UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño de un motor de código abierto y tecnología web para el desarrollo de videojuegos de aventura gráfica

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: David Ponce Segura

Tutor: Xavier Molero

Curso 2015-2016

— Una vez tuve un perro más listo que tu.
— Te habrá enseñado todo lo que sabes.

Resumen

Este proyecto plantea la implementación completa de un motor y un sistema de configuración para la creación de videojuegos de tipo aventura gráfica. Esta tipología de videojuego se caracteriza por el uso del ratón como sistema de entrada, la resolución de puzles de lógica e ingenio por parte del usuario para superar los niveles y la narrativa en forma de historia. El motor consta de una serie de librerías orientadas a facilitar la creación de este tipo de juegos y esta construido con tecnologías web de código abierto con la finalidad de ser fácilmente portado a cualquier plataforma: ordenador de sobremesa, portátil, tableta o teléfono móvil.

Palabras clave: motor, código abierto, HTML, Javascript, WebGL, videojuego

Resum

Aquest projecte planteja la implementació completa d'un motor i un sistema de configuració per a la creació de videojocs de tipus aventura gráfica. Aquesta tipologia de videojoc es caracteritza per l'ús del ratolí com a principal sistema d'entrada, la resolució de puzles de lògica i ingèni per part de l'usuari per a superar els nivells i la narrativa en forma d'història. El motor consta d'una sèrie de llibreries orientades a facilitar la creació d'aquest tipus de jocs i esta construït amb tecnologies web de codi obert amb la finalitat de ser fàcilment portat a qualsevol dispositiu: ordinador de sobretaula, portàtil, tableta o telèfon mòbil.

Paraules clau: motor, codi obert, HTML, Javascript, WebGL, videojoc

Abstract

This project involves the complete implementation of an engine for creating point-and-click videogames. This type of game is characterized by using the mouse as the main input as well as using puzzle resolution and wit to complete levels. The engine consists of a set of libraries aimed at facilitating the creation of such games and is built with opensource web technologies in order to be easily ported to any device: desktop, laptop, tablet or smartphone .

Key words: engine, open source, HTML, Javascript, WebGL, videogame

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En este capítulo se abordan los motivos que nos han llevado a desarrollar este proyecto así como una breve descripción del mismo, haciendo hincapié en los objetivos fijados para su consecución. Además, se expone una visión general del contenido de la memoria y unas notas sobre el uso de la biografía.

1.1 Motivación

A finales de la década de los 80 y hasta mediados de los 90 de siglo XX, los videojuegos de tipo aventura gráfica vivieron su particular auge: la llamada «edad de oro» de las aventuras gráficas. La calidad narrativa y visual de muchos de aquellos títulos ha hecho que pasen a la historia de los videojuegos y que algunos tengan, aun hoy, un gran impacto en la cultura popular.

Durante los años siguientes, mientras la industria del videojuego crecía, las ventas del género de aventura gráfica decrecieron hasta casi desaparecer. Entre las principales causas de este declive se señalan: la aparición de nuevas plataformas que invitan a un juego más casual, como las tabletas y los teléfonos móviles, y la imposibilidad técnica de mejorar los motores existentes en aquel momento, ya que el *hardware* de la época había superado con creces las necesidades de este género [7].

En el momento del desarrollo de este trabajo existe en el mercado una ligera recuperación del género, sobre todo por parte de pequeños estudios de desarrollo, que han visto en el una forma de expresión artística cercana al cine o a la literatura, sin la complejidad técnica que implican otros géneros, como los que se apoyan necesariamente en gráficos tridimensionales.

Esto pone de manifiesto la existencia de un sector interesado en el desarrollo de videojuegos con estas características, quizás con dotes para la narración o la ilustración, pero que no siempre dispone de las herramientas necesarias para completarlo técnicamente. Cualquier persona con una idea para una aventura gráfica necesita, en principio, conocimientos avanzados en programación para llevarla a buen término.

En origen, la motivación personal detrás de este proyecto fue la creación de una única aventura gráfica, sin ánimo de lucro ni plan de distribución. Se llegó a desarrollar parte de la misma con un motor creado a medida y no extrapolable a

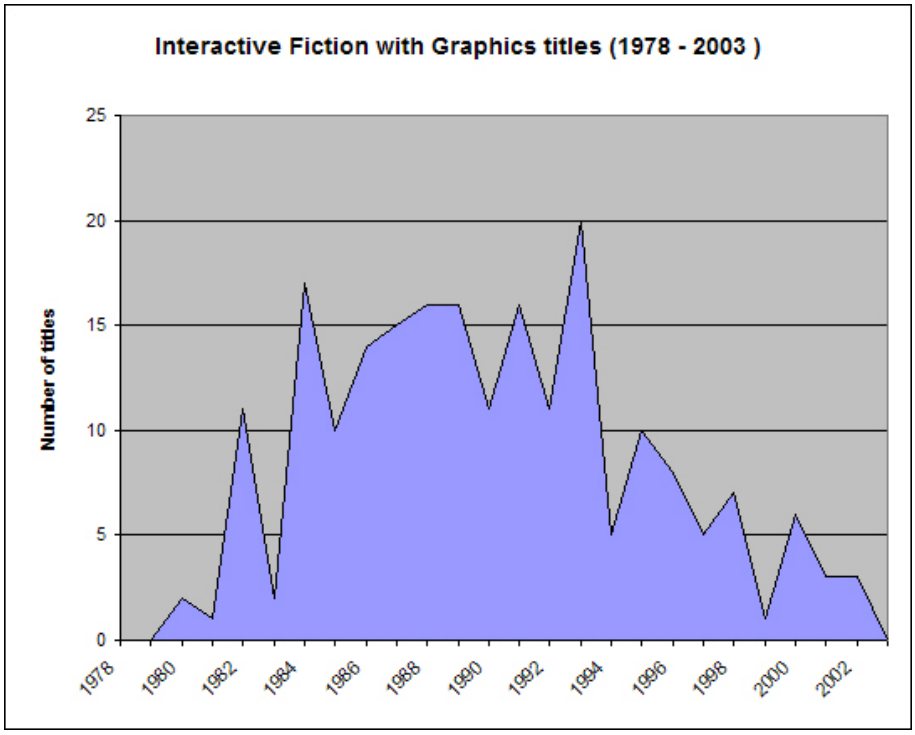

Figura 1.1: Publicación de títulos de aventuras gráficas por año. Desde 1978 hasta 2003. [11]

otros videojuegos. Más adelante se cambió el enfoque y se barajó la posibilidad de utilizar motores genéricos existentes en el mercado para, en un futuro, poder crear más aventuras gráficas con la misma estructura.

Analizando las herramientas existentes, de las que se describen las más importantes en la sección 3.1, se llegó a la conclusión de que, si bien hay algunas que, ofreciendo interfaces gráficas o simplificando las tareas de programación mediante librerías, pueden ayudar en las tareas de creación, no se ha encontrado ninguna que cumpla al mismo tiempo todos los requisitos objetivo en este proyecto: ser gratuita, de código abierto, comprenda el proceso de creación completo, sea multiplataforma y asequible sin conocimientos previos en programación.

1.2 Objetivos

El objetivo del presente trabajo es el diseño y la implementación de un conjunto de librerías o motor que permitan la creación de videojuegos de un género concreto: aventura gráfica, sin necesidad de conocimientos previos en programación por parte del usuario final; siendo además, el resultado, un *software* libre y multiplataforma, al estar construido con tecnologías web.

El motor está enfocado a una tipología concreta de videojuego por lo que, para implementarlo, primero debemos concretar cuales son las características que definen esta tipología. Podemos definir un juego como aventura gráfica si cumple las siguientes características:

- Resolución de puzles o problemas de lógica. [6]

- Narración o historia interactiva. [6][10]
- Exploración del entorno. [6]
- El jugador asume el rol de un personaje o héroe. [6][10]
- Recolección y manipulación de objetos. [6][10]

Con estos requisitos formales en mente, se propone que el motor creado sea capaz de implementar y manipular los siguientes conceptos dentro del sistema:

- Personajes: Tanto el personaje principal, que será controlado por el usuario, como el resto de personajes, controlados por el sistema, deben poder animarse para andar, hablar, coger objetos o cualquier otra acción que proponga el usuario al diseñar un videojuego usando este motor.
- Objetos: Se entienden por objetos aquellos elementos del escenario con los que el personaje principal puede interactuar. Al igual que los personajes, los objetos pueden estar animados o ser simplemente imágenes estáticas.
- Diálogos: Definimos como diálogo, dentro del sistema, a las estructuras en forma de árbol de decisiones que conforman un guión hablado entre dos personajes; es decir: dependiendo de las frases del guión escogidas por el jugador, la conversación puede avanzar por una rama u otra.

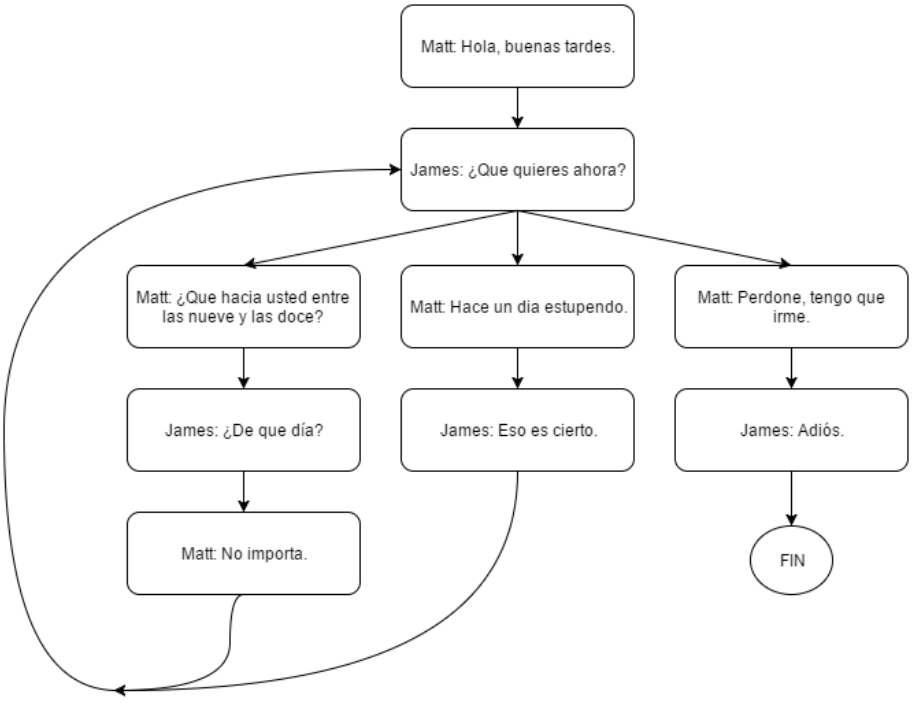

Figura 1.2: Esquema de un árbol de diálogo.

- Acciones: Llamaremos acciones (o verbos) a las diferentes opciones que se ofrecen al jugador para interactuar con los objetos del escenario. Clásicamente estas acciones solían comprender cosas como: Usar, coger, mirar, abrir, cerrar, empujar, hablar, etc. La tendencia en los últimos años ha sido reducir estas posibilidades a dos o tres, o incluso a una en ocasiones.

En todo caso, será el diseñador quien tome esta decisión. El motor estará preparado para crear tantas como se desee.

- Escenarios: Los escenarios componen las pantallas principales del juego y representan los lugares donde se encuentra el personaje en cada momento. El motor debe ser capaz de definir escenarios que, a su vez, contengan otros personajes y objetos. El personaje principal, controlado por el jugador, debe poder moverse por el escenario y pasar de un escenario a otro aunque, en ocasiones, moverse a otro escenario dependa de haber resuelto previamente un puzle.

- Inventario: Algunos de los objetos encontrados pueden ser almacenados por el personaje principal en su inventario para ser utilizados más adelante. Los objetos del inventario estarán siempre disponibles para el jugador y pueden utilizarse para interactuar con otros objetos o personajes, dando lugar a los puzles. Por ejemplo: Para abrir una puerta que de acceso a otro escenario es necesario que, previamente, el jugador tenga una llave en su inventario.

