11.4.2. EditorState

Añadiremos un nuevo estado del juego a los previamente existentes en el framework S2DP. EditorState será el estado en el que se encuentre S2DP mientras se está editando el nivel; más concretamente, será el estado actual del autómata finito (FSM) atributo de Game (ver epígrafe 8.4).

Cierto es que a nivel semántico no puede decirse que sea un estado del *juego*, pues al contrario que PlayState, StartState, PauseState, GameOverState y EndState, no es un estado por el que pasemos en el flujo normal de una partida. Sin embargo, nos conviene implementarlo como un estado más para aprovechar el resto de funciones y clases del framework sin modificar el game loop.

Al igual que ocurría con PlayState, EditorState dispone de un atributo de tipo Level y una función loadLevel para rellenarlo a partir de un archivo XML. No obstante, si en PlayState el nivel iba cambiando en base a cómo se actualizaran los objetos, en EditorState el nivel será estático y sólo cambiará a partir de las modificaciones que realice el usuario con el editor. Además, en PlayState sólo cambian los objetos del nivel, mientras que en EditorState pueden cambiar además las tiles, los backgrounds y la música.

Como en todas las clases derivadas de GameState, tenemos que implementar los métodos update y render, donde podemos empezar a ver diferencias con PlayState. Empezaremos comentando el update:

```
void EditorState::update()
{
  if (TheInputHandler::Instance().getMouseButtonState(LEFT))
  {
    handlePlacement(x_position, y_position);
```

```
  }

  if (TheInputHandler::Instance().getMouseButtonState(RIGHT))
  {
    deleteUnderPosition(x_position, y_position);
  }
}
```

Como puede verse, no se actualiza la capa de objetos, puesto que queremos que se queden en el mismo sitio en que el usuario los ha colocado con el editor. Se comprueba si se ha pulsado el botón izquierdo del ratón para, de ser así, llamar al método privado handlePlacement pasándole la posición del ratón. Este método se encarga de insertar el objeto o tile que se haya seleccionado en la GUI del editor, en la posición adecuada.

Por último se comprueba si se ha pulsado el botón derecho del ratón, lo que indica que el usuario quiere borrar un objeto o tile, de lo que se encarga el método privado deleteUnderPosition.

Algo importante a destacar aquí es que para capturar el ratón se está usando InputHandler, el gestor de entrada y salida de S2DP, que a su vez hace uso del sistema de entrada/salida de SDL. No ocurre así con el teclado, pues al contrario que con el ratón, las pulsaciones de las teclas no son capturadas por SDL al usar una ventana generada por un widget de Qt. Esto se verá más claramente en la Sección 11.6, donde se explica la implementación de la clase QSDLCanvas.

El método EditorState::render se define como sigue:

```
void EditorState::render()
{
  m_level->render();
  drawGrid();
  showSelected();
}
```

Primero se llama al método Level::render para dibujar las capas de tiles y objetos. Estos últimos, al no ser actualizados, dibujarán siempre el primer frame de la animación por defecto de su hoja de sprites.

La función de los métodos drawGrid y showSelected es dar feedback visual al usuario. drawGrid se encarga de dibujar una rejilla con tamaño de celda igual al de las tiles del nivel. showSelected dibuja en transparente el objeto o tile que el usuario tenga seleccionado (si tuviera alguno), «persiguiendo» el puntero del ratón en el editor para indicar dónde y cómo quedará una vez colocado en el nivel. Si es un objeto, se dibujarán alrededor dos recuadros con información sobre el frame y la hitbox del mismo.

Además de update y render, la clase EditorState tiene una gran cantidad de métodos, tanto privados para realizar tareas auxiliares, como públicos, dedicados en su mayoría a modificar y conocer el estado de los elementos del nivel. Estos últimos serán utilizados por el controlador EditorController para obtener información del nivel y para editarlo a partir de las acciones del usuario en la GUI.

Así, cuando utilizamos la herramienta S2PEDITOR, la parte de la ventana correspondiente al mapa del nivel que editamos y vemos no es más que el framework S2DP actualizando y dibujando el estado EditorState dentro del widget QSDLCanvas (ver Figura 11.3).

Habría que añadir una excepción al párrafo anterior, y esta es cuando pulsamos el botón para probar el nivel que estamos editando. En ese caso pasamos a otro nuevo estado llamado TestState.

No nos detendremos a explicarlo, ya que es igual a PlayState (como no podía ser de otra forma si queremos probar el juego), con la salvedad de que al llegar al final del nivel o perder todas las vidas, el nivel comienza desde el principio, en lugar de pasar al siguiente nivel o a la pantalla de Game Over como ocurriría en el juego normal.

11.5. Clase EditorController

Esta clase actúa como intermediaria entre los widgets de la GUI y el framework S2Dp. Las clases del framework a las que accederá EditorController, como se muestra en la Figura 11.3, son las siguientes:

Game Es la clase principal del framework y, como tal, una de las que más utilizará el controlador. Muchas de las variables editables del juego (valores iniciales y finales para salud, vidas y puntuación, tilesets, fichero con los niveles del juego, iconos en el HUD, etc.) son atributos de Game, por lo que desde EditorControler se llamará con frecuencia a sus getters y setters para, respectivamente, mostrar esa información en la interfaz o modificarla a través de ella, respectivamente.
Además, puesto que en el área de edición del nivel se estará mostrando la salida del framework (ya sea en el estado EditorState o TestState), el controlador también tendrá que hacer usos de los métodos de Game para controlar la inicialización, el game loop y su finalización.

TextureManager El controlador mantendrá actualizado el gestor de texturas de S2Dp, añadiendo o borrando las texturas que el usuario añada o elimine del proyecto.

SoundManager De igual forma que con TextureManager, el gestor de sonidos de S2Dp deberá mantenerse actualizado a través del controlador.

Parser Como se comentó en el apartado 11.4.1, se acudirá al parser para guardar y cargar el estado del proyecto y los niveles.

EditorState Junto con Game, será la clase más utilizada por el controlador. EditorController hará uso de EditorState para obtener información del nivel y para modificarlo según las acciones indicadas en la interfaz del editor.

TestState Esta clase se añadió al framework como una variación de PlayState que nos sirve para probar un nivel.

A continuación veremos mediante una traza de código la interacción entre las clases Qt de la interfaz, el controlador y el framework. Como ejemplo supondremos que el usuario quiere probar el nivel que está editando. Como se indicó en el apartado 11.3, la QAction destinada a probar el juego se llama actionRun. Cuando el usuario lleve a cabo dicha acción, ya sea a través de la barra de herramientas o de menú, se disparará la señal triggered de la acción actionRun. Siguiendo el convenio de nomenclatura de Qt, hemos declarado un slot privado llamado on_-actionRun_triggered, para que se ejecute automáticamente cuando se dispare esa señal. El código del cuerpo del método es el siguiente:

```
void S2PEditor::on_actionRun_triggered()
{
  disableActions();
  EditorController::Instance().runLevel(m_levelSource.toStdString());
  ui.actionStop->setDisabled(false);
}
```

La primera instrucción es una llamada a disableActions, un método privado que hemos creado para deshabilitar todas las acciones del editor mientras se está probando el juego, excepto la acción de finalizar la prueba y volver al estado de edición (actionStop).

La segunda instrucción es la que más nos interesa, pues es en la que se hace uso del controlador. Se ha implementado como un singleton, por lo que en esa instrucción se está llamando desde la clase S2PEditor al método EditorController::runLevel. El parámetro que se le pasa es el nombre del nivel; como Qt tiene su propio tipo para ristras, utilizamos su método toStdString para convertirla a std::string, ya que no queremos que el controlador tenga ninguna dependencia de Qt. Veamos la definición del método runLevel:

```
bool EditorController::runLevel(string levelPath)
{
3   if (m_Editor != nullptr)
    {
      m_Editor->saveLevel(m_testLevel);
6     TheGame::Instance().getStateMachine()->pushState(new TestState(
          m_testLevel));
      m_Editor = nullptr;
      return true;
9   }
    return false;
}
```

Aquí puede verse claramente cómo EditorController hace uso del framework S2Dp. La variable m_Editor es un atributo de tipo EditorState*, que apuntará al estado actual de la FSM siempre que estemos editando, y será nuestra manera de acceder a los métodos de EditorState.

Comenzamos comprobando que m_Editor no es nulo para confirmar que se está editando. A continuación se utiliza el método EditorState::saveLevel para guardar el nivel que vamos a probar (m_testLevel contiene una ruta predefinida donde escribir el archivo).

La línea más importante es la número 6, donde llamamos al método pushState de la FSM de Game, pasándole un nuevo TestState creado a partir del archivo guardado previamente (ver Sección 8.4). Por último ponemos m_Editor a nullptr para indicar que ya no estamos editando.

Con esto, en el área donde se estaba editando el nivel, éste pasará a ejecutarse hasta que el usuario decida pararlo. Como se ha visto, desde la interfaz (en concreto desde la clase S2PEditor) simplemente se llama a la función runLevel del controlador, y es éste el que se encarga de lidiar con los detalles del framework S2Dp.

11.6. SDL y Qt: clase QSDLCanvas

Nuestro objetivo es integrar SDL dentro de una interfaz de Qt. La forma estándar de añadir una nueva característica a una interfaz de Qt es crearse un widget personalizado, que derive

de la clase base QWidget. Como vimos en la Figura 11.3, la nueva clase que nos crearemos se llamará QSDLCanvas.

Para tener una idea general, veamos la declaración de la clase y sus miembros:

```
class QSDLCanvas : public QWidget
{
  Q_OBJECT

public:
  QSDLCanvas(QWidget* Parent);
  ~QSDLCanvas();

  bool initEditor(std::string levelName, int width = -1, int height = -1, int
      tileSize = -1);

private:
  virtual QPaintEngine* paintEngine() const;
  virtual void keyPressEvent(QKeyEvent *k);
  virtual void keyReleaseEvent(QKeyEvent *k);

  QTimer m_UpdateTimer;
  bool m_Initialized;

private slots:
  virtual void onUpdate();
};
```

Como métodos públicos, además del constructor y el destructor, tenemos initEditor, cuya función es inicializar el framework S2Dp (y por tanto SDL) en la ventana del widget.

En los privados, se sobrescribirá paintEngine(), donde le indicaremos a Qt que no vamos a utilizar sus sistemas de pintado en pantalla, y los métodos keyPressEvent() y keyReleaseEvent(), para indicar al gestor de eventos de S2Dp que se ha pulsado o levantado una tecla (más adelante se verá por qué es necesario esto. Además declararemos un método onUpdate() que será el encargado de actualizar el estado del nivel que se está editando (o probando) y dibujar su contenido.

Qt no proporciona un *idle event* (que sería llamado cada vez que la cola de eventos esté vacía), por lo que tendremos que actualizar el widget manualmente. Una forma de hacer esto es lanzar un temporizador y conectarlo a una función que actualice el widget con la frecuencia especificada. El temporizador será m_UpdateTimer y la función para actualizar el widget será onUpdate.

Vemos el método onUpdate() bajo el especificador de acceso private slots, palabra clave de Qt para indicar que los métodos son eventos asignables a las señales. Esto nos permitirá asignar el slot onUpdate() a la señal timeout() del temporizador m_UpdateTimer, que se activa cuando éste llega a cero.

A continuación se describirán con mayor detalle los métodos explicados, indagando en su código.

11.6.1. Constructor

En el constructor se realizan algunas acciones necesarias para poder ver la salida de S2Dp en la ventana del *custom widget* QSDLCanvas.

```
QSDLCanvas::QSDLCanvas(QWidget* Parent) :
QWidget(Parent),
m_Initialized(false)
{
  setAttribute(Qt::WA_PaintOnScreen);
  setFocusPolicy(Qt::StrongFocus);
  m_UpdateTimer.setInterval(0);
}
```

Primero, se activa el atributo Qt::WA_PaintOnScreen para indicarle a Qt que no utilice su propio sistema de pintado, ya que usaremos el de SDL. A continuación establecemos la política Qt::StrongFocus, para permitir que el widget reciba los eventos de teclado de Qt, pues la política por defecto es Qt::NoFocus.

Por último, establecemos el intervalo de m_UpdateTimer a cero. Este valor hará que el timer dispare una actualización cada vez que no haya otros eventos para procesar, lo cual es exactamente lo que haría un *idle event*.

11.6.2. Iniciando el editor

El método initEditor será el que nos permita iniciar el framework S2Dp «dentro» de una instancia de QSDLCanvas. El prototipo es el siguiente:

```
bool initEditor(std::string levelName, int width = -1, int height = -1, int
    tileSize = -1);
```

El parámetro levelName será el nombre del nivel que se va a cargar en el editor, es decir, en el estado EditorState del framework. Pueden darse dos escenarios:

- Si no se indican los parámetros opcionales, levelName contendrá la ruta del nivel que se quiere cargar.
- Si se indican los parámetros opcionales, levelName contendrá la ruta del nuevo nivel que se creará y cargará, cuyo ancho, alto y tamaño de tile vendrá dado por width, height y tileSize.

Veamos la definición del método:

```
bool QSDLCanvas::initEditor(std::string levelName, int width, int height, int
    tileSize)
{
  if (!m_Initialized)
  {
    connect(&m_UpdateTimer, SIGNAL(timeout()), this, SLOT(onUpdate()));
    if (EditorController::Instance().init((void*)winId(), levelName, width,
        height, tileSize))
    {
      m_UpdateTimer.start();
      m_Initialized = true;
      return true;
    }
  }
  return false;
}
```

Después de comprobar si el editor ya se ha inicializado, en la línea 5 conectamos la señal timeout del timer m_UpdateTimer con el slot onUpdate() de QSDLCanvas. Esto hará que cuando la cola de eventos esté vacía se llame a onUpdate(), desde donde actualizaremos y dibujaremos el contenido de la ventana del widget a través de S2DP y SDL.

A continuación (línea 6) llamamos al método init de la clase EditorController, que tiene el siguiente prototipo:

```
bool init(void* winHandle, const std::string& levelName, int width = -1, int
    height = -1, int tileSize = -1);
```

Como puede verse, tiene los mismos parámetros que el initEditor que nos ocupa, con el añadido del parámetro winHandle, que contendrá el identificador de la ventana donde dibujará SDL, que en este caso será la del widget. Para obtener el identificador de la ventana del widget usamos la función QWidget::winID (recordemos que QSDLCanvas es derivada de QWidget).

El controlador se encargará de llamar a los métodos de inicialización del framework, en concreto a los de la clase Game del mismo. En el método Game::init, en lugar de crearse una nueva ventana SDL con la función SDL_CreateWindow (ver Sección 8.8), se utilizará la función SDL_CreateWindowFrom para crearla a partir del identificador de ventana de la instancia de QSDLCanvas, como se explicó en el epígrafe 10.2.2.

Si no hay problemas con la inicialización, iniciamos el timer (línea 8), ponemos m_Initialized a true (línae 9) y terminamos con éxito devolviendo true.

Si el editor ya se había inicializado o EditorController::init falla, además de devolver false se mostrarán distintos mensajes de error, que no se han mostrado aquí por ahorrar espacio.