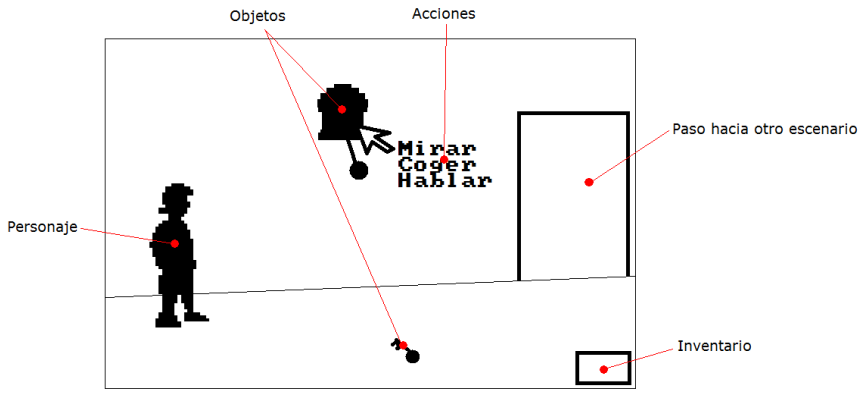

Figura 1.3: Esquema de un escenario con los diferentes elementos que lo componen.

En conjunto, el motor completo debe poder utilizarse para crear un videojuego sin necesidad de programación por parte del usuario final. Para ello, toda la definición del juego se establece mediante un único archivo de configuración en el que se definirán los escenarios, personajes, objetos y comportamientos. Este archivo estará estructurado mediante la especificación estándar para intercambio de datos JSON [8].

Pese a que el objetivo del proyecto es que el resultado sea utilizable sin conocimientos en programación, se plantea una interfaz de programación avanzada con la que, un usuario con mayores conocimientos, pueda personalizar el resultado añadiendo elementos y comportamientos no previstos por el motor.

Por último, aunque orientado a una característica fuera del objetivo de este proyecto, el archivo de configuración estará preparado para que sea posible la implementación de un editor gráfico que haga su manipulación más sencilla.

1.3 Estructura de la memoria

El presente trabajo se ha estructurado en 7 bloques diferenciados, comenzando por una aproximación al entorno de los juegos en general y las aventuras gráficas en particular y detallando después los elementos de desarrollo del motor objeto de este proyecto.

A continuación se resumen brevemente los contenidos de cada sección principal:

1. Introducción: Definición de las motivaciones que dan origen al proyecto así como los objetivos a cumplir.

2. Perspectiva histórica de las aventuras gráficas: Recorrido por la historia de las aventuras gráficas como género de videojuegos, desde su antecesor, la aventura conversacional, hasta las producciones actuales.

3. Recursos existentes: Análisis detallado de las diferentes tecnologías existentes dentro del alcance de este proyecto, proyectos similares y recursos disponibles.

4. Diseño e implementación del motor M4: Descripción del funcionamiento interno del motor, detallando sus partes y la correlación entre ellas.

5. Manual de uso del motor M4: Instrucciones de uso y ejemplos de configuración del motor para casos prácticos.

6. Conclusiones: Observaciones y ultimas consideraciones sobre el proyecto, una vez finalizado. Posibles mejoras y revisión de los objetivos.

7. Bibliografía y Apéndices: Documentación utilizada durante el desarrollo y documentos adjuntos referenciados en la presente memoria.

1.4 Notas sobre la bibliografía

Siendo el núcleo del proyecto el diseño e implementación de una librería de código, la mayor parte de la documentación consultada la componen las especificaciones técnicas de las diferentes tecnologías utilizadas. Las fuentes oficiales de las mismas son, generalmente, documentos on-line. Por este motivo, una gran parte de la bibliografía esta compuesta por documentos web. Referencias: [2], [3], [11].

Para situar el proyecto dentro de una perspectiva histórica, se han consultado algunos artículos de revistas especializadas y libros específicos del sector de las aventuras gráficas. Referencias: [5], [6]. [7]

Durante el transcurso de la implementación del motor, junto a la documentación oficial de desarrollo, han servido de apoyo distintas publicaciones sobre teoría de motores y diseño de videojuegos, todos ellos reflejados en la bibliografía adjunta a esta memoria. Referencias: [4], [8], [9], [10].

1.5 Terminología

El objeto del presente proyecto es la creación de un *software* intermediario, es decir, que sirva a sus usuarios para crear *software* para otros usuarios. Para evitar estas y otras confusiones entre términos a lo largo de la memoria, recopilaremos en esta sección las definiciones utilizadas en este texto.

- Usuario: Llamaremos usuario a la persona que utiliza el motor para crear un juego. También podemos referirnos a esta figura como diseñador, entendiendo que es la misma persona quien diseña que quien implementa el juego, aunque sabemos que esto podría no ser así en una situación real.
- Jugador: Nos referiremos siempre como jugador al usuario final que utiliza el juego resultante.
- Motor: El termino motor, en programación, se utiliza para definir la parte del código que gestiona la lógica de una aplicación, al margen de los datos. Este concepto no es exacto para referirnos al objeto del proyecto actual, pero lo utilizaremos para referirnos a el, siendo bastante aproximado.
- Código abierto/*Software* gratuito/*Software* libre: Existe cierta confusión general en el uso de estos términos de forma coloquial. Estrictamente hablando, el código abierto se refiere al código fuente de una aplicación que es visible y esta disponible para el usuario, no implicando que sea gratuito. Mientras que el *software* gratuito denota que este se distribuye sin cargo económico, pero no implica que su código tenga que estar disponible.

  El término *software* libre comprende los dos conceptos, refiriéndose a que el programa en si es gratuito y, además, el código fuente es de libre distribución.

  Aunque coloquialmente se utilizan indistintamente los tres términos para referirse al *software* libre,utilizaremos este último para referirnos al proyecto que nos ocupa.

CAPÍTULO 2
Perspectiva histórica de las aventuras gráficas

En este capítulo se expondrá una visión general del lugar que ocupan las aventuras gráficas en la historia de los videojuegos, desde sus antecedentes hasta la actualidad, pasando por la edad de oro y el declive a principios de este siglo.

2.1 Antecedentes

La aventura gráfica nace como género de videojuego a mediados de la década de 1980 y se caracteriza por alejarse de la acción de los arcade y simuladores predominantes en el mercado de aquel momento. La aventura gráfica como tal es una evolución de un género más antiguo: La aventura conversacional. Esta aventura conversacional es un tipo de videojuego basado totalmente en texto, consecuencia directa de las limitaciones gráficas de los primeros ordenadores personales.

En estos títulos, la acción se desarrollaba mediante un texto descriptivo en pantalla (en algunos juegos de la última época iba acompañado por unos sencillos gráficos estáticos). El jugador sabía donde se encontraba y que cosas había en su entorno gracias a una descripción textual y debía usar también comandos escritos para interactuar con el entorno. Como por ejemplo: Coger espada, Abrir puerta o Ir al norte.

El nacimiento oficial de la aventura conversacional data de 1975 con la creación de *Adventure* (o *ADVENT* por la limitación de 6 caracteres en los nombres de archivo de su sistema operativo original), programado por Will Crowther en un PDP-10. Aunque no fue hasta 1977 cuando el código original fue re-descubierto por Don Woods, expandido y difundido a través de ARPAnet, dando lugar a la aparición de un gran número de aficionados y muchos más videojuegos de este género. [5]

En España se publicó en 1988 una adaptación del *Adventure* de Woods con el nombre de *La Aventura Original*. El juego fue un éxito de ventas y logró un considerable aumento de los aficionados a este nuevo tipo de videojuegos en nuestro pais. Le seguirían muchos más títulos míticos como *La princesa* (1985), *Don Quijote de la Mancha* (1987) o *Abracadabra* (1988).

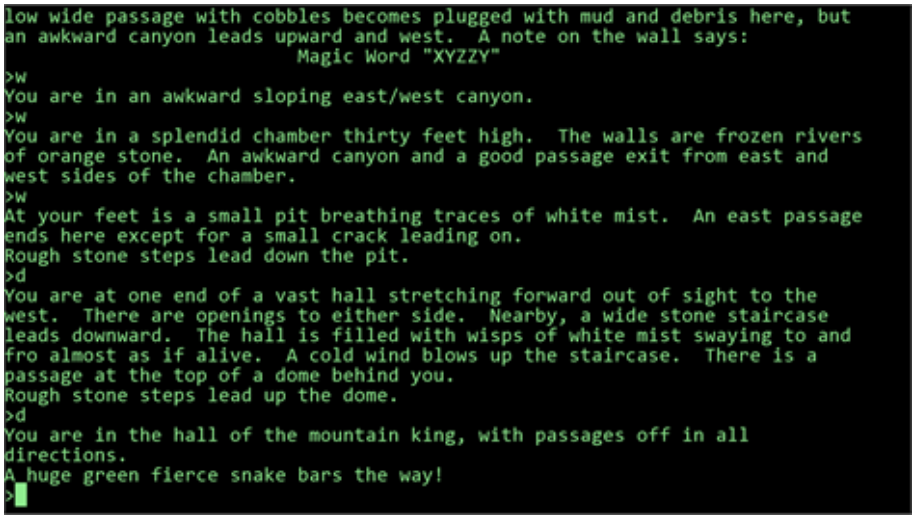

Figura 2.1: *Adventure* (1977). Will Crowther

Pese al éxito de estos títulos, el sistema de juego a base de escritura de comandos era considerado engorroso ya que, un simple error al teclear, o no escribir un comando con las palabras exactas, podía ocasionar que un puzle fuese irresoluble, causando frustración en el jugador. Por ejemplo: escribir «mirar reja» en lugar de «examinar reja» era suficiente para que el sistema no diese la acción por válida. Para resolver este problemas, en cuanto los ordenadores personales fuero técnicamente capaces, aparecieron las primeras interfaces gráficas y, con ellas, las aventuras gráficas.

2.2 Edad de oro

El término aventura gráfica se emplea para describir un género de videojuego, similar a las antiguas aventuras conversacionales, con dos principales diferencias: 1) Los entornos ya no se describen mediante texto si no que toda la responsabilidad recae sobre los gráficos. Lo que el usuario puede ver en pantalla es lo que hay disponible en el entorno del personaje. 2) Las acciones no se escriben con el teclado si no que son seleccionadas de un menú, normalmente utilizando el ratón.

El primer videojuego dentro de este género se considera *King's Quest* (1984), de la compañía Sierra Online. Aunque en este título las acciones aún deben ser escritas por el jugador utilizando el teclado, la pantalla muestra el entorno, el personaje principal y los objetos disponibles, sin necesidad describirlos textualmente. No sería hasta principios de los 90 cuando Lucas Arts introdujo la selección de acciones mediante el ratón, eliminando la necesidad de teclear y cambiando el modo de crear aventuras gráficas desde ese momento [7].

Durante la segunda mitad de la década de los 90, Sierra Online y Lucas Arts fueron las compañías detrás de las aventuras gráficas más importantes del momento con títulos como: *Leisure suit Larry* (1987), *Police Quest* (1988), *Maniac Mansion* (1987) o la que es considerada por muchos no solo la mejor aventura gráfica, si no también el mejor videojuego *The secret of Monkey Island* (1990). Producidas

Figura 2.2: *King's Quest I* (1984). Sierra

por estudios españoles durante esos años podemos destacar: *Igor Objetivo Uikokahonia* (1994), *Hollywood Monsters* (1997) o *Runaway: A Road Adventure* (2001).

Los últimos títulos de la llamada «edad dorada» se caracterizaron por interfaces gráficas basadas en iconos y gráficos y animaciones de alta calidad. Muchas de ellas contaban con diálogos interpretadas por actores de doblaje y animaciones propias de producciones cinematográficas. Habían alcanzado los límites tecnológicos del el género.[7]

Figura 2.3: *The secref of Monkey Island* (1990) Lucas Arts

2.3 Declive y maduración

A principios de los años 2000, los ordenadores personales continuaban mejorando pero, las aventuras gráficas, ya no podían aprovechar más esas mejoras y,

no pudiendo atender a la renovación constante que demandaban los aficionados en esa materia, la industria dejó de mostrar interés por el género [7]. A principios del siglo XXI eran ya pocas las distribuidoras que apostaban por las aventuras gráficas.

Como ha ocurrido con otros géneros en el mundo de los videojuegos, tras el auge inicial y el declive, las aventuras gráficas se encuentran ahora en una época de maduración en la que, pese a no ser tan populares como durante su época de crecimiento, han encontrado un lugar en el mercado del videojuego comercial.

La aparición de nuevos sistemas de creación digital que aceleran y facilitan notablemente la creación de aplicaciones, unido al nuevo mercado de las aplicaciones móviles, ha hecho que muchos estudios pequeños o desarrolladores independientes, hayan entrado con fuerza en el mundo de la programación de videojuegos.

Plataformas como Steam[1], para la distrubución y promoción de videojuegos, o sistemas completos para diseño y desarrollo como Unity3D[2], han creado una comunidad de desarrolladores entre los que las aventuras gráficas vuelven a estar presentes.

También estudios grandes como Double Fine, fundada por Tim Schafer, uno de los responsables de *Monkey Island* o Daedalic, empresa con base en Alemania especializada en el género, han vuelto a apostar fuerte por las aventuras gráficas con excelentes resultados. Como ejemplo: las reservas del juego *Broken Age* (2014) de Double Fine, superaron los dos millones de dólares antes incluso de salir al mercado.

Figura 2.4: *Deponia* (2012). Daedalic Entertainment

[1] http://store.steampowered.com
[2] https://unity3d.com

CAPÍTULO 3
Recursos existentes

Antes de comenzar con el desarrollo del motor que nos ocupa, analizaremos las opciones de proyectos similares existentes en el mercado, así como las tecnologías implicadas en el desarrollo, comparando las ventajas y desventajas de cada una de ellas para concluir la mejor forma de abordar el proyecto.

3.1 Motores actuales para aventuras gráficas

En esta sección analizaremos los motores para la creación de videojuegos existentes actualmente, tanto gratuitos como comerciales, que puedan proporcionar los requisitos que buscamos. Aunque existen motores, librerías y sistemas completos para la creación de videojuegos de diferentes tipologías, nos centraremos en los que están enfocados concretamente al género de aventura gráfica. Los parámetros principales que se han comparado en el estudio son:

- El sistema operativo para el que están disponibles. El ideal sería un sistema multiplataforma de base o que pueda compilar para diferentes sistemas.
- La licencia bajo la que se distribuyen. En este caso, se valorará si el codigo es abierto, gratuito o ambas cosas.
- La dificultad de uso para usuarios sin conocimientos previos.
- El sistema operativo en el que correrán los juegos resultantes.

La selección se ha realizado basándose en índices de popularidad obtenidos en redes sociales, buscadores y foros especializados[1]. El análisis de los 4 más relevantes según estos criterios es el siguiente:

3.1.1. Adventure Game Studio

http://www.adventuregamestudio.co.uk

También conocido por sus siglas AGS es, probablemente, el más completo y profesional de entre los analizados. Posee una interfaz gráfica muy completa e

[1] http://creandoaventuragrafica.blogspot.com.es

intuitiva. Puede utilizarse sin ningún conocimiento en programación e incorpora además herramientas de animación 2D, edición de audio y todo lo necesario para crear un videojuego completo partiendo de cero.

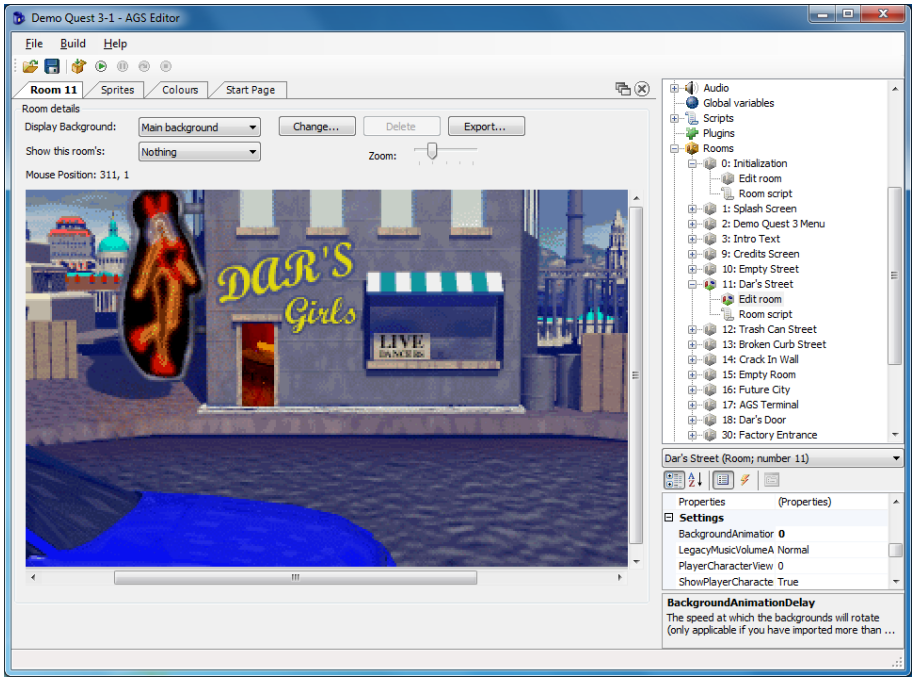

Figura 3.1: Entorno de desarrollo de *Adventure Games Studio*

Por otra parte, los juegos resultantes solo funcionan bajo el sistema operativo Windows tras ser compilados por el programa, aunque en las últimas versiones añadieron opciones, con un coste adicional, para compilar bajo otros sistemas operativos.

El programa era originalmente gratuito, salvo algunas funcionalidades. El código del programa fue liberado totalmente en 2011 así que es posible que evolucione hacia un sistema abierto a otras plataformas, aunque esto dependerá totalmente del soporte que reciba por la comunidad. Existen versiones para Linux y MacOS, pero a fecha de creación de este trabajo, están muy desactualizadas con respecto a la version para sistemas Windows.

3.1.2. WinterMute Engine

http://dead-code.org/home

Parecido en muchos aspectos a AGS, posee una interfaz gráfica muy completa y abarca todos los procesos de creación de la aventura gráfica. Menos maduro que AGS, quizás por llevar menos tiempo en el mercado, pero creado originalmente bajo licencia MIT[2].

La dificultad de uso es baja, al estar orientada al usuario medio y no requiere conocimientos previos en programación. El programa solo está disponible para

[2]La licencia MIT es una de tantas licencias de *software* originada en el Instituto Tecnológico de Massachusetts, caracterizada por promover *software* de libre distribución y modificación.

sistemas operativos Windows y los juegos resultantes correrán únicamente bajo este sistema operativos.

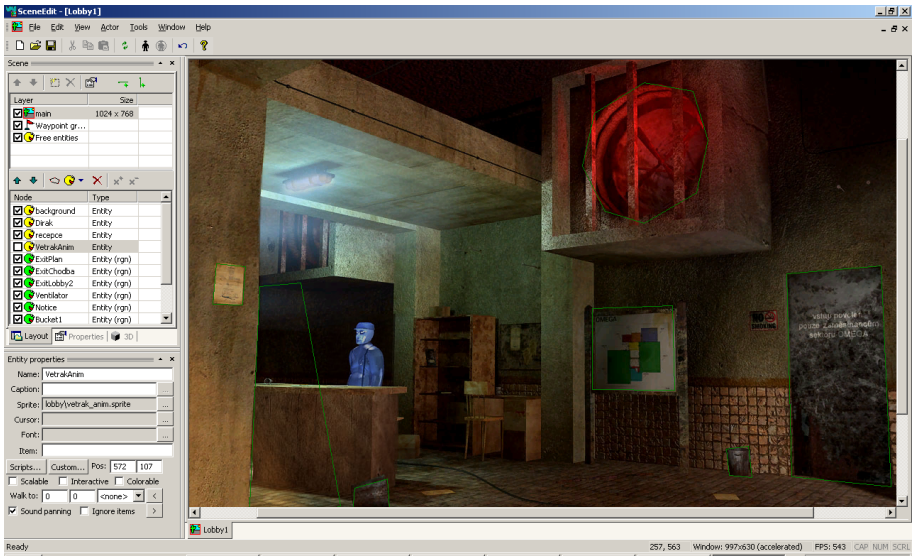

Figura 3.2: Entorno de desarrollo de *Wintermute Engine*

Una pequeña ventaja respecto al resto de motores analizados en este estudio es que soporta lo que han llamado 2.5 dimensiones. Esto es: escenarios en dos dimensiones y personajes en tres dimensiones.

3.1.3. Alpaca

http://www.alpacaengine.com

Alpaca es un motor para aventuras gráficas creado desde cero con el lenguaje para aplicaciones multimedia de Adobe: Flash AS3. No posee interfaz gráfica propia, pero se sirve de la interfaz del editor de Flash para facilitar la edición al usuario. El estar basado en esta tecnología le permite manipular los archivos multimedia de una forma muy eficiente. El propio sistema Flash, a diferencia de otros lenguajes, incorpora de serie funcionalidades y herramientas muy avanzadas para crear y editar animaciones 2D y 3D, así como edición de audio y vídeo.

El motor en si es de código abierto y gratuito, pero tanto el motor como los juegos resultantes requieren del complemento de Adobe Flash para funcionar que, aunque también es gratuito, es código cerrado y propiedad de Adobe.

Tanto el motor como los juegos resultantes funcionarán en cualquier plataforma que soporte el complemento de Adobe Flash. En el momento de la redacción de este trabajo, los principales sistemas operativos lo soportan: Windows, Linux y MacOS, aunque su uso esta siendo abandonado por los fabricantes de *software* paulatinamente. Así mismo, este motor está actualmente descontinuado por su desarrollador, por lo que no recibirá más actualizaciones.

3.1.4. Javascript Graphic Adventure Maker

http://jsgam.sourceforge.net

Figura 3.3: Captura del juego de ejemplo del motor *Alpaca*

Javascript Graphic Adventure Maker o JGAM es el candidato más aproximado, conceptualmente, al objetivo del presente proyecto.

Está desarrollado íntegramente en lenguaje JavaScript, por lo que puede ser utilizado prácticamente en cualquier sistema operativo y dispositivo moderno, incluyendo dispositivos móviles. Los juegos resultantes son también JavaScript, por lo que no tienen restricciones de portabilidad.

Figura 3.4: Captura de un juego realizado con *Javascript Graphic Adventure Maker*

El motor es gratuito y de código abierto, aunque está muy poco documentado, lo cual dificulta mucho su uso. En este caso, y a diferencia de los anteriores, si que requiere de ciertos conocimientos en programación para su correcto uso y el estilo de los juegos resultantes es menos personalizable que con el resto de motores analizados.

3.1.5. Conclusión

En la siguiente tabla se resumen las principales características de los 4 motores analizados.

	S.O. del motor	Licencia	Dificultad	S.O. del juego
AGS	Windows	AGS/Free	BAJA	Windows
WinterMute	Windows	MIT	BAJA	Windows
Alpaca	Windows/Mac	MIT	MEDIA	Multiple con VM
JSGAM	Multiple	OpenSource	ALTA	Multiple nativa

Tabla 3.1: Comparativa de motores para aventuras gráficas

Tras el estudio de los principales motores podemos observar que existen algunos muy potentes y completos, pero que presentan muchos problemas de portabilidad entre sistemas y no se adaptan fácilmente a dispositivos móviles. Otros, más abiertos y flexibles, son fácilmente portables pero presentan una complejidad de operación mucho mayor para usuarios poco experimentados.

Este proyecto pretende aunar de la mejor forma posible las principales ventajas de ambos extremos, creando un motor flexible y portable a la vez que asequible para usuarios no especializados.

3.2 Tecnologías implicadas en el desarrollo

3.2.1. JavaScript

JavaScript es un lenguaje de programación derivado del estándar ECMAScript. Se utiliza principalmente como parte de los navegadores web, permitiendo mejoras en la interfaz de usuario y la generación de páginas web dinámicas. Todos los navegadores actuales soportan código JavaScript de forma nativa, incluyendo los navegadores para dispositivos móviles.

Otra ventaja de utilizar JavasScript como lenguaje de desarrollo es que los videojuegos resultantes pueden distribuirse en archivos que funcionan de forma independiente, o bien servirse de forma dinámica a través de la web sin que el jugador tenga que tener previamente el juego descargado para empezar a jugar.