11.6.3. Actualización y dibujado

El método onUpdate se llamará cada vez que la cola de eventos esté vacía, y será donde llamaremos a las funciones de actualización y dibujado del framework.

```
void QSDLCanvas::onUpdate()
{
  EditorController::Instance().update();
}
```

En el cuerpo de onUpdate simplemente se llamará al método EditorController::update, que se encargará de actualizar y dibujar en la ventana del widget a través de S2DP. Concretamente, realizará una llamada al método Game::iterate, que como se comentó en la Sección 8.13 lleva a cabo una iteración del game loop teniendo en cuenta la gestión del tiempo.

Por parte del método sobrescrito paintEngine(), no hay mucho que explicar. Simplemente hacemos que devuelva el valor nulo como sistema de renderizado, lo cual funciona en combinación con el atributo WA_PaintOnScreen (que activamos en el constructor) para decirle a Qt que no vamos a utilizar ninguno de sus sistemas de renderizado.

```
QPaintEngine* QSDLCanvas::paintEngine() const
{
  return nullptr;
}
```

11.6.4. Eventos de entrada

Al crear la ventana SDL con SDL_CreateWindowFrom a partir de una existente creada con Qt, SDL no recibe los eventos de entrada de teclado. Esto no nos afecta para la implementación de la GUI del editor de niveles, pues para ello nos valdremos de los widgets y eventos de entrada de Qt; sin embargo, cuando queramos probar el nivel, nuestro avatar no responderá a los controles, ya que S2DP realiza toda la gestión de eventos de entrada a través de la clase InputHandler, que está basado en los eventos de entrada de SDL.

Es aquí donde surge la necesidad de sobrescribir los métodos QWidget::keypressEvent y QWidget::keyreleaseEvent, para informar al gestor de eventos de S2DP de cuándo se ha pulsado o soltado una tecla. Antes de sobrescribirlos realizaremos unas modificaciones en la clase InputHandler.

Hasta ahora, hemos sabido el estado del teclado consultando el atributo InputHandler::m_keystates (ver Sección 8.9). No podemos modificar este dato miembro ya que es de tipo const* Uint8, debido a que es el tipo de la función SDL_GetKeyboardState que llamamos cada vez que se pulsa o se suelta una tecla para actualizar el estado de m_keystates. Añadiremos por tanto a InputHandler un nuevo atributo de tipo std::map que represente el estado del teclado y podamos modificar desde nuestra clase QSDLCanvas:

```
std::map<SDL_Scancode, bool> m_QKeyboardState;
```

La clave es de tipo SDL_Scancode y el valor un booleano que nos indicará el estado de la tecla (verdadero si está pulsada y falso en caso contrario). De esta forma será sencillo indicar que una tecla concreta se ha pulsado o soltado, lo cual haremos por medio del nuevo método que añadiremos a la clase InputHandler:

```
void addQKeyState(SDL_Scancode key, bool state);
```

De esta forma, se añadirá al map una nueva pareja clave-valor cada vez que pulsemos una tecla por primera vez, y se actualizará cuando la soltemos o la pulsemos otra vez.

Por último, hay que realizar un breve cambio en la función InputHandler::isKeyDown para que consulte el estado de nuestro «nuevo teclado», quedando así:

```
bool InputHandler::isKeyDown(SDL_Scancode key)
{
  return m_QKeyboardState[key];
}
```

No es necesario rellenar m_QkeyboardState con todas las posibles claves desde el principio, pues si en un std::map intentamos acceder al valor asociado a una clave que no se ha introducido (o lo que es lo mismo, comprobar el estado de una tecla que no se ha pulsado nunca) devolverá falso, que es el comportamiento que deseamos.

Volvemos a nuestro widget de Qt personalizado y a los métodos sobrescritos keyPressEvent y keyReleaseEvent, que indicarán al InputHandler la tecla que se acaba de pulsar o soltar mediante la función addQKeyState. Como es de esperar, los códigos de Qt y SDL para las teclas no coinciden, por lo que hay que realizar un mapeo de la primera a la segunda. Recordemos la cabecera de la función keyPressEvent (la de keyReleaseEvent es igual salvo por el nombre del método):

```
virtual void keyPressEvent(QKeyEvent *k);
```

La información de la que disponemos es el argumento k de tipo QKeyEvent*, a partir del cual tenemos que obtener el SDL_Scancode. Nuestro objetivo es obtener un char* con un valor válido que pasarle a la función SDL_GetScancodeFromName, que a partir de una ristra con el nombre común de la tecla nos devuelve su SDL_Scancode, que al fin y al cabo es un entero al tratarse de un enum. En la siguiente tabla podemos ver algunos ejemplos:

Instrucción	SDL_Scancode devuelto	Entero equivalente
SDL_GetScancodeFromName("1")	SDL_SCANCODE_1	30
SDL_GetScancodeFromName("a")	SDL_SCANCODE_A	4
SDL_GetScancodeFromName("space")	SDL_SCANCODE_SPACE	44

Para obtener un nombre que será compatible con SDL_GetScancodeFromName en prácticamente todas las ocasiones, haremos uso de la siguiente función de Qt:

```
QString QKeySequence::toString(SequenceFormat format = PortableText) const
```

Como vemos, es una función de la clase QKeySequence y devuelve una ristra de tipo QString. Con la siguiente instrucción obtendremos una QString con el valor que buscamos a partir de k, el argumento de tipo QKeyEvent*:

```
QString qs = QKeySequence(k->key()).toString();
```

No se puede hacer un cast a QkeySequence desde el tipo QKeyEvent, pero sí desde el código de la letra, que es lo que nos devuelve la función QKeyEvent::key. Lo siguiente es obtener a partir de la variable qs (de tipo QString) un char* que pasarle a SDL_GetScancodeFromName; afortunadamente la clase QString tiene un método llamado toStdString que nos devuelve una std::string con la ristra, lo que en combinación con string::c_str() nos da el char* que queríamos:

```
SDL_Scancode code = SDL_GetScancodeFromName(qs.toStdString().c_str());
```

Sólo quedaría informar al gestor de eventos de S2DP de que se ha pulsado la tecla, llamando al método recién explicado InputHandler::addQKeyState. La reimplementación de keyPressEvent quedaría por tanto así:

```
void QSDLCanvas::keyPressEvent(QKeyEvent *k)
{
  QString qs = QKeySequence(k->key()).toString();
  SDL_Scancode code = SDL_GetScancodeFromName(qs.toStdString().c_str());
  TheInputHandler::Instance().addQKeyState(code, true);
}
```

El otro evento de entrada sobrescrito, keyReleaseEvent, es exactamente igual, pero indicando que la tecla se ha soltado (false) en lugar de pulsado (true):

```
void QSDLCanvas::keyReleaseEvent(QKeyEvent *k)
{
  QString qs = QKeySequence(k->key()).toString();
  SDL_Scancode code = SDL_GetScancodeFromName(qs.toStdString().c_str());
  TheInputHandler::Instance().addQKeyState(code, false);
}
```

Con esto, a partir del sistema de entrada de Qt se mantendrá actualizado el estado del teclado del InputHandler, con lo que habremos conseguido que nuestro personaje responda a los controles.

Antes se especificó que de esta forma obtendríamos un mapeo correcto entre Qt y SDL en la mayoría de ocasiones. Para algunos caracteres no alfanuméricos no ocurre así, ya que el

nombre que le da Qt a la tecla no es el que espera SDL_GetScancodeFromName, con lo cual en lugar de recibir el SDL_Scancode que queremos, recibiremos SDL_SCANCODE_UNKNOWN. En estos casos tendremos que generar el código de SDL equivalente al de Qt de forma manual. De las teclas que se utilizan para controlar el personaje, esto sólo ocurre con SHIFT izquierdo, la tecla para correr. Esto nos obliga a modificar el método keyPressEvent para capturar este caso particular:

```
void QSDLCanvas::keyPressEvent(QKeyEvent *k)
{
  switch (k->key())
  {
  case Qt::Key_Shift:
    TheInputHandler::Instance().addQKeyState(SDL_SCANCODE_LSHIFT, true);
    break;
  default:
    QString qs = QKeySequence(k->key()).toString();
    SDL_Scancode code = SDL_GetScancodeFromName(qs.toStdString().c_str());
    TheInputHandler::Instance().addQKeyState(code, true);
  }
}
```

Lo mismo haremos con keyReleaseEvent.

11.6.5. Integración en la ventana principal

Ya tenemos nuestro custom widget QSDLCanvas creado, pero falta integrarlo en la ventana principal. Como se explicó en la Sección 11.3, nuestra ventana principal es de la clase S2PEditor, derivada de QMainWindow.

Para poder desplazarnos por el nivel que estamos editando, meteremos el widget QSDLCanvas dentro de un área de desplazamiento. Debido a que en QSDLCanvas no usamos el pintado de Qt, la clase QScrollArea base nos permite desplazarnos por el widget con las scrollbars, por lo que hemos tenido que programarlas manualmente, dando como resultado el otro custom widget: QCanvasScrollArea, derivado de QScrollArea. No indagaremos en la implementación de este widget; basta saber que será formalmente el widget central de la ventana (ver Figura 11.2), aunque lo que se muestre sea el widget QSDLCanvas que contiene.

Añadiremos por tanto dos atributos a la ventana principal S2PEditor, uno de tipo QSDLCanvas* y otro de tipo QCanvasScrollArea*:

```
class S2PEditor : public QMainWindow
{
...
private:
  QSDLCanvas* m_SDLCanvas;
  QCanvasScrollArea * m_canvasArea;
...
}
```

En el constructor de S2PEditor indicaremos que m_canvasArea es su widget central. Además, habrá que establecer m_SDLCanvas como el widget contenido en m_canvasArea, e inicializar el framework S2DP para que empiece a funcionar el editor. El siguiente fragmento de código realiza estas tareas:

```
m_canvasArea = new QCanvasScrollArea(this);
setCentralWidget(m_canvasArea);

m_SDLCanvas = new QSDLCanvas(this);
m_canvasArea->setWidget(m_SDLCanvas);
m_SDLCanvas->initEditor(levelName);
```

En la primera línea inicializamos m_canvasArea, haciendo que apunte a una nueva instancia de tipo QCanvasScrollArea. El parámetro this entre paréntesis indica que el padre será el objeto actual, es decir, la ventana principal S2PEditor.

A continuación, en la línea 2, establecemos m_canvasArea como el widget central de la ventana.

En la línea 3 se inicializa m_SDLCanvas y en la línea 4 se establece como el widget contenido en m_canvasArea, lo que nos permitirá movernos por la superficie del nivel con las scrollbars.

Por último llamamos al método initEditor de m_SDLCanvas con el nivel correspondiente, completando el proceso para poder ver en el widget la salida del framework S2DP, generada con SDL.

En la Figura 11.4 se muestra el resultado, donde vemos a la clase QSDLCanvas (área central donde se ve el mapa del nivel que se está editando) funcionando conjuntamente con el resto de widgets de Qt. Éste es el aspecto que tendrá la interfaz gráfica de usuario de la herramienta S2PEDITOR.

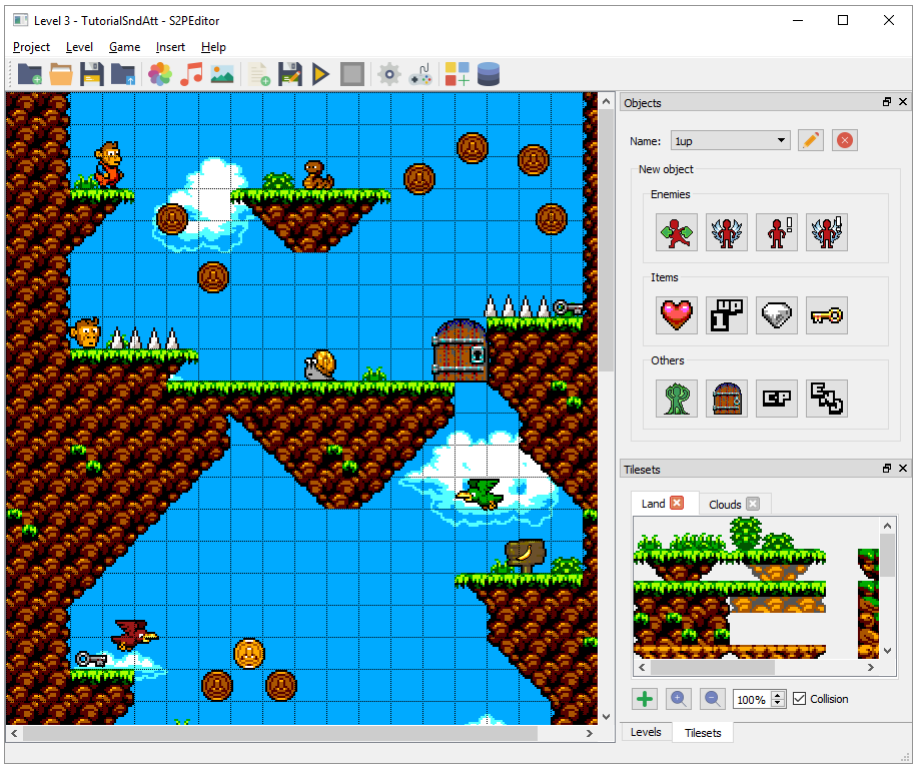


Figura 11.4 – GUI de S2PEditor.

Capítulo 12

Desarrollo de un juego con S2PEditor

Este capítulo es un tutorial en el que se comentará de principio a fin, por medio de un ejemplo sencillo, cómo desarrollar un juego utilizando la herramienta S2PEditor. El juego será independiente del editor, es decir, se generará en un directorio autosuficiente y podrá ejecutarse en sistemas que no tengan S2PEditor instalado.

Se comenzará creando un nuevo proyecto al cual se le irán añadiendo tres niveles con los que se pretende dar a conocer una muestra de las posibilidades que ofrece el editor. Una vez se haya finalizado la edición de los niveles, se procederá a generar el juego, estableciendo previamente una serie de parámetros obligatorios como su resolución de pantalla, el orden de los niveles que lo componen y otras opciones que veremos más adelante.

Para seguir el tutorial sin problema, es importante haber leído la Sección 6.3 del análisis de s2Dp, donde se describen el alcance y las características de los juegos que podemos crear con el editor. Además se hará referencia en alguna ocasión al «Manual de usuario», en especial a la Figura A.1 que muestra una visión general con los componentes de la interfaz.

Los recursos a los que se hace referencia en el tutorial se encuentran en la carpeta Recursos, disponible en la misma carpeta que el ejecutable de S2PEditor.

12.1. Empezando

Existen dos formas de iniciar un proyecto: crear uno nuevo o abrir uno existente. En caso de que sea la primera vez que ejecutamos S2PEditor, se creará un proyecto nuevo, para lo que se nos pedirá que indiquemos el nombre del mismo y los parámetros del primer nivel. Si no es la primera vez que ejecutamos el programa, S2PEditor cargará el último proyecto que se haya editado y, de no ser éste válido, preguntará por uno nuevo.