3.2.2. WebGL y Canvas

Canvas es el nombre que recibe un elemento incorporado en el lenguaje de maquetación de páginas web HTML a partir de la versión 5 (HTML5), y cuya función es permitir la generación de gráficos de forma dinámica en la ventana del navegador[4].

Es posible acceder al objeto Canvas a través de JavaScript, lo cual permite crear imágenes de forma interactiva y manipular gráficos y animaciones para que reaccionen a eventos del sistema o del usuario.

Por otra parte, WebGL es una especificación que permite mostrar gráficos 2D y 3D acelerados por hardware en páginas web. Mediante WebGL es posible acceder de forma nativa al sistema OpenGL ES 2.0 que está siendo incorporado actualmente en la mayoría de los navegadores. Esto se traduce en una gestión de gráficos de muy alto rendimiento dentro de la ventana del navegador web, que hasta hace poco se había utilizado únicamente para mostrar texto e imágenes.

Las dos tecnologías, Canvas y WebGL, proporcionan servicios gráficos similares, siendo WebGL mucho más potente, en términos de velocidad de actualización, al utilizar instrucciones directas de la tarjeta gráfica del sistema. Como inconveniente, WebGL depende del soporte del navegador en que se ejecuta. Actualmente, la mayoría de navegadores de escritorio lo soportan y se esta empezando a incorporar también en dispositivos móviles.

Ya que uno de los objetivos principales a cumplir por el motor es la portabilidad al máximo número de plataformas posible, se ha optado por el uso de estas tecnologías web para su desarrollo. A fecha de Junio de 2016, un 84.04 % de dispositivos soportan WebGL de forma nativa mientras que un 94.69 % soporta Canvas [3].

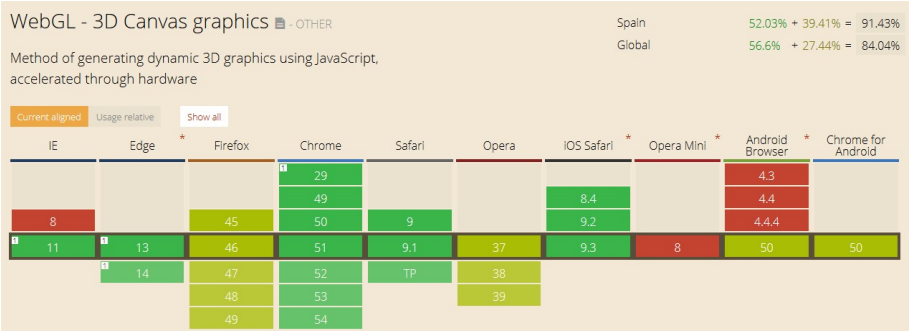

Tabla 3.2: Navegadores con soporte WebGL. Junio de 2016.
http://caniuse.com/#search=webgl

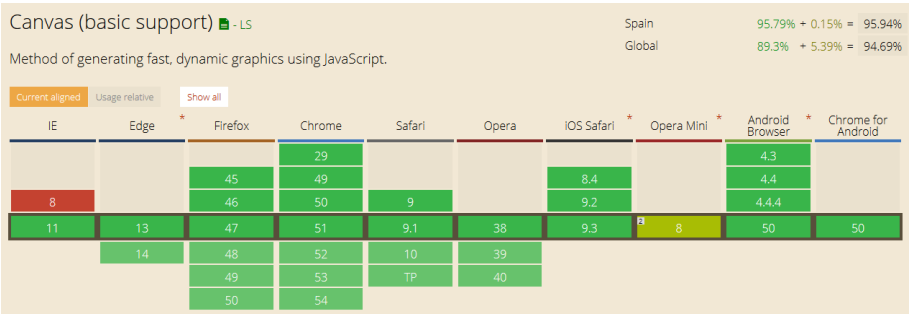

Tabla 3.3: Navegadores con soporte Canvas. Junio de 2016.
http://caniuse.com/#search=canvas

[3]Segun datos de la web http://caniuse.com/#search=canvas

3.3 Librerías gráficas

Siguiendo con el objetivo del presente trabajo sobre la implementación de un motor para aventuras gráficas que aúne las mejores características de cada uno de los motores analizados en la sección 3.1, se ha optado por el uso de una librería gráfica preexistente que facilite las tareas de gestión de imagen, diseñando y codificando el resto del conjunto desde cero.

Para ello, debemos escoger una librería de gestión de gráficos que funcione bajo JavaScript y que cumpla una serie de requisitos para encajar en el proyecto: ser gratuita y de libre distribución, para no restringir la licencia de uso final del proyecto resultante, y limitarse a la gestión gráfica, sin exceder un cierto tamaño.

A continuación se analizan las librerías gráficas actuales más relevantes compatibles con JavaScript y que cumplan los requisitos mencionados.

3.3.1. Impact

http://impactjs.com

Impact JS es una librería completa y de orientación comercial destinada a la implementación de juegos multiplataforma. Soporta sonido y gráficos en 2 y 3 dimensiones. Está extensamente documentada y posee características especificas para la creación de juegos de tipo plataformas. Además, es la única del conjunto analizado que posee herramientas propias de depuración y compilación. No es gratuita. La licencia de uso básica cuesta al rededor de 99 dólares por usuario.

3.3.2. enchantJS

http://enchantjs.com

EnchantJS es la más sencilla de utilizar de entre las analizadas, aunque también la más limitada en cuanto a funcionalidad. Es muy ágil para el desarrollo de pequeños proyectos y su sintaxis recuerda mucho al lenguaje ActionScript de Adobe, por lo que puede ser adecuada para desarrolladores que vengan de trabajar en ese entorno. Soporta gráficos en 2D y sonido. No incorpora características concretas para ninguna tipología de juego, aunque si algunas ayudas para juegos de tipo plataformas y RPG[4]. Está bien documentada y se distribuye bajo la licencia MIT.

3.3.3. Phaser

http://phaser.io

Phaser es, probablemente, la mas completa entre las librerias gratuitas. Soporta gráficos 2D, sonido y dispositivos de entrada tales como *joysticks* o mandos de

[4]Role Playing Game. Tipología de videojuego caracterizada por su estética usualmente en vista cenital.

juego. Incorpora herramientas para la gestión de animaciones, sistemas de partículas y efectos de vídeo. Además, su funcionalidad puede ampliarse mediante el uso de extensiones, algunas de ellas con coste económico. Esta desarrollada como un motor, más que como una simple librería; esto quiere decir que posee muchas funcionalidades especificas prefabricadas para videojuegos concretos, sobre todo de tipo plataformas.

3.3.4. Pixi.js

http://www.pixijs.com

Pixi.JS es una librería centrada exclusivamente en la representación y gestión de gráficos en pantalla. A diferencia de otras, no proporciona ningún tipo de soporte para el sonido ni posee estructuras específicas para animaciones, mapas o cualquier otro elemento específico de videojuegos. En cuanto a la gestión de gráficos, en su página web se declara como «la más rápida». Está extensamente documentada y tiene un tamaño muy reducido por la funcionalidad que ofrece. Se distribuye bajo licencia MIT y sigue recibiendo actualizaciones periódicas de sus desarrolladores.

3.3.5. Conclusión

Basándonos en las necesidades del proyecto, buscamos una librería que ofrezca una buena gestión de la salida gráfica y la posibilidad de utilizar indistintamente Canvas y WebGL. A esto añadimos el requisito de que sea gratuita y de código abierto. Por último, tendremos en cuenta el nivel de especialización para el que esté diseñada, es decir: usando una librería muy potente, pero orientada a la creación de videojuegos de tipo plataformas, estaríamos desperdiciando gran parte del potencial y tamaño, lo cual no es recomendable.

Librería	Plataforma	Licencia	Dificultad	Especialización
Impact	Multiple	Comercial	BAJA	ALTA
enchantJS	Multiple	MIT	BAJA	BAJA
Phaser	Multiple/Mac	MIT	MEDIA	MEDIA
Pixi.js	Multiple	MIT	MEDIA	BAJA

Tabla 3.4: Comparativa de motores para aventuras gráficas

Según las características analizadas, decidimos trabajar con Pixi.js, por ser la menos especializada de entre las que cumplen todos los requisitos anteriormente mencionados. Esto permitirá mayor flexibilidad de programación, no teniendo que adoptar estructuras de terceros y manteniendo el tamaño del código al mínimo.

CAPÍTULO 4
Diseño e implementación del motor M4

Este capítulo presenta con detalle todas las características implementadas en el motor, así como la estructura interna que las soporta, analizando cada uno de los objetos JavaScript que dan forma al sistema. Las características del motor se enumeran desde un punto de vista funcional para el usuario y la estructura interna se analiza desde una perspectiva más técnica.

4.1 Características

4.1.1. El nombre M4

Antes de abordar la idea de un motor genérico para venturas gráficas, se había comenzado con el desarrollo de una aventura gráfica independiente, con un motor gráfico y lógico propio y no extrapolable a otros juegos. El nombre del protagonista de esta historia era Matt Murphy y el lugar en que se desarrollaba en aquel momento era la planta baja de unas oficinas a la que llamaban «la mazmorra». Cuando, mas tarde, se decide crear un motor para el desarrollo de juegos de este tipo, se elije el nombre de: Motor de la Mazmorra para Matt Murphy, o MMMM, o M4.

4.1.2. Funciones del motor

Más que un motor en el sentido estricto, los cuales se encargan de gestionar exclusivamente la lógica del juego, con este proyecto se pretende crear un sistema completo para el desarrollo de videojuegos de tipo aventura gráfica, minimizando o eliminando por completo la necesidad de programar por parte del usuario. El sistema puede utilizarse para crear juegos de éste tipo con tres niveles de personalización o detalle, según el grado de especialización al que se desee llegar:

1. Sin ningún conocimiento en lenguajes de programación puede crearse un videojuego completo, con la funcionalidad prevista por el motor. Estos juegos son configurables por el usuario, aunque tienen ciertas limitaciones en

su diseño. El usuario, debe aportar todos los archivos multimedia necesarios (personajes, objetos, fondos) y definir la historia, los diálogos y el comportamiento mediante un archivo de configuración con un formato concreto.

El sistema es capaz de crear los escenarios, objetos y personajes y permitir al jugador mover al personaje principal a través de los mismos, cogiendo, usando y manipulando objetos para resolver los puzles que el diseñador haya planteado para avanzar.

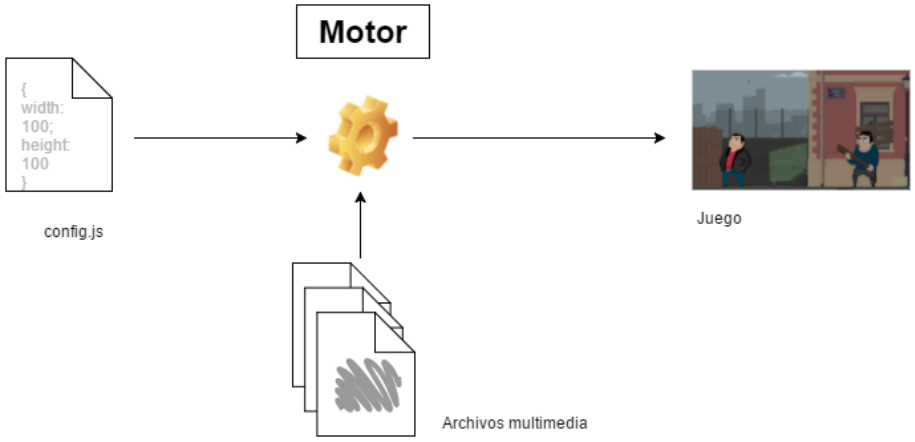

Figura 4.1: Diagrama de composición del motor y archivos de configuración

2. Además de todo lo mencionado en el punto anterior, si el diseñador tiene algun conocimiento en programación, puede ampliar la funcionalidad del motor y añadir comportamientos no previstos, haciendo los juegos más variados y personalizados. Para este propósito existen ciertas zonas en el documento de configuración en el que se permite la inserción directa de código JavaScript.

   Estos pequeños fragmentos de código serán analizados por el motor e incorporados a la funcionalidad del juego en el momento en que sean necesarios de forma dinámica.

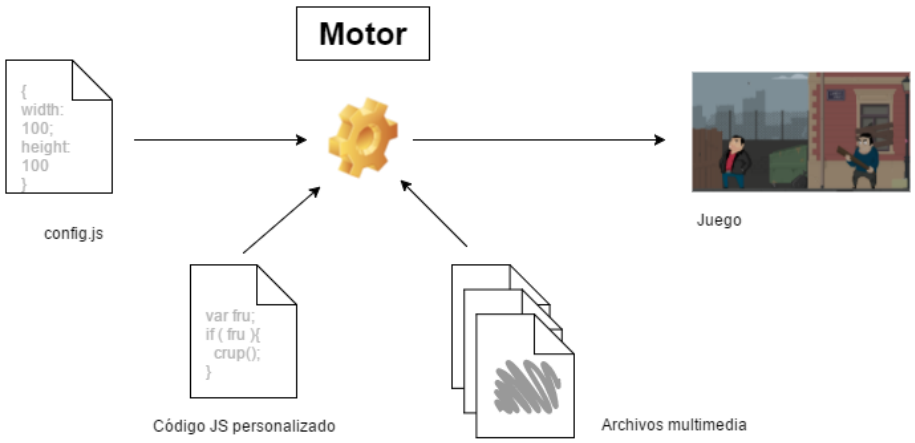

Figura 4.2: Diagrama de composición del motor y archivos de configuración, incluyendo código personalizado

3. Como tercer nivel de personalización, y por la propia naturaleza de código abierto de este proyecto, se permite la modificación del mismo a nivel de núcleo. Esto requiere de conocimientos avanzados tanto de programación como de la arquitectura interna del motor. Por otro lado, esto permite su ampliación, mejora y especialización sin nunguna restricción más que los límites físicos de la máquina que lo soporte.

4.2 Estructura interna

El motor esta construido mediante el modelo de programación orientado a objetos [9], en lenguaje JavaSrcipt y utilizando WebGL para la salida gráfica en pantalla. En caso de que el dispositivo no soporte WebGL, las instrucciones gráficas se redirigen automáticamente al sistema Canvas, más antiguo y lento, pero soportado por un mayor número de dispositivos.

Uno de los retos al desarrollar un sistema multiplataforma es que, a priori, es difícil prever la configuración de *hardware* que vamos a encontrar en el entorno de uso final. En el caso de una aplicación gráfica, como un videojuego, la gestión de la salida gráfica es muy importante para garantizar la usabilidad del producto.

La mayoría de dispositivos actuales, ya sean ordenadores de sobremesa, tabletas o teléfonos móviles, soportan WebGL, un sistema de representación gráfcia aceleraada por *hardware*. Por otro lado, aún quedan otros muchos dispositivos que no lo soportan y utilizan en su lugar Canvas.

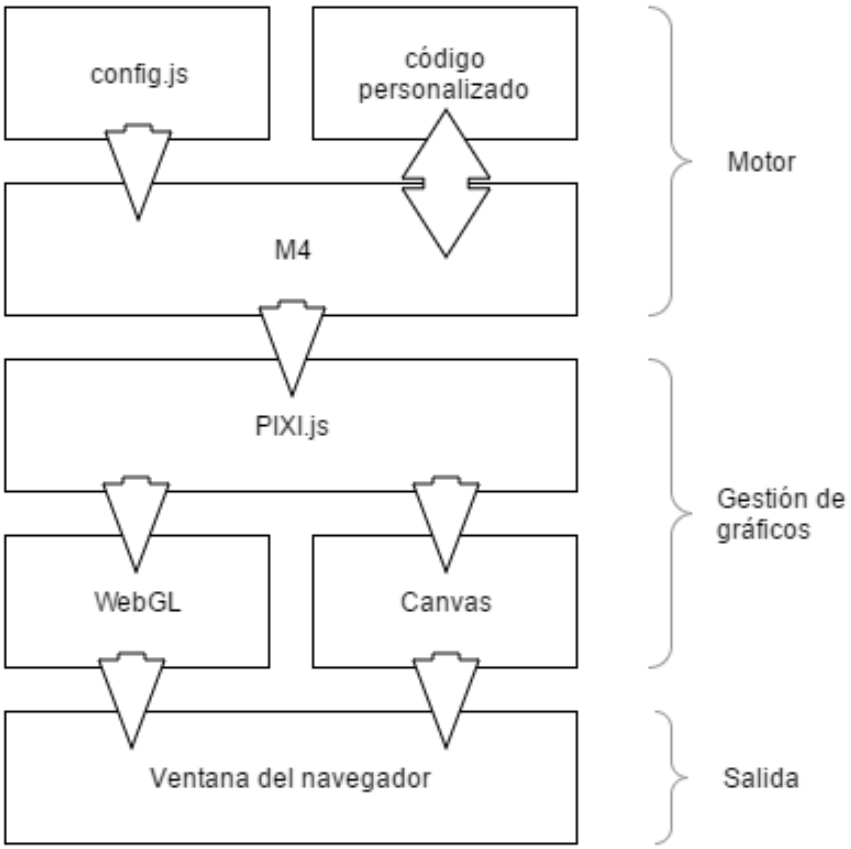

Figura 4.3: Diagrama de capas de la aplicación

El género de aventura gráfica es, precisamente, uno de los géneros que menos potencia de cálculo o gráfica requieren del dispositivo en el que corren, aunque sería deseable poder utilizar WebGL siempre que sea posible.

Como se ha mencionado en la sección 3.3.5, se ha utilizado la librería Pixi.js para confiarle la gestión de la salida gráfica. Esta librería se encarga de recoger las instrucciones de dibujo del motor y enviarlas, si es posible, al sistema WebGL y, si no está disponible, a Canvas. Esta gestión se hace de forma totalmente transparente para el programador.

El sistema cuenta con 8 clases principales, además de otras muchas de apoyo, las cuales se analizarán con detalle en las siguientes secciones de este capítulo. La carga de la gestión recae sobre la clase *Core*, que será la encargada de crear y administrar el resto de objetos.

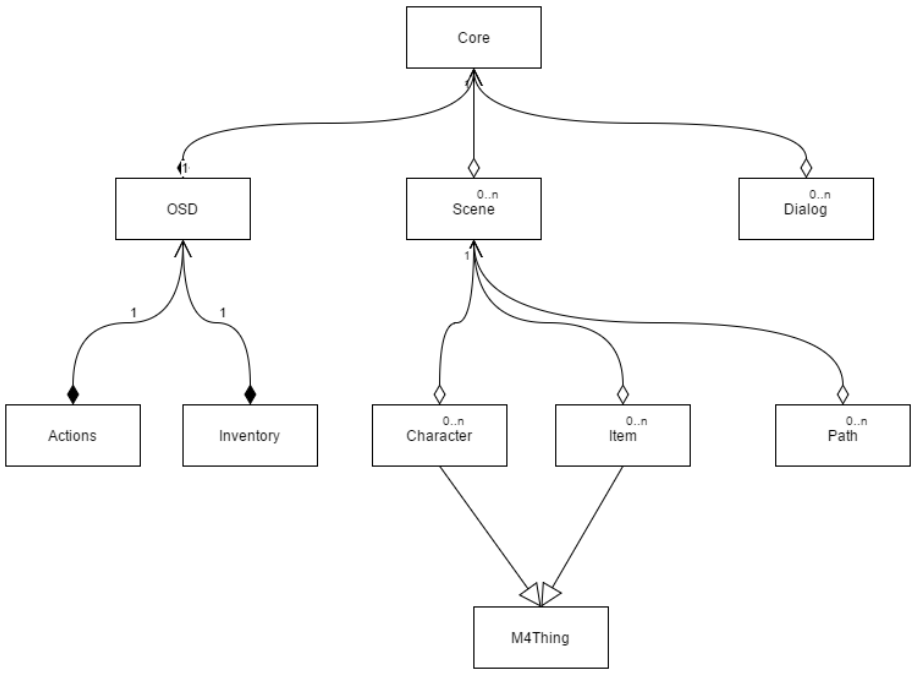

Figura 4.4: Diagrama de clases de la aplicación

4.2.1. Core.js

El núcleo del motor lo compone una clase llamada *Core* que se encarga de inicializar y gestionar al resto de componentes. Desde esta clase se crea y configura el objeto Pixi, el cual se encargará de gestionar la salida gráfica. También aquí se lee el archivo config.js, el cual contiene la definición completa del juego. Por último se crean los objetos necesarios para hacer funcionar el juego según lo especificado en config.js y se pone en marcha todo el sistema.

4.2.2. Scene.js

Los objetos de tipo *Scene* componen cada una de las pantallas del juego. Pueden crearse tantos como sean necesarios y cada uno de ellos es responsable de gestionar todos los objetos de esa pantalla, así como de los eventos de entrada que

sucedan en ese escenario. Una escena puede contener fondos, personajes y objetos, además de otros elementos de control. Los objetos *Scene* también se encargan de de gestionar las pulsaciones de ratón para determinar donde debe moverse el personaje principal.

- Fondos: Los fondos son los elementos que actúan de lienzo para cada escena. Normalmente representarán escenarios tales como habitaciones, paisajes o cualquier lugar en que trascurra la acción. Los fondos pueden estar compuestos por una o varias imágenes (capas). En caso de estar compuestos por más de una imagen, se genera automáticamente un efecto *Parallax*, que consiste en hacer que las imágenes superpuestas se muevan a diferentes velocidades, dando una sensación de profundidad.

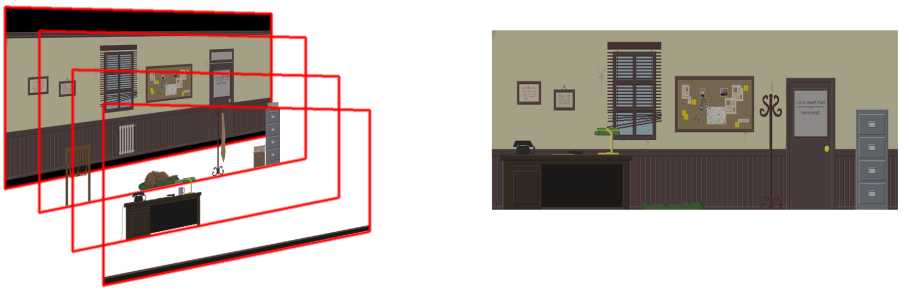

Figura 4.5: Escenario compuesto por varias capas para crear la sensación de profundidad con efecto *Parallax*

- Personajes: Cada personaje será una instancia del objeto *Character*. Cada escena puede contener uno o varios personajes. Uno de ellos debe ser siempre el personaje principal, controlado por el jugador. El personaje principal será el encargado de llevar a cabo las acciones dictadas por el jugador, que tomará este rol durante el juego. Este personaje puede recoger objetos, usarlos, o interactuar con otros personajes manteniendo conversaciones.
- Cámara: El objeto cámara, dentro del programa, es un objeto que se encarga de mantener las coordenadas X e Y así como el alto y el ancho de la ventana de visión del usuario dentro del escenario. Con ello, las rutinas de dibujado pueden saber que parte del escenario deben dibujar cuando el escenario es más grande que la pantalla física. La cámara sigue automáticamente el movimiento del personaje principal para mantenerlo siempre a la vista, aunque esto puede modificarse por el usuario, si lo desea.
- Areas: Los fondos de los escenarios son imágenes planas, pero suelen representar espacios con altura, anchura y profundidad. Así pues, aunque para el sistema todo el fondo sea una imagen rectangular, los personajes no tienen por que moverse por toda ella de igual manera. Es aquí donde los objetos *Area* limitan los movimientos de los personajes a ciertas zonas de la pantalla.

  Estos objetos almacenan formas geométricas complejas en forma de lista de coordenadas y se encargan de controlar que ningún personaje se mueva fuera de estas áreas. Además, cuando el jugador indica que quiere desplazarse

a un punto concreto del escenario, los objetos *Area* se encarga de procesar el camino óptimo, si existe, o de encontrar el punto válido más cercano en caso de no encontrar un camino.

Figura 4.6: En verde, objeto *Area* visible calcula y limita los movimientos de los personajes por el escenario.

- Objeto: Los objetos, dentro del juego (no confundir con los objetos del lenguaje de programación), son instancias del objeto *Item* y representan elementos del escenarios con los cuales el jugador, a través del personaje, puede interactuar. Por ejemplo, un reloj en la pared, una silla, o una puerta son objetos dentro de la escena.