Supondremos que los nombres elegidos para el proyecto y su primer nivel son Tutorial y Level 1 respectivamente. Como tamaño de tile eligiremos 32 píxeles, pues como veremos más adelante, las imágenes que utilizaremos para los tilesets están compuestas por tiles de este tamaño. Al nivel le daremos un tamaño de 20x15 tiles, es decir 20 tiles de ancho y 15 de alto (ver Figura 12.1); de esta forma su tamaño en píxeles será 640x480, el mismo que tendrá la ventana del juego. En los siguientes niveles utilizaremos tamaños superiores, pero en este primer nivel preferimos que no haya desplazamiento de cámara lateral ni horizontal para que se aprecie la diferencia con los posteriores.

Al pulsar el botón OK se cargará el mapa del primer nivel y sobre él veremos una rejilla cuyas celdas tienen una separación igual al tamaño de tile que hayamos elegido. En este mapa se irán reflejando los cambios que introduzcamos en el nivel.

Para guardar los avances en el proyecto podemos usar la opción del menú Project→Save o el botón  de la barra de herramientas. Además al realizar ciertas acciones, como añadir o eliminar un nivel, el proyecto se guardará automáticamente. Se recomienda guardar el proyecto con cierta frecuencia a medida que se sigue este tutorial.

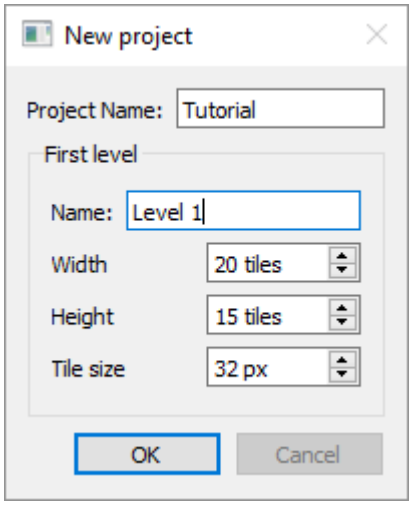


Figura 12.1 – Diálogo para crear un nuevo proyecto.

12.2. Niveles

En este apartado se discutirán las características que presentan los niveles en S2PEditor y se añadirán los niveles 2 y 3, que serán los que formen parte del proyecto Tutorial junto al nivel creado al inicio del mismo en el apartado 12.1. Los niveles son cada una de las partes individuales en que se divide el juego que se va a generar. Al crear un nivel tenemos que especificar una serie de propiedades:

Nombre El nombre del nivel, con el que lo identificaremos en el editor.

Ancho Ancho del nivel medido en tiles.

Alto Alto del nivel medido en tiles.

Tamaño de tile Tamaño de las tiles en píxeles. Las tiles deben ser cuadradas, por tanto sólo es necesario un valor.

Con estos parámetros se sabe el tamaño del nivel, con lo cual ya puede ser construido y mostrado en el editor, aunque sólo veamos en principio un fondo gris representando que está vacío y una cuadrícula con separación igual al tamaño de las tiles, que nos facilitará la colocación de éstas por el escenario. Cada nivel dispone además de una serie de elementos propios cuya elección será la que realmente defina su apariencia y su contenido:

Escenario Llamaremos así a la combinación de todas las tiles dispuestas por el nivel.

Backgrounds Imágenes que vemos de fondo, detrás del escenario.

Objetos Aportan el contenido dinámico del juego como enemigos, ítems coleccionables, *checkpoints*, la meta o el propio avatar del jugador.

En sus secciones correspondientes se irán definiendo estos elementos para cada nivel, mientras que en ésta nos limitaremos a crearlos y explicar qué acciones se pueden realizar con ellos. Procedemos pues a añadir otro nivel al proyecto, mediante opción del menú LEVEL→NEW o el botón de la barra de tareas. Se nos mostrará un diálogo pidiendo la misma información que se pedía para el primer nivel del proyecto; lo llamaremos Level 2 e indicaremos que tendrá 60 tiles de ancho y 15 de alto, con tamaño de tile igual a 32 (ver Figura 12.2). Al aceptar se añadirá el nuevo nivel y se guardará el proyecto automáticamente.

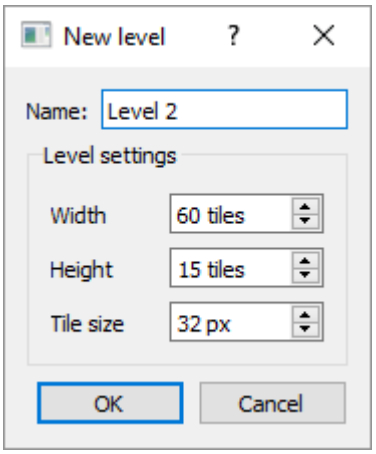


Figura 12.2 – Diálogo para añadir un nuevo nivel.

Con estos parámetros crearemos un nivel el triple de ancho que el primero y con su misma altura. Notaremos que el nivel actual ha cambiado tanto por el mapa que se nos muestra, más grande, como por el título de la ventana principal. Por último crearemos un nivel de 20 tiles de ancho y 45 de alto, llamado Level 3 y con el mismo tamaño de tile usado hasta ahora, el cual es el mismo para los tres niveles puesto que compartirán tilesets.

Los niveles se gestionan por medio de una ventana anclada a la que en adelante llamaremos **gestor de niveles**; dicha ventana se encuentra por defecto solapada con la de los tilesets y para verla tendremos que pulsar sobre la pestaña LEVELS (ver Figura A.1). Se puede modificar libremente la posición del gestor de niveles (así como del resto de ventanas ancladas) dentro de la ventana principal o dejarlo como una ventana flotante. Si se han seguido los pasos descritos, deberíamos ver en el gestor los tres niveles añadidos, tal como se muestra en la Figura 12.3.

La parte central del gestor de niveles es un recuadro con una lista de los niveles del proyecto, donde podremos seleccionarlos haciendo un único click del ratón sobre ellos (con doble click se cargará el nivel). A la derecha vemos una serie de botones para interactuar con los niveles y en la parte baja de la ventana podemos leer el nombre del nivel seleccionado y la ruta del archivo XML que lo contiene. Las acciones que podemos realizar con los niveles de la lista son los siguientes:

Subir Para mover el nivel hacia arriba un puesto en la lista, pulsar el botón .

Bajar Para mover el nivel un puesto hacia abajo en la lista, pulsar el botón .

Cargar Para cargar un nivel de la lista , hacer doble click sobre su nombre. No se guardarán los cambios en el nivel actual.

Guardar y cargar Para cargar un nivel de la lista guardando previamente los cambios en el nivel actual, pulsar el botón .

Eliminar Para eliminar un nivel del proyecto, pulsar el botón . Esta acción borrará el nivel permanentemente, por lo que se pedirá confirmación.

Es importante resaltar que ni los niveles de la lista ni el orden en que aparecen tienen por qué coincidir con los del juego final, como veremos en el apartado 12.9.

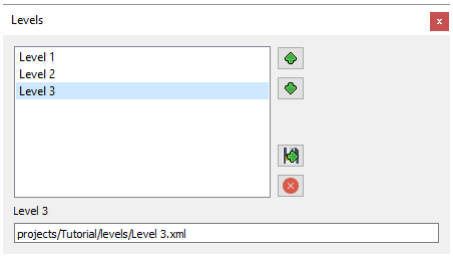


Figura 12.3 – Ventana del gestor de niveles.

12.3. Tilesets

Con los niveles del proyecto ya creados, lo siguiente será añadir algunos tilesets que nos permitan dan forma a los escenarios. Para ello debemos disponer de imágenes que se puedan dividir en tiles, es decir, en fragmentos de la imagen que podremos seleccionar y disponer por el escenario de manera individual. Estas tiles tienen que ser cuadradas y medir todas lo mismo. La imagen que utilizaremos para crear el primer tileset está disponible en la carpeta Recursos con el nombre tilesetLand.png; en la Figura 12.4 podemos verla dividida en tiles. También se puede usar cualquier otra imagen que el usuario desee siempre y cuando cumpla las características descritas.

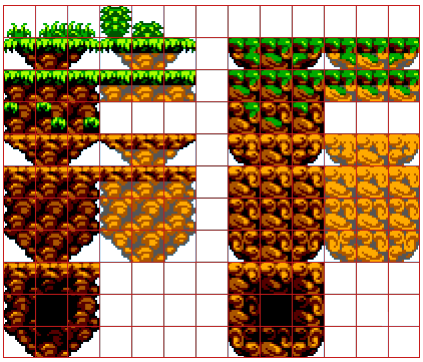


Figura 12.4 – Imagen del primer tileset dividida en tiles de 32x32 píxeles.

De forma análoga a como ocurría con los niveles, los tilesets se administran desde una ventana anclada a la que denominaremos **gestor de tilesets** (ver Figura A.1). En un instante describiremos todas sus funciones, pero de momento nos centraremos en el botón +, cuyo cometido es añadir un nuevo tileset. Al pulsarlo se abrirá un diálogo pidiéndonos información sobre el nombre y la imagen del tileset. Lo bautizaremos Land para recordar que se compone de las tiles para dibujar porciones de tierra.

De la imagen se pide el archivo que la contiene y una serie de parámetros. Si hemos añadido alguna imagen al proyecto podemos seleccionarla en la caja desplegable, pero como en nuestro caso no es así tendremos que buscarla pulsando el botón para examinar ... Se nos abrirá un diálogo NEW IMAGE que, dada una imagen y un nombre que la identifique (ID), la añadirá al proyecto; pulsamos en el botón de examinar, elegimos la imagen tilesetLand.png y le damos el nombre tilesetLand, tal como se muestra en la Figura 12.6.

Cuando pulsemos OK, la imagen recién añadida se marcará como seleccionada en la caja desplegable de ID. El resto de parámetros son el tamaño de las tiles (*tile size*), el espacio entre

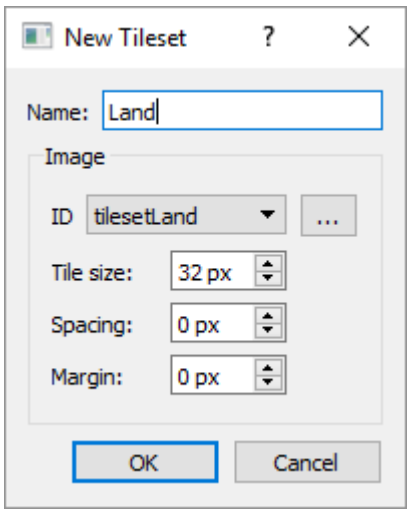


Figura 12.5 – Diálogo para añadir un nuevo tileset.

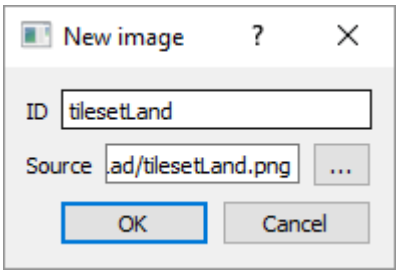


Figura 12.6 – Diálogo para añadir una nueva imagen al proyecto.

ellas (*space*) y el margen que tenga la imagen (*margin*), todo ello en píxeles; nuestra imagen no tiene margen y se compone de tiles de 32x32 píxeles sin espacio entre ellas, por lo que el diálogo NEW TILESET debería quedar como en la Figura 12.5.

Al aceptar habremos añadido el tileset, y veremos cómo aparece una nueva pestaña en el gestor de tilesets (ver Figura 12.7). Los nuevos tilesets que vayamos sumando al proyecto se cargarán en sus respectivas pestañas, y podremos cambiar de tileset haciendo click izquierdo sobre éstas.

Las acciones que podemos realizar sobre cada tileset, accesibles desde el gestor de tilesets, son las siguientes:

Seleccionar tile Para seleccionar la tile que vamos a situar en el mapa, basta con hacer click izquierdo sobre ella en la imagen del tileset. Sabremos que está seleccionada correctamente cuando veamos un recuadro rojo delimitándola.

Modificar aumento Podemos ampliar o reducir la imagen del tileset pulsando los botones y , especificando numéricamente el valor porcentual en la *spinbox*, o mediante el atajo CTRL + RUEDA DEL RATÓN. Es importante resaltar que esta modificación de tamaño es sólo para visualizar el tileset, y las tiles que coloquemos en el nivel seguirán teniendo el mismo tamaño que especificamos al añadir el tileset.

Activar y desactivar colisión Los objetos del mapa, principalmente el jugador y los enemigos, colisionarán con las tiles si al colocarlas la *checkbox* llamada COLLISION está marcada, y las atravesarán si no lo está. A las primeras las llamaremos tiles colisionables o sólidas, y a las últimas tiles no-colisionables o transparentes. Por defecto, la checkbox estará marcada y por tanto las tiles que se coloquen serán colisionables.

Eliminar tileset Pulsando sobre el botón de cada pestaña eliminaremos el tileset para siempre y se borrarán las tiles de dicho tileset de todos los niveles del proyecto. Ésta es una acción irreversible y se pedirá confirmación antes de llevarla a cabo definitivamente.

Añadiremos otro tileset que nos servirá para dibujar nubes, al que se sugiere llamar Clouds. La imagen que se usará es tilesetClouds.png y nuevamente las tiles son de 32x32 píxeles, sin

Figura 12.7 – Ventana de tilesets después de añadir el primero.

separación ni margen. Al aceptar el diálogo de NEW TILESET veremos que se ha sumado una pestaña al gestor de tilesets con el nombre Clouds o el que hayamos elegido.

Con estos dos tilesets incluidos en el proyecto, ya estamos preparados para construir los escenarios de los niveles. Para insertar en el mapa la tile seleccionada en el gestor de tilesets, tendremos que cambiar el modo de inserción (por defecto no se insertará nada) desde el menú, mediante la opción INSERT→TILE, o pulsando el botón de la barra de herramientas. Para colocar las tiles en el mapa del nivel basta hacer click izquierdo sobre el lugar donde queramos situarla; también pueden borrarse haciendo click derecho. Cada una de las celdas de la cuadrícula que se ve sobre el mapa del nivel puede albergar una tile sólida y una transparente.

En las figuras 12.8, 12.9 y 12.10 puede verse cómo hemos dispuesto las tiles para crear los escenarios de Level 1, Level 2 y Level 3, respectivamente. Las tiles correspondientes a la vegetación y a las nubes son no-colisionables; en Level 1 se resaltan para mayor claridad. El usuario es libre de seguir estos ejemplos o no, pero es importante señalar que los escenarios se han creado así teniendo en cuenta los backgrounds y objetos que se le van a añadir, por lo que si se piensa seguir el tutorial hasta el final es recomendable construirlos como mínimo semejantes.

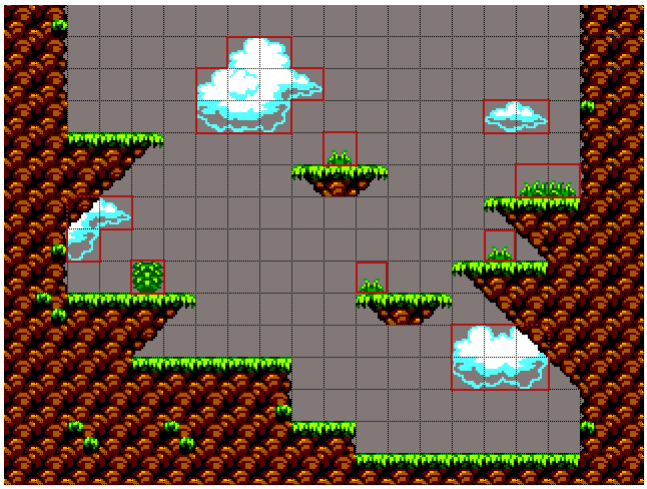


Figura 12.8 – Escenario del nivel Level 1 con las tiles transparentes resaltadas.