  El jugador puede interactuar con ellos a través de las acciones definidas durante el diseño del juego. Los tipos de interacción clásica suelen ser acciones tales como: coger, mirar, empujar, abrir o usar.

4.2.3. Character.js

Los objetos de tipo *Character* representan a los personajes dentro del juego. Uno de ellos será el personaje principal, controlado por el usuario. El resto serán personajes secundarios con los cuales se puede interactuar. No hay diferencia estructural entre los personajes principal y secundarios más que el nombre de instancia cuando son creados. Ambos tipos de personaje pueden andar, hablar y cualquier otra animación que diseñe el usuario.

Figura 4.7: Personaje en distintos fotogramas de su animación.

Para gestionar las animaciones, tanto los objetos *Character* como los objetos *Item*, heredan de una clase común *M4Thing* encargada de gestionar las imágenes que componen cada objeto, los fotogramas asignados a cada animación y la velocidad de la misma.

4.2.4. Item.js

Los objetos *Item* son los encargados de gestionar los elementos manipulables del escenario. Un elemento manipulable es todo aquel con el que el personaje principal puede interactuar de cualquier forma, excluyendo a los demás personajes, que podriamos considerar como un tipo especial de *Items*.

Un *Item* está compuesto por una o varias imágenes y puede tener asociados diferentes comportamientos para responder a las diferentes acciones del jugador. También existe la posibilidad de crear *Items* sin imágenes. En este caso, se debe especificar un alto y un ancho y el objeto será transparente. Esto puede ser útil para definir objetos con cierta interactividad, pero cuya representación gráfica está dibujada en el fondo de la escena.

4.2.5. Osd.js

Llamaremos *Osd*, siglas de *On Screen Display*, al objeto encargado de gestionar todos los elementos gráficos que no forman parte del escenario, pero son necesarios para el desarrollo del juego. Como son: Las acciones, el acceso al inventario, y la interfaz para los diálogos entre personajes. Estos tres elementos del motor se detallan a continuación.

Figura 4.8: En verde, resaltados los elementos que componen el sistema OSD.

4.2.6. Action.js

La clase *Action*, de la cual solo se crea un objeto, es la encargada de mostrar el menú de acciones disponibles cuando el jugador selecciona un objeto o un perso-

naje del escenario. Estas acciones vendrán definidas por el usuario al diseñar el juego y provocarán diferentes reacciones.

Una vez el jugador ha seleccionado un objeto o un personaje, se mostrará el listado de acciones disponibles y se permitirá elegir una de ellas. Como consecuencia, se ejecutará la programación que el objeto o el personaje tengan asociada a esa acción.

Por ejemplo, si el jugador selecciona un papel sobre una mesa, el listado de acciones disponibles podria ser: coger, mirar, hablar. De entre estas tres opciones el jugador selecciona coger, por lo que el papel pasa a formar parte de su inventario.

4.2.7. Inventory.js

El objeto *Inventory* representa el inventario del personaje, es decir, un almacén en el que puede transportar objetos de una pantalla a otra, para usarlos en el momento adecuado. El funcionamiento del inventario esta totalmente automatizado por el motor y, siempre que exista una acción coger, el objeto será eliminado del escenario y añadido al inventario.

Figura 4.9: Inventario abierto, mostrando objetos recolectados.

4.2.8. Dialog.js

Los objetos de tipo *Dialog* almacenan y ejecutan conversaciones de texto entre dos personajes. Para ello, disponen de una estructura de árbol en la que el jugador puede escoger de entre varias opciones/frases que decir, a lo cual el otro personaje responderá con una frase o acción. Por ejemplo, ante una pregunta, un personaje puede responder con una frase o entregando un objeto.

Estos diálogos pueden utilizarse como puzles adicionales, obligando al jugador a llevar la conversación por el camino correcto para conseguir un objetivo. O simplemente pueden aportar un hilo conductor a la historia.

Las animaciones de los personajes están programas con una característica especifica para la sincronización de la boca del personaje con el texto que esta diciendo en ese momento. Esta propiedad se ha programado expresamente en los

objetos de tipo *Charactger* y permite asignar diferentes fotogramas a fonemas concretos como se muestra en la tabla 4.1.

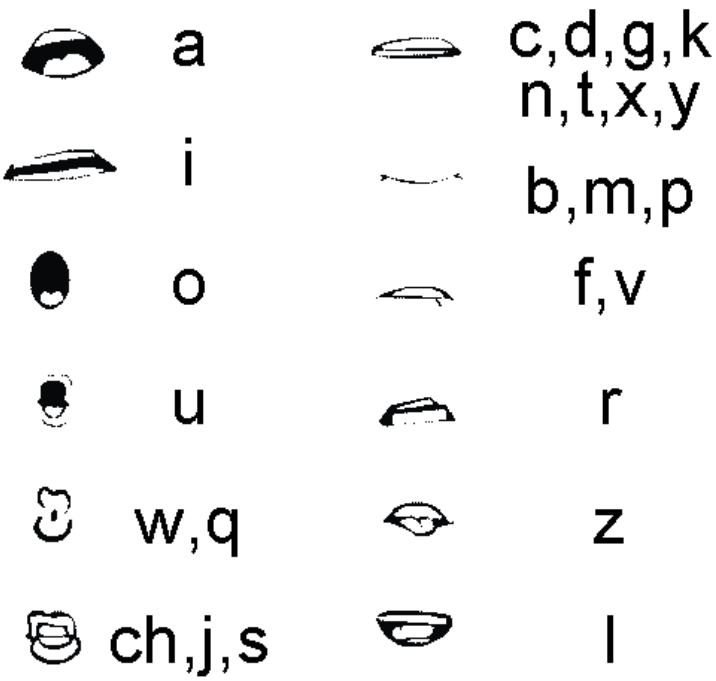

Tabla 4.1: Relación entre caracteres y expresiones faciales para sincronización de los diálogos.

CAPÍTULO 5
Manual de uso del motor M4

Aunque el objetivo es que, para crear un juego de tipo aventura gráfica utilizando este motor, no sean necesarios conocimientos en programación, harán falta ciertas directrices para configurar correctamente los archivos necesarios en cada proyecto. En este capítulo se detallan cada uno de los pasos a seguir en este proceso, analizando el archivo principal de configuración y definiendo el formato de los archivos multimedia necesarios.

5.1 Preparación de los recursos multimedia

Al margen de la definición del archivo *config.js*, de la que hablaremos con mas detalle en la sección 5.3, para crear un videojuego de tipo aventura gráfica con el motor M4 es necesario disponer de los archivos gráficos que compongan el diseño. El sistema acepta archivos de imágen con formato JPEG, PNG y GIF, cada uno de ellos con sus peculiaridades:

- JPEG: El estandar JPEG (ISO/IEC 10918)[1] fue creado en 1992. Se trata de un sistema de compresión de imagen con perdida. El espacio de color es de 24 bits, por lo que es adecuado para el almacenamiento de imágenes con calidad fotográfica. No admite transparencia.
- PNG: El formato PNG es la primera recomendación de la W3C[2], la organización que regula los estándares en Internet. Es un formato de imagen con color indexado, es decir, la paleta de colores está limitada por una tabla interna que, por contra, ayuda a reducir el tamaño del archivo sin producir pérdidas de calidad. PNG soporta píxeles con diferentes grados de transparencia.
- GIF: Creado por Compuserve en junio de 1987, es hoy en dia de libre uso y distribución[3]. Al igual que PNG, almacena la paleta de color en una tabla, lo que le permite ocupar menos espacio que JPEG a cambio de tener una limitación en el número de colores. Tambien soporta pixeles transparentes, con la diferencia de que no admite niveles de transparencia sino únicamente

[1] https://jpeg.org/jpeg/
[2] http://www.libpng.org/pub/png/spec/
[3] https://www.w3.org/Graphics/GIF/spec-gif87.txt

pixeles transparentes u opacos. Es el único de los tres formatos que soporta internamente animaciones.

Los fondos y fotogramas que componen las animaciones pueden almacenarse como archivos independientes o como hojas de *sprites*. Una hoja de *sprites* es un solo archivo de imagen compuesto por varias imágenes dispuestas en cuadricula. En este caso, el motor acepta tamaños de celda variables y no hay limite en el tamaño del archivo resultante.

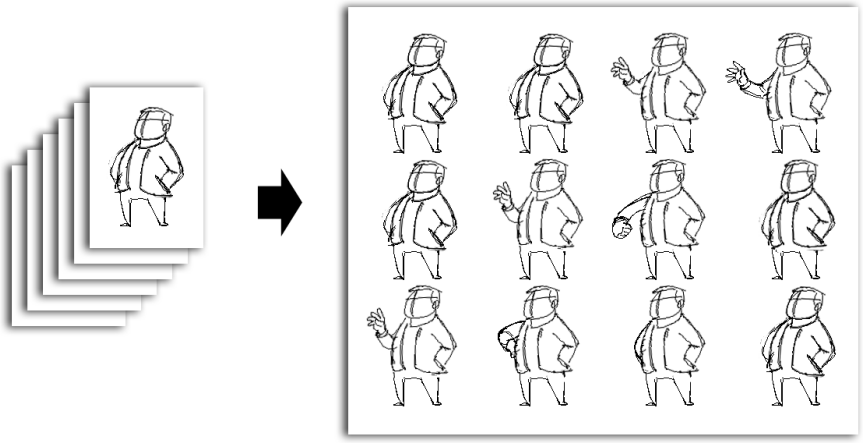

Figura 5.1: Hoja de *sprites*. Un único archivo de imágen compuesto por un mosaico de múltiples archivos.

5.2 El arbol de archivos del motor M4

Un juego completo, creado utilizando el sistema descrito en este proyecto, constará de múltiples archivos y directorios. A continuación se describe la estructura en disco de un proyecto, describiendo cada uno de sus bloques.

```
├── index.html
├── config.js
├── m4engine
│   ├── fonts
│   ├── m4engine.js
│   └── pixi.js
└── juego
    └── sprites.png
```

- index.html: Es el archivo base para arrancar el proyecto en un navegador web. Conforma un archivo siguiendo el estándar HTML5 y se encarga de cargar los archivos JavaScript necesarios para el funcionamiento del juego. El nombre index.hmtl es un convencionalismo que hace que, en caso de no especificar ningún archivo, un servidor web sirva este archivo por defecto, pero no tiene mayor importancia.

```
<!DOCTYPE HTML>
<html>
    <head>
      <title>Motor</title>
        <meta charset="utf-8">
      <style>
        body {
          margin: 0;
          padding: 0;
          background-color: #000000;
                /*cursor: none;*/
        }
            canvas{
                display: block;
                margin: 0px auto;
            }
      </style>

      <script src="m4engine/pixi.js"></script>
        <script src="config.js"></script>
        <script src="m4engine/m4engine.js"></script>

    </head>
    <body>

    </body>
</html>
```

Listing 5.1: Archivo HTML basico para la cargar la aplicación bajo un navegador web

- config.js: Contiene la definición completa del juego en formato JSON. El formato json es un formato de almacenamiento de datos estructurados compatible con JavaScript. Tiene la peculiaridad de que puede ser modificado y administrado con herramientas visuales existentes, pero también es editable como texto plano sin dificultad. La estructura interna de este archivo se detalla en la sección 5.3.

- m4engine: Directorio que contiene los archivos que componen el motor de juego. El nombre de este directorio no debe modificarse. En la versión de producción contendrá un archivo m4engine.js y pixi.js que no deben modificarse. En la versión de desarrollo, el archivo m4engine.js está dividido en varios archivos (uno por clase) que pueden modificarse si se desea, modificando el comportamiento del motor.

  El directorio fonts almacena las tipografías utilizadas por el motor para representar textos en pantalla. Por el momento no es posible cambiarlas sin cambiar también el código fuente en m4engine.js, aunque seria una característica deseable en futuras versiones.

- juego: El directorio juego contendrá todos los archivos multimedia necesarios para el funcionamiento del juego diseñado. En el ejemplo se ha incluido un archivo llamado sprites.png, pero el contenido de este directorio será el

que el usuario necesite, sin ningún tipo de restricción. El nombre del directorio también puede modificarse, siempre que corresponda con la configuración establecida en el archivo config.js.