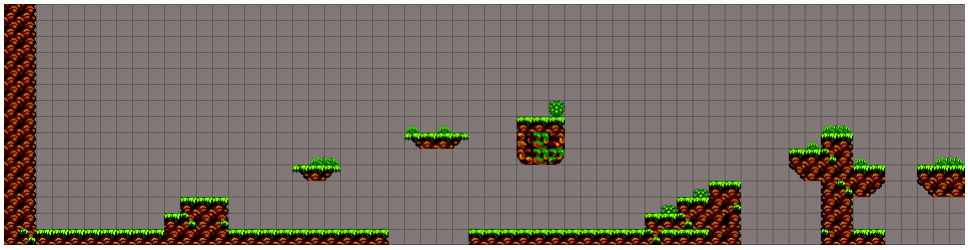


Figura 12.9 – Escenario del nivel Level 2.

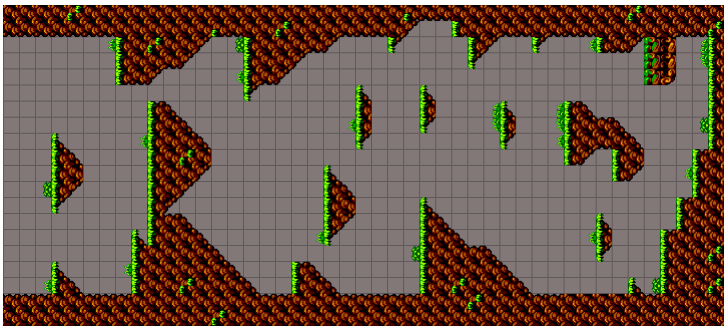


Figura 12.10 – Escenario del nivel Level 3 rotado 90º.

12.4. Imágenes

Hasta ahora, cuando nos ha hecho falta seleccionar una imagen (ver Figura 12.5), la hemos añadido al proyecto sobre la marcha. Si de antemano sabemos qué imágenes vamos a utilizar puede ser más cómodo añadirlas al principio y, cuando estemos creando los distintos elementos que las utilicen, preocuparnos sólo de seleccionarlas. Esto puede hacerse por medio del **gestor de texturas**, al que se accede desde la opción del menú PROJECT→TEXTURES o desde el el botón de la barra de tareas.

Al abrirlo veremos una lista con las imágenes que hemos añadido al proyecto ordenadas alfabéticamente, que si se ha seguido el tutorial serán tilesetClouds y tilesetLand (ver Figura 12.11). Seleccionando una imagen de la lista podemos ver la ruta del archivo fuente que la contiene, así como la propia imagen; es posible aumentar el tamaño de la ventana para ver la imagen al completo. El gestor de texturas es sencillo, pues aparte de esto las únicas acciones que podemos llevar a cabo son añadir una nueva imagen y eliminar una existente:

Añadir imagen Pulsando el botón se nos mostrará un diálogo NEW IMAGE (ver Figura 12.6) preguntándonos por el nombre que le vamos a dar a la nueva imagen (ID) y el archivo en el que se encuentra.

Eliminar imagen Para eliminar una imagen añadida previamente, debemos primero seleccionarla en la lista y después pulsar sobre el botón . Es importante asegurarse antes de que ningún elemento la está utilizando.

Se deja a elección del usuario si añadir las imágenes de la carpeta Recursos ahora o ir añadiéndolas cuando haga falta. Para lo que resta de tutorial se supondrá por simplicidad que ya están añadidas todas las imágenes con ID igual al nombre del archivo sin la extensión; esto quiere

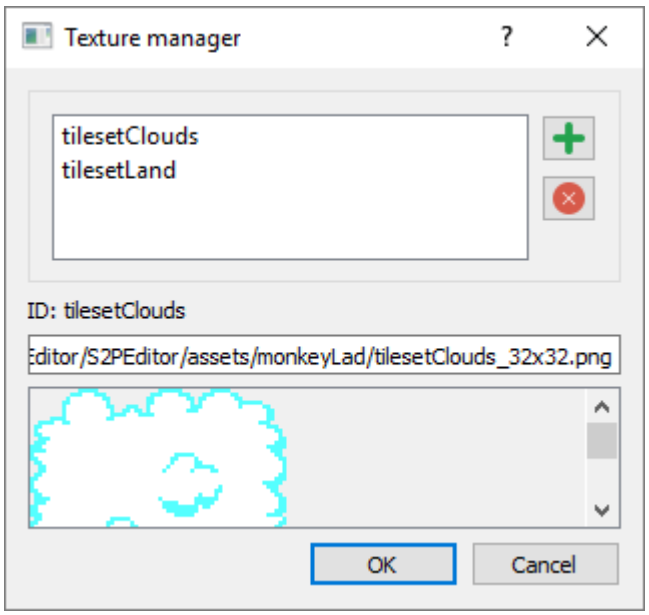


Figura 12.11 – Gestor de texturas.

decir que si por ejemplo se indica «seleccionamos la imagen bckBlue» y el usuario aún no la tiene en su proyecto, sabrá que debe añadir la imagen bckBlue.png y seleccionarla.

12.5. Sonidos

Accederemos al **gestor de sonidos** mediante la opción del menú PROJECT→SOUNDS o desde el el botón de la barra de tareas. Las opciones que nos permite el gestor de sonidos son análogas a las que nos permitía el de texturas: añadir o eliminar un sonido. La diferencia principal con las texturas es que tendremos dos tipos de sonidos:

- Los **efectos de sonido** o sfxs son los distintos sonidos que se reproducen cuando los objetos realicen ciertas acciones especiales.
- A cada nivel se podrá asociar una **pista de música** que sonará en bucle mientras estemos en dicho nivel.

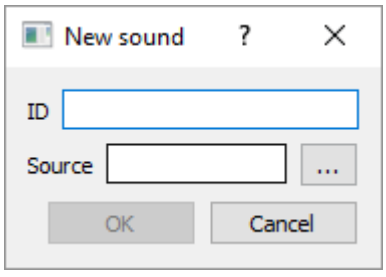


Figura 12.12 – Diálogo para añadir un archivo de sonido al proyecto.

Por tanto, el gestor de sonidos estará dividido en dos paneles, cada uno con sus botones y para añadir o eliminar sfxs o pistas de música. Al pulsar en el botón para añadir un sonido (sea sfx o música), se mostrará un diálogo NEW SOUND como el de la Figura 12.12, que dada la ruta de un archivo de sonido y la ID que queramos darle, lo añadirá al proyecto.

Añadiremos el archivo de sonido playerJump.wav con la ID playerJump, que más adelante asociaremos al salto del jugador. reproduciremos cuando el jugador salte. Además añadiremos la pista de música que asignaremos

al primer nivel, a la que llamaremos random, a partir del archivo random.ogg.

En la Figura 12.13 podemos ver cómo quedará el gestor de sonidos después de añadir estos dos archivos. Puede verse que playerJump está en la parte de efectos de sonido y random en la de pistas de música. Además, podemos ver la ruta del archivo fuente y el tipo (sfx o música) del sonido que seleccionemos.

Al igual que se dijo con las imágenes, de aquí en adelante se supondrá que ya se han añadido todos los sonidos e imágenes, y nos referiremos a ellos por su nombre sin la extensión. En la carpeta Recursos se encontrarán los archivos, con extensión WAV para los sfxs y OGG para las pistas de música.

Música en los niveles

Para asignar una pista de música de las que hayamos añadido al nivel actual, iremos a la opción del menú LEVEL→MUSIC. Se nos mostrará un sencillo diálogo (Figura 12.14) con una lista desplegable que nos permitirá elegir una de entre todas las pistas de música añadidas. Para el primer nivel elegiremos random, para el segundo HouseInAForest y para el tercero AJourneyAwaits.

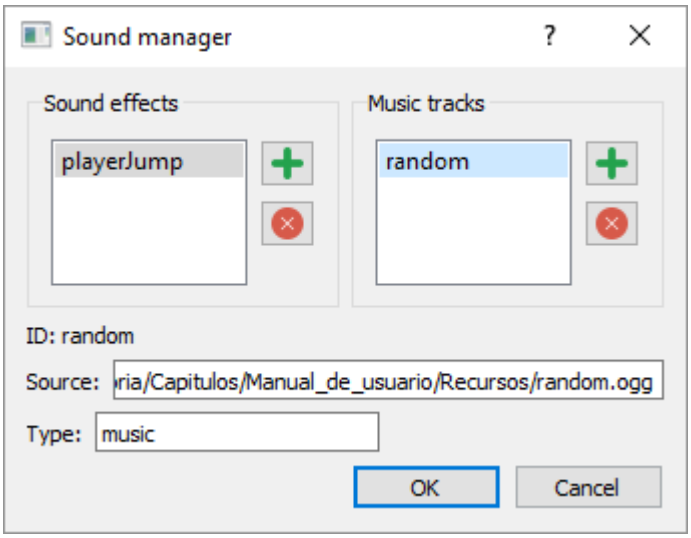


Figura 12.13 – Gestor de sonidos

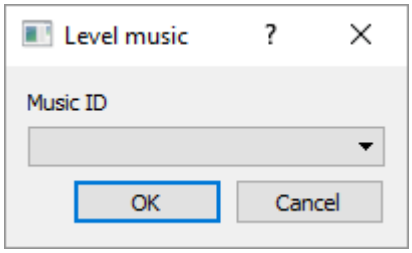


Figura 12.14 – Diálogo para elegir la música a un nivel.

12.6. Backgrounds

Añadiremos a nuestro primer nivel, el cual es del mismo tamaño que la pantalla del juego, un background estático. Para ello accederemos al **gestor de backgrounds**, por medio de la opción del menú LEVEL→BACKGROUNDS o pulsando el botón de la barra de herramientas. En la nueva ventana veremos que no hay ningún background añadido; para remediarlo pulsaremos el botón y se nos abrirá un nuevo diálogo. Los parámetros que se nos piden en el diálogo son:

Type Podemos elegir tres tipos de background: STATICBCK, PARALLAXBCK y MOVINGBCK, que son abreviaturas para *static*, *parallax* y *moving* background respectivamente. El valor por defecto es PARALLAXBCK.

Image Imagen del background. Puede seleccionarse cualquier imagen del proyecto o añadirse una nueva. El valor por defecto es la primera imagen del proyecto y, si no hubiera ninguna, la caja desplegable estará vacía.

X speed Velocidad en píxeles a la que se moverá el background en el eje horizontal, por defecto igual a cero. Los tipos *static* y *parallax* ignorarán este valor.

Y speed Velocidad en píxeles a la que se moverá el background en el eje vertical, por defecto igual a cero. Los tipos *static* y *parallax* ignorarán este valor.

Cambiaremos el tipo a StaticBck y seleccionaremos la imagen bckBlue. En la Figura 12.15 vemos cómo quedarán los campos del diálogo después de estos pasos. Al aceptar veremos en el gestor el background que acabamos de añadir, cuyo nombre será el mismo que le hayamos dado a la imagen. Podemos ver el valor de sus parámetros, aunque no podemos modificarlos por lo que si nos hemos equivocado deberemos borrarlo e incluirlo nuevamente. Además, debajo de los parámetros podremos ver la imagen, que en este caso no tiene mucho interés puesto que simplemente es un fondo azul (ver Figura 12.16). Al pulsar OK veremos cómo ha aparecido en el nivel un fondo azul detrás de las tiles.

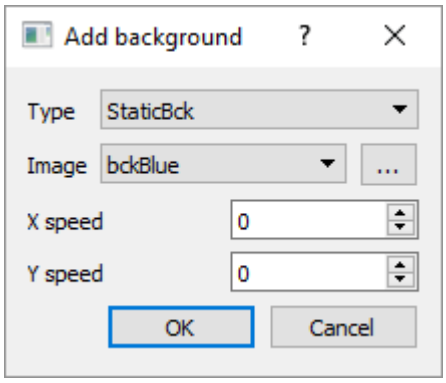


Figura 12.15 – Diálogo para añadir un nuevo background.

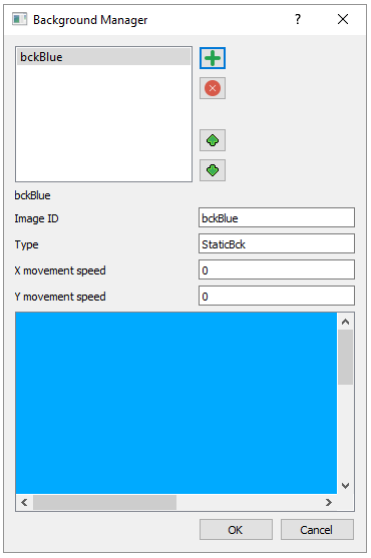


Figura 12.16 – Gestor de backgrounds del primer nivel.

Al contrario que los tilesets o las imágenes, y como cabría esperar, los backgrounds no son comunes al proyecto sino específicos de cada nivel; por tanto si cargamos el nivel Level 2 y abrimos el gestor de backgrounds veremos que está vacío. Le añadiremos uno de tipo ParallaxBck con la imagen bckCloudsH, la cual es un fondo azul con nubes. Nótese que en Level 1 las nubes estaban hechas con tiles transparentes y el escenario simplemente era el fondo azul, mientras que ahora las nubes forman parte de la propia imagen del escenario. Al confirmar habremos añadido nuestro *parallax* background, y moviendo la barra de scroll horizontal podemos apreciar que se mueve a distinta velocidad (más lento) que el escenario; no obstante, cuando probemos el nivel lo veremos más claramente.

Seguiremos el mismo proceso para el nivel Level 3, añadiendo un background de tipo ParallaxBck usando en este caso la imagen bckCloudsV.

En el segundo nivel el fondo se moverá horizontalmente y en el tercero verticalmente. Esto es algo que no es necesario especificar: como se mencionó previamente, S2PEditor calcula automáticamente la velocidad a la que el background tiene que moverse en ambos ejes, por tanto si una de las dos dimensiones coincide con la del nivel, la velocidad será cero y no se moverá en ese eje.

En caso de que se quiera incluir un MovingBck, habrá que especificar la velocidad a la que queremos que se mueva en cada eje. El número que se indique serán los píxeles que el fondo se mueva cada paso, siendo un paso una iteración completa del bucle de actualización del juego[1]; por defecto este número es cero, lo que equivale a un StatickBck. En los recursos se incluye la imagen bckCloudsMv.png, la cual puede usarse para probar un background móvil horizontalmente.

12.7. Objetos

A pesar de tener un comportamiento básico predefinido según su tipo, los objetos tienen una serie de atributos modificables que, junto al escenario, serán en última instancia los que definan lo divertido y jugable que es el producto final.

Un objeto no se actualiza hasta que aparece por primera vez en pantalla. En el caso de los enemigos esto quiere decir que se quedarán quietos en el lugar en el que los hayamos colocado hasta que entren por primera vez dentro de la cámara, momento en que se empezarán a mover según los valores que les hayamos dado.

12.7.1. Gestor de objetos

Los objetos se administran desde el **gestor de objetos** (ver Figura A.1), el cual nos permite realizar las siguientes acciones:

Crear objeto Dentro del recuadro New object hay un botón por cada tipo de objeto, representados por los mismos iconos que se han mostrado al principio de cada descripción en el epígrafe 6.3.4 del análisis de s2Dp. Al pulsar el botón correspondiente al objeto que queremos crear, se nos abrirá un diálogo con las opciones que queremos configurar; profundizaremos en este diálogo en el apartado «Crear un objeto». Al especificar los parámetros deseados y aceptar, el nuevo objeto se guardará con el nombre que le hayamos dado.

Seleccionar objeto Una vez que hayamos creado un objeto con los parámetros que queramos, se guardará y podremos seleccionarlo en la caja desplegable Name para incluirlo al nivel, editarlo o eliminarlo.

Editar objeto Pulsando el botón  podemos modificar la configuración de un objeto previamente creado, sobrescribiéndolo o guardándolo como un nuevo objeto con otro nombre. Si vamos a sobrescribir se pedirá confirmación.

Eliminar objeto Para eliminar un objeto hay que seleccionarlo y pulsar el botón  . Se pedirá confirmación.

[1] En un segundo el bucle del juego se actualiza 50 veces.

12.7.2. Crear un objeto

En este apartado comentaremos cómo crear un objeto explicando los distintos campos del diálogo NEW OBJECT, el cual es ligeramente distinto según el tipo que vayamos a crear. Aprovechando que los objetos que creemos serán accesibles desde todos los niveles del proyecto, empezaremos creando uno que vayamos a utilizar en los tres: el **avatar del jugador**. Para ello pulsaremos el botón correspondiente, el primero del grupo OTHERS, y veremos que aparece una nueva ventana con un diálogo para crear un nuevo jugador; podemos ver el diálogo con todos los campos completos en la Figura 12.20. Iremos explicando uno a uno los campos de que se compone la ventana:

Name Nombre que le vamos a dar al objeto que estamos creando. Es importante darle un nombre que lo represente bien, para que no tengamos dudas después a la hora de seleccionarlo. Le llamaremos playerMonkey, puesto que nuestro jugador usará la imagen de un mono.

Type Tipo del objeto. Este campo se rellena automáticamente según el botón del recuadro NEW OBJECT que hayamos pulsado, y no se puede modificar.

Sprite Aquí es donde indicaremos a S2PEDITOR qué apariencia va a tener el objeto. En la caja desplegable ID seleccionaremos la imagen que hará de hoja de *sprites* o *spritesheet*, es decir, que contendrá todos los *frames* que se utilizarán para animar el objeto. Al abrir el diálogo por defecto nos aparecerá seleccionada la primera imagen del proyecto por orden alfabético.

Seleccionaremos la imagen monkeySheet, y podremos visualizarla en la parte inferior derecha de la ventana. Le daremos más importancia a la spritesheet más adelante, pero de momento sólo nos fijaremos en las spinboxes WIDTH y HEIGHT, las cuales se habrán rellenado automáticamente con las dimensiones de la imagen que hayamos elegido que, si ha sido la propuesta, serán 384 píxeles de ancho y 192 píxeles de alto. Ésta cantidad sin embargo no es la que nos interesa en este caso[2], pues lo que tenemos que indicar son las dimensiones de un único frame, es decir, del sprite que representa al objeto en el mundo del juego.

Para calcular el ancho y el alto de un frame hay que, respectivamente, dividir el ancho de la spritesheet entre el número de columnas y el alto entre el número de filas (ver Figura 12.18): en nuestro caso el ancho serán $384/6 = 64$ píxeles, y el alto $192/4 = 48$ píxeles.

Bounding box Aquí definiremos la superficie colisionable del objeto. Podemos indicarlo numéricamente, aunque lo más sencillo es hacerlo gráficamente pulsando el botón  , con lo que se nos abrirá una nueva ventana en la que podremos dibujar la caja de colisión haciendo click con el botón derecho y arrastrando (ver Figura 12.17).

Speed Aquí se especificarán los diferentes valores relacionados con la velocidad del objeto. Como ocurría con los background móviles, la velocidad se mide en píxeles por paso.

Habrá objetos en los que no tengamos que tocar nada y otros en los que haya que indicar su velocidad en uno o ambos ejes. El tipo PLAYER además nos pide que fijemos la velocidad a la que asciende el jugador cuando salta, por lo que las spinboxes a rellenar serán:

[2]En casos en los que la imagen se componga de un único frame sí que coincidirían los valores.

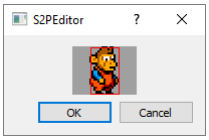


Figura 12.17 – Indicando gráficamente la superficie de colisión de un objeto.

x Velocidad a la que se mueve el jugador horizontalmente. Le daremos un valor de 3 píxeles por paso.

y Velocidad a la que se mueve el jugador verticalmente, es decir a la que cae. Es equivalente a la gravedad. Un valor de 5 píxeles por paso servirá.

Jump Velocidad a la que se mueve el jugador en el eje y cuando salta. En el eje x se seguirá moviendo a la indicada en la spinbox x. Fijaremos el mismo número de píxeles por paso que para la gravedad: 5.

Other En esta sección se agruparán atributos especiales de cada objeto, como la cantidad de daño que causan los enemigos, la cantidad que suma un ítem recogido a su categoría o, en el caso del jugador, los siguientes:

Invincible Tiempo que el jugador permanece invulnerable tras recibir daño, medido en pasos del juego. Cualquier valor entre 50 y 100 servirá a nuestros propósitos.

Jump height Altura que alcanza el salto del jugador, medida en píxeles. Un valor de 70 píxeles casa bien con lo que hemos usado hasta ahora.

Sounds A cada objeto se podrán asignar uno o varios efectos de sonido, dependiendo de su tipo. En el caso del jugador son los siguientes.

Death Sonido reproducido cuando el jugador muere, es decir, cuando pierde toda la salud. Le asignaremos el sonido playerDeath.

Jump Sonido reproducido cada vez que el jugador presiona el botón de salto. Le asignaremos el sonido playerJump.

Animation frames En esta tabla es donde le daremos al editor la información sobre las animaciones del objeto. Para facilitar la tarea, debajo de la tabla se puede ver la spritesheet que hemos seleccionado previamente.
La tabla tiene tantas filas como animaciones diferentes realice el objeto que se va a crear. Los enemigos tienen dos animaciones, una para cuando están quietos y otra para cuando se están moviendo, mientras que los ítems sólo tienen una puesto que no se mueven. El tipo PLAYER es el más complejo y dispone de cuatro animaciones: quieto, caminando, saltando y muriendo.Las columnas de la tabla de animaciones señalan los tres parámetros que necesita conocer el editor para cada animación:

Row Fila de la spritesheet que corresponde a la animación. Las filas se empiezan a contar desde el número cero.

Number Número de frames que contiene la animación o, lo que es lo mismo, la fila de la spritesheet que hemos indicado en ROW.

Speed Tiempo de espera entre un frame de la animación y el siguiente, expresado en milisegundos.

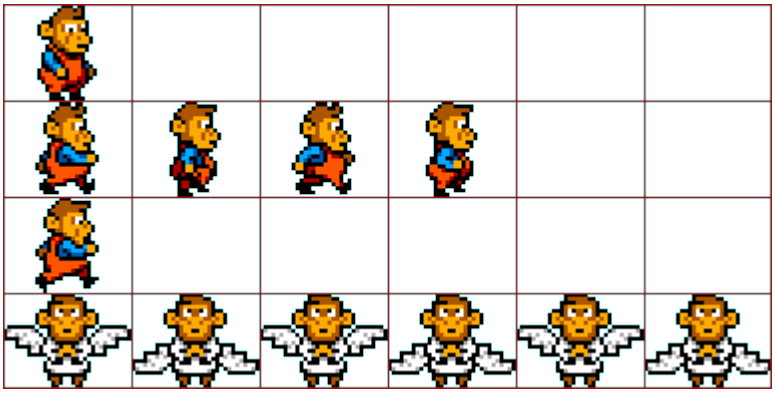


Figura 12.18 – Spritesheet del avatar del jugador dividida en frames.

En la Figura 12.20 pueden verse los valores elegidos para la tabla de animaciones, así como para el resto de los atributos del objeto PLAYER.

Si confirmamos el diálogo se guardará el objeto con la configuración de atributos que le hemos indicado y con el nombre playerMonkey. Observaremos además que el objeto recién añadido se seleccionará automáticamente. Para insertar un objeto en el mapa del nivel procederemos de forma análoga a como lo hicimos con las tiles, cambiando el modo de inserción por medio de la opción del menú INSERT→OBJECT o el botón de la barra de herramientas. Si, como debería ser llegado a este punto, tenemos el modo de inserción en objetos y el objeto playerMonkey seleccionado, al mover el ratón por encima del nivel apreciaremos una previsualización de cómo quedaría el objeto si lo colocamos en ese punto, como vemos en la Figura 12.19; el recuadro rojo nos delimita la superficie de colisión y el negro nos indica hasta dónde llega el frame. Como con las tiles, colocaremos los objetos en el mapa con el botón izquierdo del ratón y los borraremos con el derecho.

Figura 12.19 – Previsualización de un objeto en el editor.

Ya tenemos el avatar del jugador, la parte más importante del juego. El resto de objetos se crean de igual forma, con la salvedad de que la tabla de animaciones, los atributos especiales y los efectos de sonido del diálogo NEW OBJECT varían según el tipo el objeto que vayamos a crear.

En el apartado 12.7.3 se presenta una selección de objetos con la que se cubren todos los tipos. Estos objetos, junto a playerMonkey, serán los que repartamos por los tres niveles creados para dar forma al juego. El resultado final puede verse en el epígrafe 12.7.4.

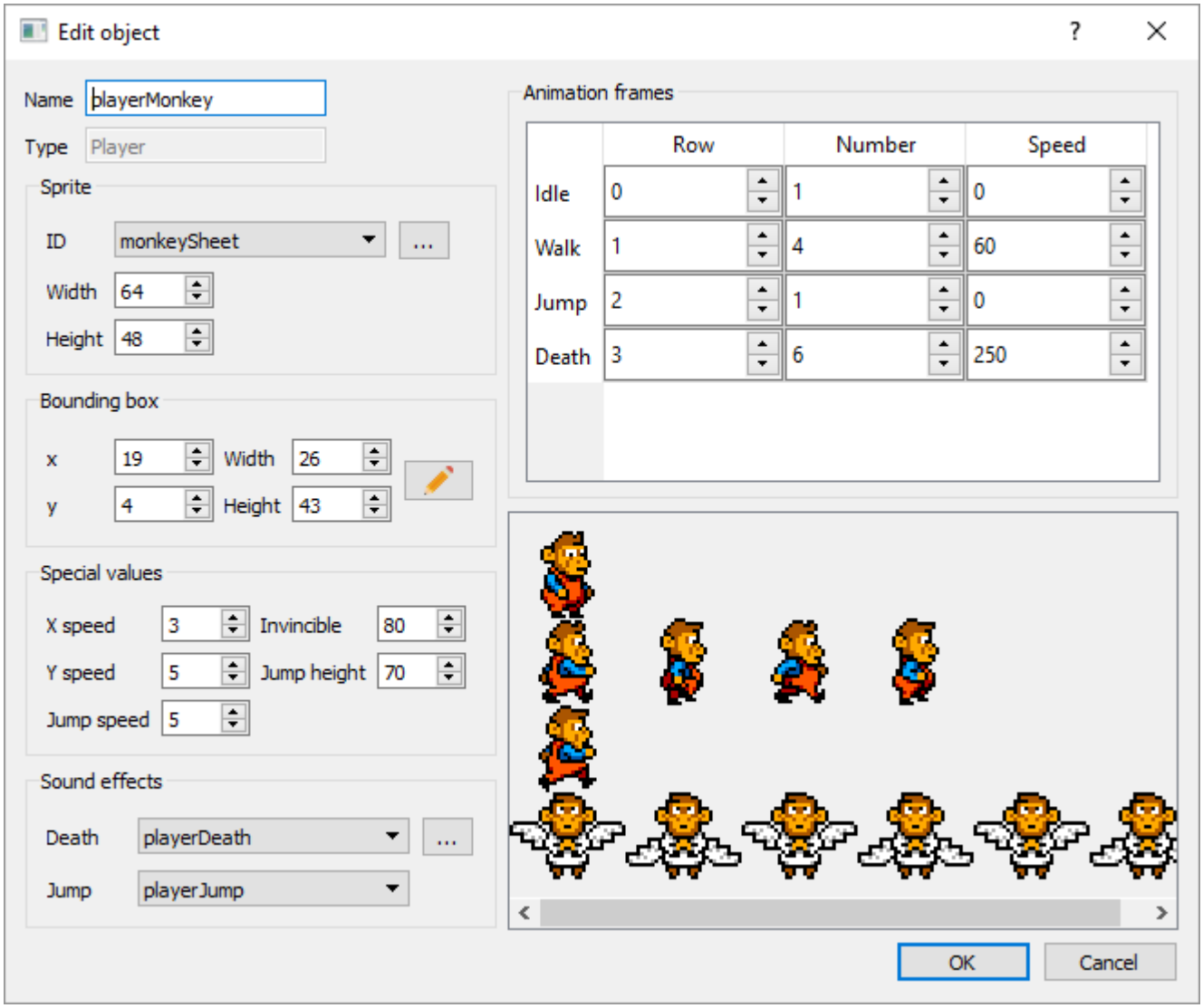


Figura 12.20 – Diálogo con los valores definitivos para crear un nuevo PLAYER.

12.7.3. Ejemplos

Aquí se mostrarán los objetos que se han creado para construir los niveles. Se presentará una breve descripción de cada objeto seguida de su hoja de sprites, y varias tablas con información sobre sus atributos numéricos, sus sonidos y sus animaciones. Los dividiremos en los mismos grupos en que se dividen en el gestor de objetos, empezando por los enemigos.

El daño que hacen los enemigos o la cantidad que suman los ítems se han decidido teniendo en cuenta los valores máximos que se elegirán para el juego, indicados en «Generar juego». Se aconseja al usuario que varíe estos valores, así como la velocidad a la que se mueven y animan los objetos, y experimente hasta que encuentre la sensación de movimiento que desee alcanzar.

Por último, es importante señalar que en las tablas se han indicado las velocidades en términos absolutos. Una velocidad horizontal (X SPEED) positiva hará que el objeto se empiece moviendo hacia la derecha y una velocidad vertical (Y SPEED) positiva hará que comience moviéndose hacia abajo. Si se quiere invertir este comportamiento, basta con cambiarle el signo a la velocidad.

Enemigos

A continuación se describen los enemigos a los que tendrá que hacer frente el jugador. Los pinchos no son *enemigos* en sí, pero entran dentro de esta categoría al crearse de la misma forma.