Por tanto, para el uso previsto del motor, necesitaríamos el archivo index.html y el directorio m4engine, ambos sin modificar. Y el archivo config.js y el directorio juego, en los que el usuario modificará todo lo necesario para crear su aventura gráfica.

5.3 Estructura del archivo *config.js*

El archivo config.js esta compuesto internamente por una sola estructura de datos en formato JSON. Para el correcto funcionamiento del motor, esta estructura no debe contener errores y debe cumplir unos requisitos asi como contener unos datos mínimos concretos. La idea es que, modificando estos datos, el usuario pueda crear un juego totalmente nuevo sin necesidad de modificar el código fuente JavaScript.

5.3.1. El formato estándar JSON

JSON se basa en un subconjunto de definiciones de datos de JavaScript, en su versión ECMA-262 de Diciembre de 1999. JSON se almacena en formato de texto plano, por lo que es completamente independiente del lenguaje de programación bajo el que se use, aunque su nomenclatura será muy familiar para programadores que hayan trabajado con CC, C++, Java, JavaScript, Perl, Python y otros muchos [8].

Su característica mas importante para el desarrollo del presente proyecto es que puede almacenar objetos complejos y vectores de datos de forma que sean fácilmente interpretables por un algoritmo de programación, pero también legibles por un humano.

Tipos básicos: Los tipos de datos básicos que soporta JSON son: cadenas de texto y datos numéricos. En el caso de las cadenas de texto, deben ir siempre encerradas entre comillas . En cuanto a los datos numéricos, no hace distinción entre enteros y decimales ni entre la precisión de los decimales. En este caso, los digitos no enteros se separan por un punto.

Ejemplos:

```
1 "verde";
2 "azul";
3 12;
4 213.44;
5 13.4;
```

Listing 5.2: Tipos básicos en JSON

Vectores: Los vectores en JSON se definen como una serie de valores entre corchetes [] y separados por comas. Un vector puede almacenar indistintamente cadenas, datos numéricos u objetos.

Ejemplos:

```
["verde", "azul", "amarillo"];
["rojo", 14, "negro", 22.11];
```

Listing 5.3: Vectores en JSON

Objetos: Los objetos JSON no pueden contener funciones, ya que se trata de un formato de intercambio de datos, pero pueden almacenar y representar objetos mediante pares nombre/valor. El objeto debe estar contenido entre llaves y los pares nombre/valor separados por dos puntos :. Los valores almacenados pueden ser de cualquiera de los tipos expuestos en este apartado, incluyendo vectores u otros objetos. Los pares se separan entre si mediante comas.

Ejemplos:

```
{
  color: "azul",
  velocidad: 120,
  ancho: 4.5,
  alto: 2,
  caracteristicas: ["AC","EE","CC"]
}
```

Listing 5.4: Objetos compuestos en JSON

5.3.2. Formato del archivo config.js

El archivo config.js, dentro del motor, es un caso concreto de archivo JSON, que debe cumplir con una estructura preestablecida para ser leído por el núcleo del programa y generar correctamente los elementos del juego. Este archivo puede (y debe) ser editado por el usuario para crear el juego según el diseño deseado, pero cumpliendo siempre las especificaciones de JSON y la estructura establecida por propio motor, detallada a continuación:

```
{
  "name": "Mi aventura personal",
  "folder": "juego",
  "assets": [
    "sprites.png",
    "inicio.jpg"
  ],
  "width": 640,
  "height": 360,
  "textHeight": 20,
  "textPadding": 10,
  "textSize": 16,
  "colorOsdLo": "#f6a800",
  "colorOsdHi": "#F5CE7A"
  "verbs": [
    {
      "name": "coger",
      "defaultText": "No puedo coger eso."
    },
    {
```

```
      "name": "mirar",
      "defaultText": "No le veo nada raro."
    },
    {
      "name": "hablar",
      "defaultText": "No se que decir."
    }
  ],
  "characters": [
    {
      "name": "Simba",
      "color": "#c03e57",
      "images": [
        "quieto.png",
        "matt_anda_1.png",
        "matt_anda_2.png",
        "matt_anda_3.png"
      ],
      "animations": {
        "stop": [0],
        "walk": [1,2,3],
        "talk": [0]
      }
    }
  ],
  "scenes": [
    {
      "name": "Calle",
      "img": "capa_exterior_despacho_3.png",
      "areas": [
        [30,347,1820,34]
      ],
      "items": [
        {
          "name": "farola",
          "x": 612,
          "y": 36,
          "width": 37,
          "height": 327,
          "actions": []
        },
        {
          "name": "chicle",
          "x": 623,
          "y": 200,
          "actions":[]
        }
      ],
    }
  ],
  "dialogs": []
}
```

Listing 5.5: Estructura completa de un archivo config.js

El objeto principal definido en el listado JSON cuenta con las siguientes propiedades:

- name: Nombre del juego. Es título que se desea dar al juego completo y que aparecerá como título en la ventana del navegador si se ejecuta en un navegador convencional.

- folder: Directorio en el que se buscaran los recursos multimedia especificados por el vector *assets*. El directorio debe estar al mismo nivel que el archivo config.js y su nombre por defecto es juego.

- assets: Listado de recursos multimedia necesarios para generar los elementos del juego. Es decir: fondos, fotogramas de las animaciones de los personajes, imágenes de los objetos, interfaz gráfica, etc.

- width: Ancho en píxeles de la pantalla de juego.

- height: Alto en píxeles de la pantalla de juego.

- textHeight: Alto en píxeles de una línea de texto.

- textPadding: Distancia mínima entre el borde de la pantalla de juego y cualquier texto que aparezca.

- textSize: Tamaño de la tipografía.

- colorOsdLo: Color en formato hexadecimal para los textos de la interfaz de usuario.

- colorOsdHi: Color secundario en formato hexadecimal para los textos de la interfaz de usuario.

- verbs: Listado de verbos admitidos por el juego. Estos verbos serán las acciones disponibles sobre los objetos cuando el jugador haga click sobre ellos. Además se especifica una frase textual por defecto que el personaje pronunciará en caso de que no se haya definido ninguna acción para el objeto seleccionado.

- characters: Listado de personajes, cada uno de ellos definido mediante un objeto JSON.

- scenes: Listado de escenas que componen el juego completo, cada una de ellas definida mediante un objeto JSON. Las escenas, ademas de sus propiedades internas, contienen objetos y personajes, todo ello definido en vectores dentro de la estructura de la propia escena.

- dialogs: Listado de dialogos implicados en el juego, cada uno de ellos definido como un objeto JSON. Los dialogos deben tener nombres únicos para diferenciarlos y el sistema los pondrá en marcha cuando un personaje se disponga a hablar con otro.

5.3.3. Como crear un personaje

Para definir personajes utilizaremos, por un lado, las imágenes necesarias para animarlo, y por otro la estructura detallada en el listado de ejemplo del apartado anterior, de la que se concretan las funciones de cada una de las propiedades a continuación.

```
{
  "name": "Simba",
  "color": "#c03e57",
  "images": [
    "quieto.png",
    "matt_anda_1.png",
    "matt_anda_2.png",
    "matt_anda_3.png"
  ],
  "animations": {
    "stop": [0],
    "walk": [1,2,3],
    "talk": [0]
  }
}
```

Listing 5.6: Ejemplo de personaje en config.js

- name: Nombre del personaje. Se mostrara al pasar el ratón sobre el, a menos que se trate del personaje principal, que no puede ser seleccionado. El nombre también se utiliza para referirse a este personaje en otros puntos del archivo de configuración. Por ejemplo, para agregarlo a un escenario.
- color: Color del texto en pantalla cuando habla este personaje. Útil para facilitar al jugador el diferenciar unos personajes de otros durante los diálogos.
- images: Listado de archivos que forman todos los fotogramas de este peronajes. No es necesario que esten en ningun orden especial, ya que el orden de los fotogramas por cada animación se especifica en la propiedad *animations*. Es necesario que las imagenes aqui descritas hayan sido cargadas previamente en la pripiedad *assets* del objeto principal.
- animations: Listado de animaciones, cada una de ellas compuesta por un nombre y un vector indicando el número de fotograma de vector *animations*. Por ejemplo: "walk": [1,2,3] indica al sistema que, cuando el personaje realiza la funcion de andar, deben alternarse los fotogramas 1, 2 y 3 del listado *animations*.

5.3.4. Como definir una escena

Al igual que con los personajes, para definir una escena utilizaremos, por un lado, las imágenes necesarias para componer el fondo y los objetos, y por otro lado, la estructura detallada en el listado de ejemplo del apartado anterior, de la que se concretan las funciones de cada una de las propiedades a continuación.

```
{
  "name": "Calle",
  "img": "capa_exterior_despacho_3.png",
  "areas": [
    [30,347,1820,34]
  ],
  "items": [
    {
      "name": "farola",
      "x": 612,
      "y": 36,
      "width": 37,
      "height": 327,
      "actions": []
    },
    {
      "name": "chicle",
      "x": 623,
      "y": 200,
      "actions":[]
    }
  ],
}
```

Listing 5.7: Ejemplo de escena en config.js

- name: Nombre de la escena. Debe ser un nombre único y se utilizara dentro del motor para referirse a esta escena en concreto a efectos de cargarla, descargarla, añadirle objetos, personajes o cualquier otra acción que se desee realizar sobre ella.

- img: Recurso gráfico que hará la función de fondo de la escena. Debe ser un archivo existente en el directorio del juego y definido previamente en pa propiedad *assets* del objeto principal. Si en vez de una imagen se especifica un vector de imágenes, se generará automáticamente un escenario por capas, con efecto de profundidad.

- areas: Listado de áreas geométricas por las que puede moverse el personaje principal. Éstas áreas estaran definidas por pares de coordenadas x e y, especificadas en píxeles.

- items: Listado de objetos interactivos en este escenario. Cada uno de ellos debe definirse con un nombre, unas coordenadas x e y, dadas en píxeles y una imagen que previamente habrá sido cargada desde la propiedad *assets* del objeto JSON. Es posible no especificar ninguna imagen si el objeto esta, por ejemplo, dibujado directamente sobre el fondo pero, en ese caso, se debe especificar su alto y ancho mediante las propiedades *width* y *height*. Además, cada uno de los objetos cuenta con una propiedad *actions* que permite añadir programación a medida al motor. Esto se analiza con más detalle en la sección 5.3.6.

5.3.5. Como definir un diálogo

Los diálogos son modos especiales dentro del juego en los que la acción normal se interrumpe y se inicia un modo exclusivo de conversación entre dos personajes. El jugador puede seleccionar una de entre una serie de opciones o frases que decir, a lo que el personaje controlado por el sistema contestará en consecuencia. Estos diálogos pueden utilizarse para dar mayor dimensión a la narraración de la historia que hace de hilo conductor del juego, o pueden formar parte de los puzles, teniendo que elegir las frases adecuadas para llegar a un buen término. Esto es decisión del diseñador del juego final. A continuación de detalla la estructura con la que se define un diálogo en el archivo de configuración, pudiendo definir tantos como sean necesarios.

```
{
      "name": "hablar con hombre enfadado",
      "lines": [
        {
          "id": 0,
          "options": [
            {
              "text": "Dejeme pasar.",
              "goto": 1
            },{
              "text": "Parece usted enfadado.",
              "goto": 2
            }
          ]
        },{
          "id": 1,
          "say": [
            {
              "character": "Pepe",
              "text": "Dejeme pasar"
            },{
              "character": "hombre enfadado",
              "text": "Nadie va a entrar ni salir de este edificio."
            }
          ],
          "goto": 0
        },{
          "id": 2,
          "say": [
            {
              "character": "Pepe",
              "text": "Parece enfadado."
            },{
              "character": "hombre enfadado",
              "text": "Y usted huele muy mal."
            }
          ],
          "goto": -1
        }
    ]
}
```

Listing 5.8: Ejemplo de diálogo definido en config.js

Cada objeto *dialog* se compone de tan solo dos propiedades: *name* y *lines*. La propiedad *name* hace referencia al nombre único que debe tener cada objeto de diálogo dentro del juego para el correcto funcionamiento interno del motor. La propiedad *lines* es un vector de objetos mas complejos que definen un arbol de decisión a lo largo de un diálogo.