Caracol Se mueve despacio y resta poca vida. Es el enemigo básico.

Serpiente Se mueve el doble de rápido que el caracol y hace el doble de daño.

Escorpión Veloz enemigo que perseguirá por tierra al jugador desde que entre en su rango de visión, causándole bastante daño si lo alcanza.

Pájaro verde Enemigo básico volador que se mueve horizontalmente. Si queremos que se mueva en diagonal debemos darle otro valor a Y SPEED.

Pájaro rojo Como el escorpión, el pájaro rojo persigue al jugador en cuanto lo ve, ahora también por el aire. Quita el doble de vida que el pájaro verde.

Rey Serpiente Gran serpiente que hará perder una vida al jugador de un único toque.

Pinchos 1 Pinchos del ancho de una tile que, al contacto, harán perder una vida al jugador.

Pinchos 5 Hilera de pinchos de cinco tiles de ancho que, al contacto, harán perder una vida al jugador.

Tabla de atributos

En la siguiente tabla podemos ver el tipo de cada enemigo, la imagen de su hoja de sprites y el valor de los atributos numéricos. Vemos que algunos enemigos tienen un valor de *health* igual a cero; esto en la práctica significa que no se pueden matar, por lo que entrar en contacto con ellos provocará la muerte del jugador.

Tipo	*Enemigo*	*Imagen*	*x speed*	*y speed*	*Damage*	*Health*
	Caracol	enemSnail	1	5	1	1
	Serpiente	enemSnake	2	5	2	2
	Escorpión	enemScorpion	3	5	3	0
	Pájaro verde	enemBird	2	0	1	1
	Pájaro rojo	enemBirdRed	2	2	2	2
	Rey Serpiente	enemSnakeKing	1	5	10	0
	Pinchos 1	spikes1tile	0	0	10	0
	Pinchos 5	spikes5tiles	0	0	10	0

Tabla de animaciones

A continuación se muestra la configuración que hemos elegido para las animaciones de cada enemigo. No se muestran los pinchos puesto que son estáticos.

Enemigo	*Animación*	*row*	*number*	*speed*
Caracol	idle	0	1	0
	walk	0	2	200
Serpiente	idle	0	1	0
	walk	0	2	150
Escorpión	idle	0	1	0
	walk	0	2	150
Pájaro verde	idle	0	1	0
	walk	0	2	250
Pájaro rojo	idle	0	2	150
	walk	0	2	250
Rey Serpiente	idle	0	1	0
	walk	0	2	150

Tabla de sonidos

Indicaremos qué sonido se reproducirá cuando el enemigo sea dañado y cuando éste dañe al jugador. El enemigo «Rey Serpiente» y los pinchos no aparecen en la tabla, ya que son invencibles (es decir, no tienen *hurt sound*) y matan al jugador de un golpe, con lo cual se reproducirá el sonido de cuando el jugador muere, que ya indicamos previamente cuando lo creamos.

Enemigo	*Hurt Sound*	*Attack Sound*
Caracol	snailHurt	playerHurt
Serpiente	snakeHurt	playerHurt
Pájaro verde	birdHurt	playerHurt
Pájaro rojo	birdHurt	playerHurt
Escorpión	- -	playerHurt

Ítems

Se crearán cuatro ítems de puntuación, tres de salud, dos de llaves y uno para recuperar una vida.

Puntuación +1 Moneda de bronce que aumenta la puntuación en 1.

Puntuación +5 Moneda de plata que aumenta la puntuación en 5.

Puntuación +10 Monea de oro que aumenta la puntuación en 10.

Puntuación +25 Moneda especial que gira sobre sí misma y aumenta la puntuación en 25.

1UP Da al jugador una vida adicional.

Salud +1 Plátano que recupera un punto de salud.

Salud +3 Racimo de plátanos que recupera tres puntos de salud.

Salud llena Poción mágica que recupera la salud al máximo.

Llaves +1 Llave de bronce que aumenta nuestras llaves en una unidad.

Llaves +3 Llave de oro que aumenta nuestras llaves en tres unidades.

Tabla de atributos

A continuación podemos ver la información de cada ítem. La columna «Añade» se refiere a la puntuación, salud, vida o llaves sumadas, dependiendo del ítem. No se muestran los valores para las velocidades en `x` e `y` puesto que los ítems no se moverán. «Pick Sound» es el sonido que se reproduce cuando el ítem es recogido; se ha añadido a la tabla aprovechando que es más pequeña que la de los enemigos, por ahorrar espacio.

Tipo	*Ítem*	*Imagen*	*Añade*	*Pick Sound*
	Puntuación +1	coinBronze	1	score1
	Puntuación +5	coinSilver	5	score5
	Puntuación +10	coinGold	10	score10
	Puntuación +25	coinFish	25	score25
	1up	1up	1	1up
	Salud +1	healthBan	1	health
	Salud +3	healthBans	3	health
	Salud llena	healthPotion	10	healthFull
	Llaves +1	keyBronze	1	keyBronze
	Llaves +3	keyGold	3	keyGold

Tabla de animaciones

Solamente indicaremos los valores de la tabla de animaciones para «Puntuación +25», puesto que el resto de ítems son estáticos, es decir, su hoja de sprites se compone de un único frame.

Ítem	*Animación*	*row*	*number*	*speed*
Puntuación +25	idle	0	4	150

Otros

A continuación se muestran los ítems para el *check point*, el fin de nivel y las distintas puertas. No se mostrará tabla de animaciones ya que todos ellos son estáticos.

Check point Cartel de madera con un plátano dibujado.

Final de nivel Cartel de madera más grande que el del punto de control con un racimo de plátanos dibujado.

Puerta 1 Puerta de plata que requiere del uso de una llave.

Puerta 2 Puerta plateada con dos cerraduras que requiere del uso de dos llaves.

Puerta 3 Puerta dorada que requiere del uso de tres llaves.

Tabla de atributos

La columna «Sound» indica el sonido que hacen el *check point* y el fin de nivel al ser alcanzados, y las puertas al ser abiertas. La columna «keys» indica el número de llaves necesarias para abrir las puertas.

Tipo	*Nombre*	*Imagen*	*Sound*	*keys*
	checkPoint	cpBanana	cpSound	- -
	finishBanana	finishBanana	endSound	- -
	door1	doorSilver	doorOpen	1
	door2	doorSilver2	doorOpen	2
	door3	doorGold	doorOpen	3

12.7.4. Niveles finalizados

El último paso para terminar los niveles es poblarlos de objetos. Se recomienda al usuario que vaya probando distintas combinaciones hasta que encuentre la disposición que más le guste; no obstante mostramos ejemplos de cómo podrían colocarse tanto en el nivel Level 1 (Figura 12.21), como en Level 2 (Figura 12.22) y Level 3 (Figura 12.23).

Figura 12.21 – Nivel Level 1 con objetos.

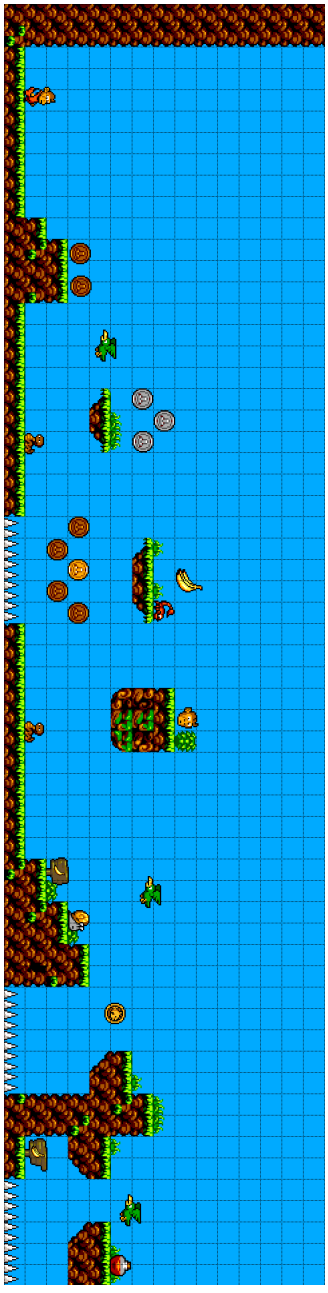


Figura 12.22 – Nivel Level 2 con objetos, rotado 270º.

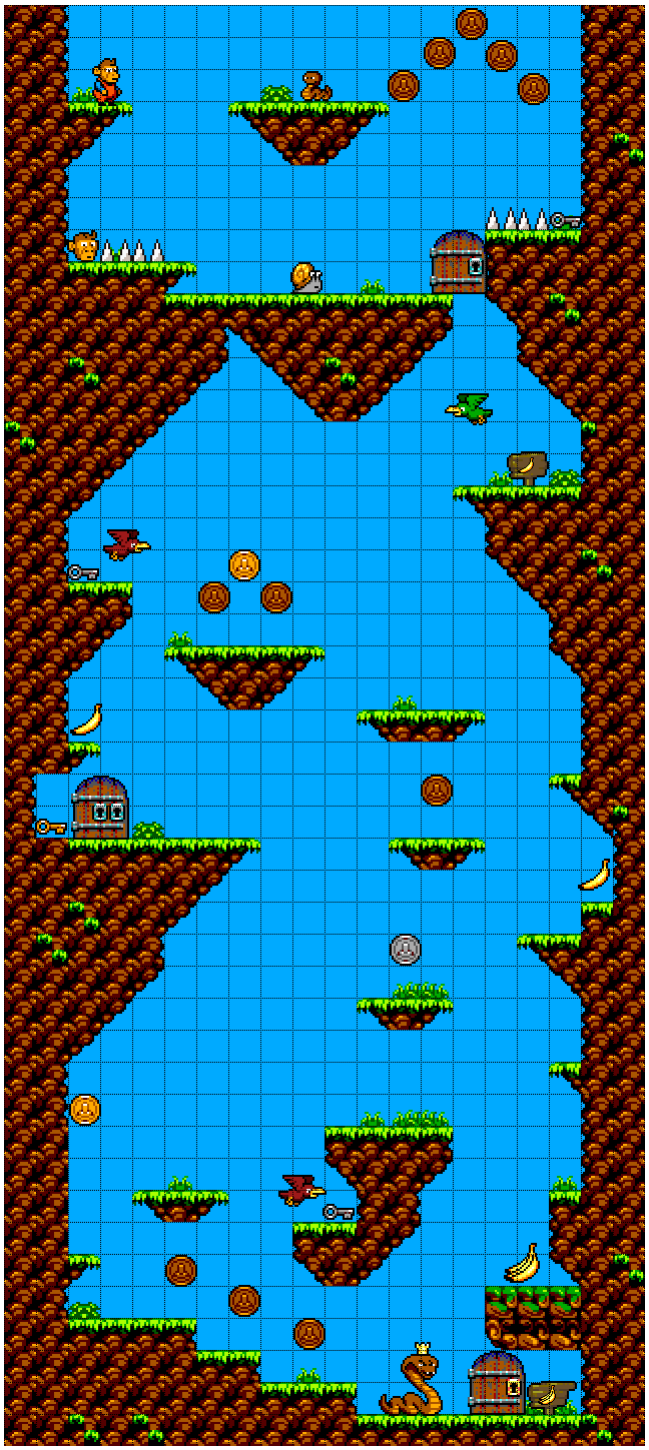


Figura 12.23 – Nivel Level 3 con objetos.

12.8. Probar niveles

Una de las características principales de S2PEditor es que los niveles pueden ser probados en la misma ventana del editor. Para ello podemos usar la opción del menú Level→Run o el botón de la barra de herramientas; se vuelve al modo de edición por medio de Level→Stop o el botón .

Para manejar al jugador se utilizan los controles descritos en el epígrafe 6.3.6, y el nivel transcurrirá de la misma forma que en el juego final, con la excepción de que al perder todas las vidas o llegar al final del nivel, éste se reiniciará. Si se modifica el tamaño de la ventana en la que se ejecuta el editor, la cámara se ajustará en consecuencia

12.9. Generar juego

Antes de generar el juego tendremos que configurar una serie de opciones que afectarán al resultado final; lo haremos a través del **gestor de opciones del juego**, usando la opción del menú Game→Settings o el botón de la barra de herramientas.

El gestor de opciones está dividido en siete recuadros, cada uno de ellos haciendo referencia a un apartado del juego:

Salud Podemos elegir la salud inicial y máxima del jugador, así como el icono que representa la salud restante en el HUD. Como salud inicial y máxima elegiremos 3 y 5, y como icono la imagen healthBan.

Vidas Como número de vidas inicial y máximo hemos elegido 3 y 99, y como icono la imagen 1up.

Puntuación Como puntuación inicial y máxima hemos elegido 0 y 100, y como icono la imagen coinBronze.

Llaves Como número de llaves inicial y máximo hemos elegido 0 y 99, y como icono la imagen keyBronze.

Pantallas Aquí indicaremos qué imágenes le darán forma a la pantalla de inicio, pausado, *game over* y final (ver el apartado 6.3.3); utilizaremos, respectivamente, las imágenes screnStart, screenPause, screenGameOver y screenEnd. Es altamente recomendable que el tamaño de las imágenes que se utilicen coincida con el de la ventana del juego, pues de lo contrario las imágenes quedarán recortadas (si son más grandes) o no llenarán del todo la ventana (si son más pequeñas).

Resolución de la ventana Aquí indicaremos las dimensiones de la ventana donde se ejecutará el juego. Utilizaremos una resolución de 640 píxeles de ancho por 480 de alto, que si recordamos es el tamaño que le dimos al primer nivel, en el que no hay movimiento de cámara vertical ni horizontal.

Niveles Veremos dos listas de niveles, una Editor con los niveles del proyecto actual y otra Game con los niveles que formarán parte del juego, que en principio debería estar vacía.

Eligiendo uno de los niveles del proyecto y pulsando el botón lo añadiremos a los niveles del juego. El orden en el que aparezcan los niveles en la lista GAME será el que tengan en el juego; para modificarlo basta con seleccionar un nivel y pulsar los botones arriba y abajo para adelantarlo o retrasarlo un puesto.

En la Figura 12.24 vemos cómo deberían quedar las opciones del gestor si se han seguido nuestras indicaciones. Al pulsar OK informaremos a S2PEDITOR de los nuevos parámetros para la creación del juego.

Ahora sí estamos listos para generar el juego final, para lo cual usaremos la opción del menú GAME→EXPORT o el botón de la barra de herramientas. Se nos abrirá un sencillo diálogo pidiéndonos el nombre del juego y, al aceptarlo, se nos preguntará por el directorio en el que queremos guardarlo. En dicho directorio se creará una carpeta con el nombre que se le haya dado al juego, que contendrá el ejecutable del juego con el nombre game.exe, acompañado de todos los recursos necesarios para que el juego se ejecute, principalmente imágenes, bibliotecas dinámicas (dll) y los archivos de los niveles que lo componen.

Por ejemplo, si le damos al juego el nombre Tutorial game y elegimos el escritorio como directorio destino, en el escritorio se creará una carpeta llamada Tutorial game dentro de la cual estará el archivo game.exe. Esta carpeta es autosuficiente, es decir, puede llevarse a otro equipo o dispositivo de almacenamiento externo y seguirá funcionando. Para más detalles sobre la distribución de juegos y proyectos, ver la sección A.3 del Apéndice A.

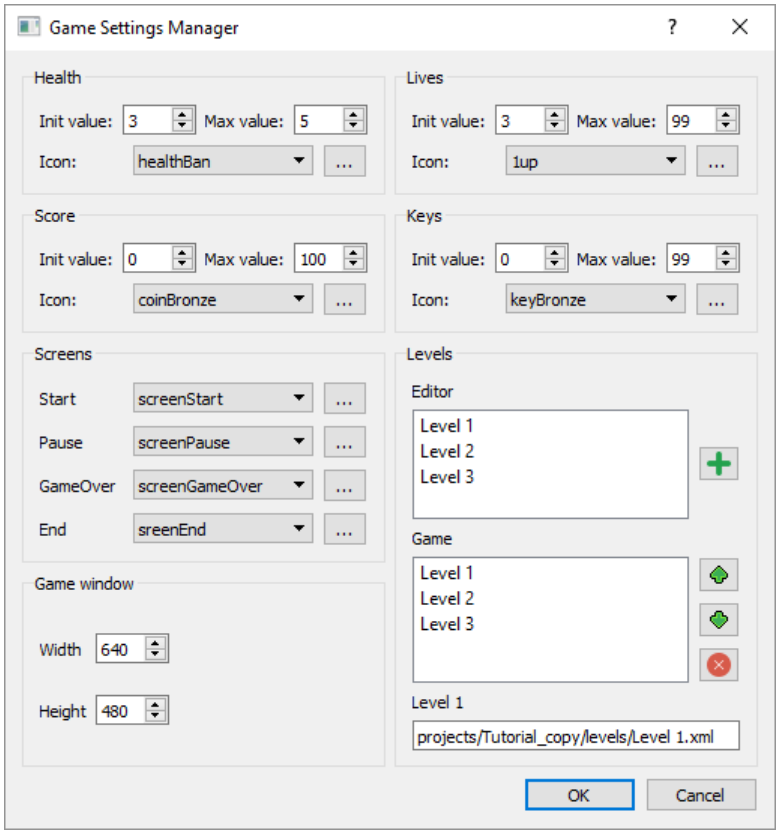


Figura 12.24 – Gestor de opciones del juego.

Parte IV

Conclusiones

Capítulo 13

Costes

Se ha llevado un recuento del tiempo de trabajo apuntando al final de cada sesión a qué se ha dedicado, pero aún así es difícil extraer el número de horas que computarían para el coste. Esto es así porque existe un alto grado de solapamiento entre las tareas que son computables y las que no. Es difícil desacoplar análisis y diseño (computables) con estudio de bibliografía y generación de documentación, que al ser un proyecto académico no es igual a la que se generaría con un proyecto laboral. De igual forma es complicado separar la implementación del estudio y aprendizaje de nuevas herramientas, siendo esto último algo que en el ámbito laboral en teoría no debería darse o sería mucho menor.

No obstante, con los datos disponibles se ha intentado realizar una aproximación que se asemeje lo máximo posible a lo que sería un escenario real. Así, todo el tiempo dedicado a bibliografía y aprendizaje en general se ha excluido del cómputo. Quizá esto no sea del todo correcto, puesto que siempre que se va a realizar un proyecto hay un proceso de estudio previo que supone horas de trabajo. Sin embargo, se ha considerado que esto se compensa con la lentitud en el desarrollo comparado con lo que tardaría un profesional, sobretodo en las primeras etapas del proyecto.

También se ha excluido el tiempo invertido en probar las distintas bibliotecas candidatas, tanto para la construcción del framework s2Dp como del editor S2PEditor, puesto que suponemos que en el ámbito laboral se tendría mucho más claro con qué herramientas trabajar desde el principio. Evidentemente la redacción de esta memoria también queda fuera, excepto la generación de la documentación del editor, que sería una combinación del tutorial del Capítulo 12 y del «Manual de usuario».

Según *Wage Indicator*[37], el salario medio bruto en España de un ingeniero de software licenciado, trabajador por cuenta propia y sin experiencia es de **9.57€/hora**. Según el XVIII Convenio de Ingenierías, publicado en el BOE del 18 de enero de 2017[38], los licenciados y titulados de segundo y tercer ciclo universitario tendrían un nivel salarial de 1687.02€ al mes, lo que equivale a **10.54€/hora** tomando como referencia 40 horas semanales. Por último, consultando las nóminas del Convenio Colectivo TIC en la web de CCOO[39], vemos que un titulado superior se sitúa en 1569.25€/mes, equivalente a **9.81€/hora.** Haciendo una aproximación a partir de la media de las tres cantidades obtenemos un sueldo de **9.97€/hora**, que será el que usemos para calcular el coste.

En la siguiente tabla se muestra el desglose de las horas, los costes parciales y el coste total.

Tarea	Horas	Coste
Desarrollo framework s2Dp	253	2522.41€
Desarrollo editor S2PEditor	307	3060.79€
Desarrollo común a s2Dp y S2PEditor	259	2582.23€
Documentación editor S2PEditor	45	448.65€
Total	**864**	**8614.08€**

Capítulo 14

Valoración

Tras haber terminado el desarrollo del Proyecto de Fin de Carrera, estamos en disposición de evaluar si se han cumplido los objetivos propuestos en el Capítulo 2.

Los dos objetivos principales que se perseguían eran el 2 y el 3. El 2 consistía en construir un framework orientado al desarrollo de videojuegos en dos dimensiones, el cual se ha presentado y explicado con detalle en la Parte II de este documento. El siguiente objetivo era crear un editor de niveles gráfico que, en conjunción con framework, permitiera desarrollar juegos de plataformas sencillos en 2D a partir de una GUI. Éste objetivo también se ha cumplido, como puede verse a lo largo de la Parte II de la memoria.

El objetivo 4 era secundario o derivado, podríamos decir que de validación de los anteriores. Consistía en desarrollar un pequeño juego de ejemplo haciendo uso exclusivamente del editor gráfico, para demostrar la utilidad de la herramienta. Se ha creado un juego compuesto por tres niveles, cuyo desarrollo se detalla paso a paso en el Capítulo 12.

Por último, el objetivo 1 era el más genérico e intangible, pues trataba de comprender las etapas del desarrollo de un juego a bajo nivel y adquirir destreza con C++ y SDL. El hecho de que el resto de objetivos se hayan cumplido prueba que éste también.

Puede decirse por tanto que todos los objetivos académicos establecidos al inicio del proyecto se han cumplido exitosamente, al margen de que se haya tardado bastante más de lo previsto.

Si hablamos del aprovechamiento personal del proyecto y del cumplimiento de las expectativas depositadas, en líneas generales la sensación es satisfactoria, aunque no sin altibajos. Se disfrutó mucho la primera mitad del proyecto, la puesta en contacto con el mundo del *game development* y el constante aprendizaje que esto supuso. Por momentos era abrumador ver la cantidad de nueva información que surgía al investigar sobre cualquier elemento que en principio podía parecer básico, pero a la vez era apasionante.

La evolución natural de alguien que se inicie en el desarrollo de juegos seguramente hubiera sido seguir mejorando el framework según el juego o juegos que se tuvieran en mente, sin embargo hubo que cortar de raíz para pasar al desarrollo del editor. Aunque tuvo mucho valor didáctico construir la herramienta, supuso una desconexión con lo anterior y un bache en la curva de aprendizaje, pues de descubrir nuevas mecánicas y posibilidades del desarrollo de juegos se pasó a tener que estudiar otra herramienta y otro ámbito no necesariamente relacionados con la motivación principal del proyecto: Qt y el desarrollo de interfaces de usuario. A pesar de esto, fue muy satisfactorio terminar la primera versión funcional del editor y ver cómo realmente podían construirse juegos divertidos y variados con él.

Mención aparte merece la integración del sistema de renderizado de SDL en la GUI de Qt. Fue sin duda el apartado técnico más complicado de llevar a cabo, hasta el punto de que en ocasiones se pensó que no se iba a conseguir. Visto con perspectiva, se considera que no mereció la pena el tiempo y el esfuerzo dedicado. Si se pudiera volver atrás, se realizaría el editor íntegramente en Qt, aunque supusiera no poder reutilizar las clases del framework para la visualización del nivel que se está editando. Tampoco se podría probar el nivel dentro del editor, una de las

características más llamativas de S2PEditor, pero se podrían adoptar soluciones alternativas como generar una ventana externa con SDL cada vez que se pruebe (como hacen algunos editores comerciales).

No se ha decidido si se continuará con el proyecto, pero de ser así los esfuerzos se centrarían en mejorar el framework, adaptándolo a algún juego en concreto que se quisiera crear. S2PEditor está bien para prototipado o para usuarios que no tengan conocimientos de programación y no les importen las limitaciones de la herramienta, pero evidentemente existen herramientas mucho más completas que hacen difícil justificar el uso de este editor. Incluso si se quisiera seguir usando el framework S2Dp, sería más potente y flexible generar los niveles con un editor de mapas como *Tiled* (aunque sólo se cree el archivo y no se pueda probar sobre la marcha) y modificar las clases del framework según lo que se quiera conseguir.

Capítulo 15

Trabajo futuro

Hay muchas limitaciones que saltan a la vista desde que se trabaja unos minutos con el editor. Las más obvias son relativas a los objetos que podemos usar; podemos cambiar su apariencia, velocidad y otros atributos, pero al usuario le asaltarán preguntas como: ¿qué hago si quiero añadir una animación para que el jugador se agache? ¿Y si quiero que un enemigo que dispare? ¿No puedo incluir una animación para cuando un enemigo muere? Y muchas más, a las que se sumarían las que no tengan que ver con los objetos.

Sin embargo, al hablar de las limitaciones, hay que hacer una distinción entre el framework y la GUI del editor. Si tenemos en cuenta sólo lo que se puede hacer con la GUI del editor, puede dar la impresión de que las características de los juegos están más limitadas de lo que realmente están. Por poner un ejemplo sencillo, desde el editor no se puede añadir una pantalla entre un nivel y otro, pero con conocimientos de programación, no es complicado añadir al framework un nuevo estado entre niveles e implementar la funcionalidad deseada.

A pesar de lo dicho en el párrafo anterior, es innegable que el software desarrollado es ampliable hasta el infinito, o hasta donde permitan los videojuegos. Al fin y al cabo este proyecto es sólo un primer contacto con el desarrollo de videojuegos a bajo nivel, nunca se pretendió que fuera una herramienta que permitiera crear juegos complejos.

Por todo lo comentado, es difícil hacer una lista de ampliaciones concretas que acoten el trabajo futuro; no obstante, se enumerarán algunas de las que se tomó nota durante el desarrollo del proyecto, tanto de funcionalidad como de rendimiento o de mejora en la calidad del código.

Hojas de sprites no uniformes

Las herramientas desarrolladas en el proyecto usan y soportan sólo hojas de sprites uniformes, es decir, en las que el tamaño de cada uno de los frames que las compone es el mismo.

La ventaja de las hojas de sprites no uniformes es que se podría aprovechar el espacio mejor y el dibujado sería más eficiente, ya que con las uniformes los frames que contienen los sprites más pequeños serán igual de grandes que el que contenga al sprite más grande.

Se descartó utilizar hojas de sprites no uniformes puesto que no es un añadido trivial. Añadir soporte para esto implicaría añadir, tanto en el framework como en la GUI, opciones para que se pueda especificar dónde se encuentra cada frame y qué frames componen cada animación. Además, ya no se podría utilizar la misma hitbox para todos los frames de la hoja de sprites, lo cual complicaría la implementación de las colisiones.

Realmente, el método que se usa en la industria es asociar a cada hoja de sprites un metafichero (XML, JSON) que contenga la información sobre las animacions y los frames contenidos en ella, por lo que la ampliación consistiría en implementar esto.

Hitbox de las tiles

Una vez que indicamos un tamaño de tile, todas las tiles sólidas que coloquemos son colisionables en toda su superficie, lo que haría que si usamos tiles cuya imagen no llene toda esta superficie, el jugador y el resto de objetos dieran la impresión de estar flotando sobre ella. Además esta limitación imposibilita la construcción de superficies de colisión curvas. Mejorar el sistema de colisión de los objetos con el escenario es una de las actualizaciones más importantes y de las que más complejas se prevén.

Mayor control del HUD

Ahora mismo sólo podemos elegir qué iconos representarán los cuatro elementos del HUD: salud, vidas, puntuación y llaves. No se pueden modificar desde el editor o desde el archivo con los datos del juego la fuente con la que se muestran los números, la disposición de los iconos en pantalla o el espacio entre ellos.

Estados con pantallas estáticas

En general, son muy mejorables los estados consistentes en una pantalla estática (inicio, pausado, *game over* y final del juego), y si se decidió dejar así es porque el objetivo principal (la edición de los propios niveles) ya estaba conseguido, y aunque se añadiera la posibilidad para poner botones o animaciones, el usuario iba a seguir estando limitado a la hora de decidir el flujo entre pantallas y niveles o estados.

Añadir mecanismos para alterar el flujo de los estados del juego desde la GUI o archivos externos supone una actualización extensa del software desarrollado.

Animaciones

Las animaciones son una de las partes más complejas técnicamente a la hora de desarrollar un juego, y existe muchísimo margen de mejora: que el salto del personaje tenga varios frames y se distinga la subida y la bajada, una animación distinta para correr que para caminar, que el *check point* y el objeto de final de nivel realicen una animación al ser alcanzados, y un largo etcétera.

Dirty Rectangles

Se podría usar la técnica con este nombre, consistente en detectar en cada frame qué áreas de la pantalla se han modificado y por tanto actualizar sólo ésas.

Component Pattern

A medida que el código se expanda y aparezcan más clases que toquen varios ámbitos, se valoraría usar el *Component pattern* en lugar de usar extensivamente la herencia[40].

Apéndice A

Manual de usuario

S2PEditor (*SDL 2D Platformers Editor*) es un editor de videojuegos de plataformas en dos dimensiones, cuyos niveles se construyen a partir de la combinación de escenarios y de objetos.

Los escenarios se componen de *tiles* y *backgrounds*, mientras que los objetos son el contenido dinámico del juego: enemigos, ítems coleccionables y otros elementos que, aunque presentan un comportamiento básico predefinido, admiten ciertas variaciones.

La variedad en la apariencia de estos escenarios y objetos no viene limitada por el editor; cada usuario decide qué aspecto tiene cada elemento utilizando sus propios recursos artísticos. De esta forma se permite la creación de juegos y niveles con estilos visuales muy diversos.

Para conocer las características de los juegos que se pueden crear y los controles del personaje, acudir a la Sección 6.3 del análisis de s2Dp.

A.1. Instalación

S2PEditor no requiere de un proceso de instalación. Los requisitos para ejecutarlo son un sistema operativo Windows y la carpeta S2PEditor con el ejecutable S2PEditor.exe y el resto de recursos. Al lanzar el ejecutable se nos presentará la interfaz gráfica del editor y podremos acceder a toda su funcionalidad.

A.2. Interfaz de usuario

Este apartado pretende ser una referencia esquemática de las acciones que pueden llevarse a cabo con el editor. Dichas acciones se han explicado en profundidad en el Capítulo 12; aquí sólo se señalará qué acción ejecuta cada elemento de la interfaz.

La interfaz se divide principalmente en cuatro ventanas: una principal con su menú principal y barra de herramientas, y tres acoplables para gestionar objetos, niveles y *tilesets*.

Menú principal y barra de herramientas

A continuación se muestran todas las opciones del menú principal separadas en tablas según su cabecera.

En la barra de herramientas hay un botón por cada opción del menú principal, exceptuando Project→Exit y Help→About. Cada botón tiene el mismo icono que acompaña al texto de la opción del menú principal que simboliza.

Project	Acciones que afectan al proyecto actual.
New	Crear un nuevo proyecto.
Open	Abrir un proyecto existente. Se mostrarán sólo los archivos con la extensión adecuada (.s2pe).
Save	Guardar los avances del proyecto actual.
Export	Exportar el proyecto a una carpeta portable.
Textures	Abrir gestor de texturas.
Sounds	Abrir gestor de sonidos.
Exit	Salir del programa.

Level	Acciones que afectan al nivel actual.
New	Añadir un nuevo nivel al proyecto.
Save as...	Guardar una copia del nivel actual como otro nivel del proyecto.
Run	Probar el nivel actual.
Stop	Dejar de probar el nivel y volver al modo edición.
Backgrounds	Abrir gestor de backgrounds.
Music	Elegir la música del nivel.

Game	Acciones relacionadas con el juego final.
Settings	Abrir gestor de opciones del juego.
Export	Exportar el juego a una carpeta portable.

Insert	Para cambiar el modo de inserción.
Tile	Cambiar el modo de inserción a tiles.
Object	Cambiar el modo de inserción a objetos.

Help	Información que puede resultar útil.
About	Muestra información básica del programa.

Ventanas acoplables

Al iniciar S2Peditor, además de la ventana principal se cargarán tres ventanas acopladas a ella: el **gestor de objetos**, el **gestor de tilesets** y el **gestor de niveles**. Por defecto se cargarán a la derecha, con el gestor de objetos en la parte superior y los gestores de tilesets y niveles solapados en la parte inferior, como se muestra en la Figura A.1. Podemos modificar su posición y acoplarlas debajo, encima o a la izquierda del mapa del nivel, o incluso separarlas de la ventana principal y dejarlas como ventanas individuales.

Figura A.1 – Vista general de la GUI del programa.

A.3. Distribución

Los proyectos y juegos creados con S2PEditor pueden ser ejecutados en un entorno distinto al que se crearon. En el caso de los juegos, se generan directamente en un directorio independiente preparado para ser distribuido; en el caso de los proyectos, debemos llevar a cabo unos pasos previos.

Proyectos

Los distintos proyectos que creemos se guardarán en el subdirectorio projects. Este subdirectorio está en el directorio raíz de S2PEditor desde el que hemos lanzado el ejecutable. Cada vez que creemos un proyecto nuevo podremos observar cómo dentro de projects se crea una nueva carpeta con el nombre del proyecto. Dentro de esta carpeta es donde tendremos que copiar los proyectos creados en otro entorno, pero antes tendremos que exportar el proyecto que queremos transportar.

Cada proyecto que creamos hace uso de una serie de recursos (generalmente imágenes) que se encuentran en el equipo en que estamos trabajando. Por esta razón no podemos simplemente copiar la carpeta que contiene un proyecto, llevarla a otro equipo y pegarla en S2PEditor/projects. Antes habría que copiar todos los recursos de los que hace uso el proyecto y cambiar la forma en la que nos referíamos a ellos, pues en el equipo original los recursos se encontraban en otro sitio. Todo este proceso se puede hacer automáticamente y es lo que llamamos exportar un proyecto.

Para exportar el proyecto que estamos editando podemos usar la opción del menú Project→Export o el botón de la barra de herramientas. Se nos pedirá un nombre para el proyecto exportado y un directorio en el que guardarlo. Para abrir el proyecto en otro equipo hay que seguir dos pasos:

1. Copiar la carpeta con el proyecto exportado en el nuevo equipo, dentro del directorio S2PEditor/projects.
2. Lanzar S2PEditor en el nuevo equipo y usar la opción del menú Project→Open o el botón de la barra de herramientas para abrir el proyecto. Se nos pedirá el archivo del proyecto, que tendrá el nombre que le hayamos dado al exportarlo y la extensión S2PE.

Juegos

En el caso de los juegos es mucho más sencillo, pues ya se generan teniendo en cuenta que se van a ejecutar de manera independiente por lo que, al contrario que los proyectos, no dependen de los recursos del equipo en que son creados.

La carpeta generada mediante la opción Game→Export (ver apartado 12.9) contiene el ejecutable game.exe y se puede transportar de un equipo a otro o ejecutarse desde un dispositivo de almacenamiento externo.

Una vez en el equipo destino, se recomienda cambiarle a game.exe el nombre por uno que identifique al juego que se ha creado, así como crear un acceso directo y ponerlo fuera de la carpeta, aislado del resto de archivos, para mayor comodidad a la hora de ejecutarlo.

Apéndice B

Recursos utilizados

Los recursos utilizados durante el desarrollo del proyecto son de libre uso. A continuación se enumeran todos, divididos por categorías. Merece una mención especial el portal *OpenGameArt*[1], con una extensa base de datos de la que se han extraído la mayoría de sprites y sonidos.

Sprites

Monkey Lad in Magical Planet, por Carl Olsson (a.k.a: surt).

Monsterboy in Wonder World, por Carl Olsson.

420 pixel art icons for RPG, por Henrique Lazarini (a.k.a: Ails)

arachnestuff, por Arachne.

Items and elements, por Jason *GrafxKid*.

Interaction Assets, por Madebyoliver de FlatIcon.

Essential Colection , por Madebyoliver de FlatIcon.

Interaction Assets, por Madebyoliver de FlatIcon.

Technology Elements, por Madebyoliver de FlatIcon.

Sonido

The Essential Retro Video Game Sound Effects Collection, por Juhani Junkala.

Random silly chip song, por Bart Kelsey.

House in a Forest Loop, por HorrorPen

A Journey Awaits por Pierre Bondoerffer (@pbondoer)

TrueType fonts

Boxy-Bold, por Clint Bellanger. Sitio web: http://clintbellanger.net.

Karma Suture, por Typodermic Fotns.

[1]https://opengameart.org/

Glosario

Arcade Término genérico para referirse a las máquinas recreativas.

Background Imagen que se ve al fondo del nivel, detrás de los objetos y el escenario.

Binding Adaptación de una biblioteca para que pueda ser utilizada con un lenguaje de programación distinto a aquellos que soporta de forma nativa.

Caja de colisión, *hitbox* Rectángulo que nos delimita la superficie colisionable de un objeto.

Checkbox Caja o casilla de verificación. Tiene dos estados, activada y desactivada.

Checkpoint Punto en el que aparecerá el jugador tras perder una vida, ahorrándose empezar desde el principio del nivel.

Colisionable, sólido Dícese de un objeto o tile que causa colisión al contactar con el resto, haciendo necesaria la gestión de colisiones.

Diálogo Ventana emergente en la interfaz gráfica de usuario, cuya función es normalmente informarnos de algún suceso o pedirnos información.

Flag Uno o más bits utilizados para almacenar valores binarios que tienen asociado algún significado.

FPS Tasa de frames de dibujado, expresada en frames por segundo.

Frame Cada una de las imágenes que, reproducidas en sucesión, simulan movimiento formando una animación.

FSM Del inglés *Finite State Machine*. Autómata finito que se usará para la gestión de los estados del juego.

Game loop Bucle principal del juego, consistente en recibir la entrada del usuario, actualizar y dibujar en pantalla.

Getters / Setters Métodos de una clase destinados a obtener y establecer el valor de sus atributos.

GUI Interfaz gráfica de usuario. De las siglas *Graphic User Interface*.

Género Tipo de videojuego. Los juegos que pertenecen a un mismo género comparten una serie de elementos esenciales.

HUD Del inglés *head-up display*. Panel que muestra la información sobre el estado del jugador.

IDE Entorno de desarrollo integrado. Software que normalmente incluye un editor de código, un compilador, un depurador y una GUI.

Indie Videojuegos desarrollados sin el apoyo de grandes distribuidoras, normalmente con pocos recursos económicos y humanos.

Joystick Mando o controlador para jugar a un videojuego.

Jugador, *player* Se utiliza para hacer referencia al avatar del jugador.

Meta Punto que, al ser alcanzado, finaliza con éxito el nivel actual.

Parsear De *parse* en inglés, proceso de analizar una secuencia de símbolos para determinar su estructura gramatical.

Parser Herramienta destinada a parsear archivos.

Paso Iteración completa del bucle de actualización del juego.

Renderer Estructura de datos que se encargará del proceso de renderizado de las texturas en pantalla.

Singleton Patrón de diseño que asegura que una clase sólo tiene una instancia, proporcionando un punto global de acceso a ella.

Spinbox Caja que contiene un valor numérico.

Sprite Representación gráfica de un objeto en el juego.

Spritesheet Series de frames de una o varias animaciones agrupados en una única imagen.

Tile Cada una de las porciones de las mismas dimensiones en que se divide una imagen mayor. Se utilizan para formar el escenario del juego.

Tileset Conjunto de tiles.

Bibliografía

[1] *Ofertas de trabajo en Gamasutra*. URL: http://jobs.gamasutra.com (visitado 15-10-2014).

[2] *Ofertas de trabajo en LinkedIn*. URL: https://www.linkedin.com/jobs (visitado 15-10-2014).

[3] Jean-loup Gailly y Mark Adler. *Licencia zlib*. URL: http://www.gzip.org/zlib/zlib_license.html (visitado 2014).

[4] *Sitio web de la biblioteca Allegro*. URL: http://liballeg.org (visitado 05-01-2017).

[5] Laurent Gomila. *Frequently Asked Questions (FAQ) en el sitio web de SFML*. URL: http://www.sfml-dev.org/faq.php (visitado 20-12-2016).

[6] Laurent Gomila. *Términos de la licencia de SFML*. URL: http://www.sfml-dev.org/license.php (visitado 20-12-2016).

[7] Dean Ellis. *From XNA to MonoGame*. Ed. por Game Developer Magazine from Gamasutra. 15 de mayo de 2013. URL: http://www.gamasutra.com/view/feature/192209/from_xna_to_monogame.php.

[8] *Sitio web oficial de Monogame*. URL: http://www.monogame.net (visitado 20-12-2016).

[9] *Microsoft Public License (MS-PL)*. URL: https://opensource.org/licenses/MS-PL (visitado 14-10-2014).

[10] Tim Sweeney. *If you love something, set it free*. Ed. por Unreal Engine. 2 de mar. de 2015. URL: https://www.unrealengine.com/blog/ue4-is-free.

[11] *Sitio web del Unity Engine*. URL: https://unity3d.com (visitado 05-01-2017).

[12] Lazy Foo. *Setting up SDL 2 on Visual Studio 2010 Ultimate*. URL: http://lazyfoo.net/tutorials/SDL/01_hello_SDL/windows/msvsnet2010u/index.php (visitado 2014).

[13] Sam Lamtinga y Mattias Engdegard. *Sitio web de SDL_Image 2.0*. URL: https://www.libsdl.org/projects/SDL_image (visitado 2014).

[14] *2D Accelerated Rendering*. URL: https://wiki.libsdl.org/CategoryRender.

[15] Lazy Foo. *Beginning Game Programming v2.0*. URL: http://lazyfoo.net/tutorials/SDL/index.php.

[16] *SFML and Visual studio*. URL: http://www.sfml-dev.org/tutorials/2.1/start-vc.php (visitado 2014).

[17] *Download SFML 2.1*. URL: http://www.sfml-dev.org/download/sfml/2.1 (visitado 2104).

[18] *Documentation of SFML 2.1*. URL: http://sfml-dev.org/documentation/2.1/classsf_1_1VideoMode.php (visitado 2014).

[19] Tutorials for SFML 2.1, ed. *Shapes*. URL: http://www.sfml-dev.org/tutorials/2.1/graphics-shape.php (visitado 2014).

[20] Tutorials for SFML 2.1, ed. *Sprites and Textures*. URL: http://www.sfml-dev.org/tutorials/2.1/graphics-sprite.php.

[21] RB Whitaker. *Monogame - Getting Started Tutorials*. URL: http://rbwhitaker.wikidot.com/monogame-getting-started-tutorials (visitado 2014).

[22] RB Whitaker. *Setting Up XNA*. URL: http://rbwhitaker.wikidot.com/setting-up-xna (visitado 2014).

[23] RB Whitaker. *Managing Content*. URL: http://rbwhitaker.wikidot.com/monogame-managing-content (visitado 2014).

[24] Wikipedia, ed. *List of games using SDL*. URL: https://en.wikipedia.org/wiki/List_of_games_using_SDL#Arcade_and_game-console_emulators_which_use_SDL (visitado 2014).

[25] Metacritic, ed. *Atom Zombie Smasher*. URL: http://www.metacritic.com/game/pc/atom-zombie-smasher (visitado 14-10-2014).

[26] *Stack Overflow*. URL: http://stackoverflow.com.

[27] *Game Development Stack Exchange*. URL: http://gamedev.stackexchange.com.

[28] Shaun Mitchell. *SDL Game Development*. 2013.

[29] Martin Fowler. *UML Distilled Third Edition*. 2003.

[30] Scott Meyers. *More Effective C++*. 1996.

[31] *Sitio web de SDL-widgets*. URL: http://members.chello.nl/w.boeke/SDL-widgets (visitado 24-12-2016).

[32] Mittwoch. *Sitio web de la biblioteca Game Gui*. 15 de jul. de 2015. URL: http://voxels.blogspot.de/2015/07/opengl-game-gui-widgets-with-source.html (visitado 24-12-2016).

[33] *Licencia de la biblioteca libRocket*. URL: http://librocket.com/wiki/license (visitado 24-12-2016).

[34] *Documentación de libRocket*. URL: http://www.librocket.com/wiki/documentation (visitado 24-12-2016).

[35] David Wimsey y col. *libRocket en GitHub*. URL: https://github.com/libRocket/libRocket (visitado 24-12-2016).

[36] Laurent Gomila. *Integrating to a Qt interface*. URL: http://www.sfml-dev.org/tutorials/1.6/graphics-qt.php (visitado 10-04-2017).

[37] WageIndicator 2017, ed. *Tusalario.es - Compara tu salario*. URL: http://www.tusalario.es/main/salario/comparatusalario# (visitado 05-05-2017).

[38] Ministerio de Empleo y Seguridad Social. *XVIII Convenio de Ingenierías*. Ed. por Boletín Oficial del Estado. 18 de ene. de 2017.

[39] CCOO, ed. *Nómina Convenio TIC: Tablas salariales 2017*. URL: http://bankintercomite.es/pag/bk/general/nomina/tic/index.php?conc=tablas.

[40] Robert Nystrom. *Game Programming Patterns*. 2014.