Cada uno de los elementos en el vector *lines* es, a su vez, un objeto JSON con las siguientes propiedades:

- id: Identificador único, dentro de este diálogo, del nodo del árbol. Se trata de un identificador numérico que no debe repetirse a lo largo de este mismo objeto diálogo. No es necesario que sigan ningún orden, pero esto ayuda a su lectura por un lector humano.

- say: Objeto que especifica que frase del diálogo se dirá al llegar a este nodo del arbol y que personaje la dirá (ejecutará su animación de hablar, si la tiene). Esta propiedad es incompatible en un mismo nodo con la propiedad *options*.

- options: Listado de opciones, cada una de ellas compuesta por un texto literal y una propiedad *goto*. Cuando el diálogo se encuentra con este nodo, muestra el listado de todas las opciones en pantalla y espera a que el usuario seleccione una. Hecho esto, el sistema pasa a ejecutar el nodo con el *id* correspondiente a la propiedad *goto* de la opción seleccionada. Esta propiedad es incompatible en un mismo nodo con las propiedades *say* y *goto*.

- goto: Una vez terminada la ejecución del nodo actual, el sistema pasa a ejecutar el nodo con el *id* correspondiente a esta propiedad. Esta propiedad no es compatible en un mismo nodo con la propiedad *options*. Si el valor de esta propiedad es -1, el diálogo termina y el juego vuelve al modo de ejecución normal.

5.3.6. Ampliar funcionalidad mediante JS

El objetivo principal y diferenciador de otros proyectos similares en el que se centra del presente desarrollo es la capacidad de definir un juego completo y funcional a través de un archivo de configuración, sin necesidad de programar código en sentido clásico. Este archivo de configuración podría, en un futuro, ser generado con un software visual, haciendo todo el proceso más asequible para cualquier usuario.

Aún así, se ha incluido la capacidad de ampliar la funcionalidad de todo el sistema, dividiendo este proceso en dos fases de diferente alcance. Una de ellas sería la completa modificación del código fuente, como se expresa en la seccion 4.1.2. En este apartado se describe un método intermedio: Añadir código JavaScript directamente sobre el archivo de configuración config.js.

Con este fin, se pueden crear propiedades dentro de la estructura JSON que compone el archivo de configuración en las que puede incluirse código JavaScript completo.

- scene: Dentro del objeto *scene*, pueden añadirse tres propiedades que admiten código JavaScript:
  - before: El código JavaScript introducido como valor de esta propiedad se ejecutará antes de que la escena sea mostrada en pantalla al usuario.
  - after: El código JavaScript introducido como valor de esta propiedad se ejecutará en el momento en que la escena sea eliminada, normalmente para cargar otra escena.
  - update: El código JavaScript introducido como valor de esta propiedad se ejecutará continuamente, mientras la escena actual este activa.
- item: Dentro del objeto *item*, puede añadirse una propiedad JavaScript por cada acción asociada al objeto (mirar, hablar, coger, etc).
  - actions: El código Javascript introducido como valor de esta propiedad se ejecutará en el momento en que el usuario seleccione esta acción sobre el objeto actual.
- dialog: Dentro del objeto *dialog*, puede añadirse una propiedad JavaScript por cada linea de guión que deba ejecutar un personaje.
  - actions: El código Javascript introducido como valor de esta propiedad se ejecutará en el momento en que el personaje ejecute esta linea del diálogo.

Ejemplo:

```
"items": [
        {
          "name": "farola",
          "x": 612,
          "y": 36,
          "width": 37,
          "height": 327,
          "actions": [
            "mirar": "
                farola_vista++;
                if(farola_vista>3)core.character.say('Estoy harto de
                    mirar esta farola');
            "
          ]
        }
]
```

Listing 5.9: Inserción de código personalizado en el archivo config.js

Como puede apreciarse en el ejemplo, a un *item* se le ha añadido la propiedad *actions*, que contiene líneas de código JavaScript por cada tipo de acción existente en el sistema. Este código se ejecutará en el momento en que el jugador interactúe con este objeto usando esa acción.

En el ejemplo, cada vez que el jugador mira la farola, se incrementa una variable *farola_vista*. Si esta variable ha llegado a 3, el personaje dice que esta harto de mirar la farola. Este es un comportamiento no soportado a priori por el motor, pero conseguido a través de estas inserciones de código.

Mediante estos bloques de código integrados en el archivo de configuración, el comportamiento del sistema puede personalizarse mas allá de lo previsto por el diseño inicial del motor, sin llegar a tener que manipular directamente el código del núcleo.

5.3.7. Ejecución en dispositivos móviles

Cualquier juego creado con el motor objeto de este trabajo puede ejecutarse en un dispositivo, siempre que este disponga de un navegador web compatible. Esto incluye todos los navegadores web modernos tanto de escritorio como móviles. Al estar construido íntegramente con tecnologías web, puede distribuirse como un paquete de archivos o alojarse en un servidor, desde el cual el jugador podrá acceder sin necesidad de tener previamente el juego descargado.

Además de la forma de ejecución anterior, el juego generado puede compilarse como aplicación nativa de diferentes sistemas operativos, incluyendo dispositivos móviles. Esto permite alojar cualquier producto generado con este motor en las tiendas oficiales de aplicaciones, pudiendo distribuirlo e incluso venderlo.

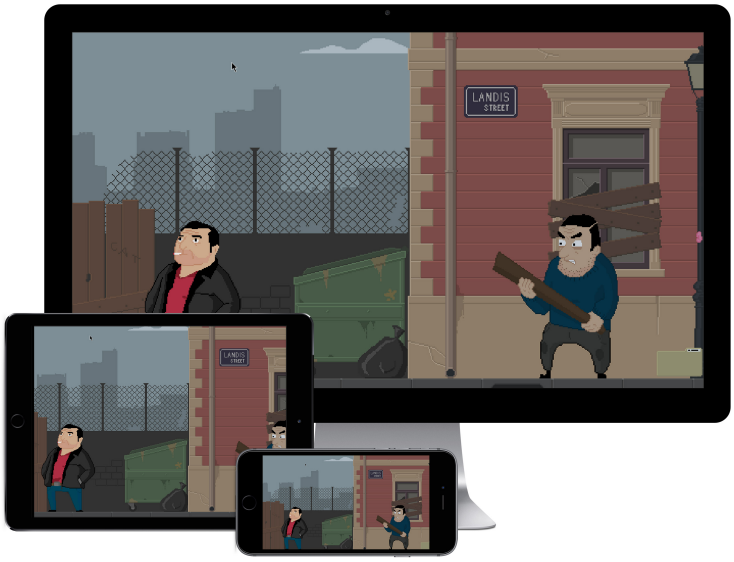

Figura 5.2: Motor y juego adaptado a diferentes pantallas y dispositivos.

Para generar una aplicación móvil nativa, tanto para sistemas operativos Android, iOS o Windows Mobile, son necesarios dos requisitos:

1. El *software* de desarrollo oficial de cada sistema operativo.
2. Un *software* especifico para compilar aplicaciones web para sistemas operativos móviles.

En cuanto a *software* que permita esta característica, cabe destacar Apache Cordova[4]. Cordova es un proyecto desarrollado por el equipo Apache Software

[4] https://cordova.apache.org/

Foundation[5], responsable del servidor web Apache, actualmente el más utilizado en servidores web Unix en todo el mundo.

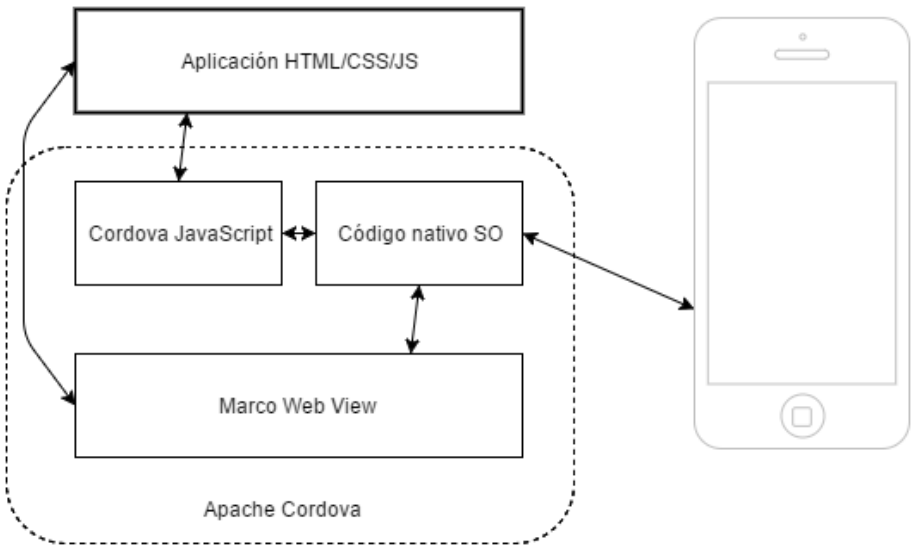

Figura 5.3: Diagrama de funcionamiento de Apache Cordova.

Cordova permite convertir una aplicación creada con tecnologías web en una aplicación nativa para diferentes sistemas operativos móviles. Para ello se sirve de el motor de representación web de Firefox, integrándolo en un proyecto nativo para sistemas Android, iOS o Windows Mobile. Además incorpora funciones para facilitar la comunicación de estas aplicaciones web con el *hardware* del dispositivo, como el acelerómetro, GPS, cámara, etc.

```
$ cordova create MiAplicacion
$ cd MiAplicacion
$ cordova platform add Android
$ cordova run Android
```

Listing 5.10: Comandos de compilación de una aplicación con Apache Cordova

Basados en Apache Cordova han surgido otros proyectos, algunos mejorando a éste en algunos aspectos, aunque no siempre han mantenido el mismo modelo de licencia gratuita y código abierto del proyecto original. Los más destacados son Cocoon.js y PhoneGap.

[5] https://projects.apache.org

CAPÍTULO 6
Conclusiones

En este último capítulo analizaremos los problemas y posibles soluciones encontradas durante la implementación del proyecto así como un análisis de los objetivos propuestos y cumplidos. Para finalizar, se establece una lista de posibles ampliaciones y mejoras que podrían aplicarse al sistema en un futuro.

6.1 Consideraciones finales

A lo largo del proceso de implementación nos hemos encontrado con varios problemas, algunos de tipo técnico y otros mas conceptuales o de diseño. A continuación describiremos los mas relevantes y, si procede, la solución que se ha aplicado.

- Uno de los retos más importantes durante el diseño, ha sido el decidir el nivel de automatización en el desarrollo que debía ofrecer el sistema. Al tratarse de un motor que automatiza la tarea de creación de videojuegos, nos encontramos ante la posibilidad de aumentar al máximo la generación automática de elementos del juego, limitando así la libertad del usuario, o preservar la libertad del usuario para incluir sus propias variaciones, creando un sistema más flexible, pero también mas complejo en su uso.

  Analizando en perspectiva los objetivos del proyecto, ha primado la opción de mantenerlo asequible para usuarios sin conocimientos en programación. Por tanto, un juego completo puede ser desarrollado sin necesidad de programar, aunque se han incluido las propiedades *action* en el archivo de configuración para permitir la inserción de código a los usuarios más avanzados.

- Desde un primer momento, el desarrollo de las funciones que componen el núcleo del motor se ha abordado de forma que puedan ejecutarse de forma asíncrona. Esto permite que en el juego puedan suceder varios eventos al mismo tiempo, sin interferir ente ellos. Por ejemplo: Un personaje puede estar hablando mientras otro personaje esta andando.

  Esto ha supuesto un problema a la hora de querer realizar acciones de forma síncrona. Por ejemplo: Un personaje anda hasta un punto del escenario y, una vez ha llegado, coge un objeto.

La solución ha pasado por la creación de una clase especifica para gestionar las listas de acciones, que decide cuales deben ejecutarse en paralelo y cuales deben esperar a que otras terminen para continuar. Esto ha añadido algo de complejidad a la implementación del resto de funciones, que deben adaptarse a este modelo, pero ha solucionado el problema permitiendo utilizar las dos opciones según sea necesario.

Según los objetivos que se plantearon al inicio del proyecto, y que se describen con detalle en la sección 1.2, se ha conseguido desarrollar un sistema que cumple con todos los requisitos deseados.

El sistema es capaz de crear un juego funcional a partir de los archivos multimedia y de un único archivo de configuración config.js, siempre este sea correcto.

El motor creará los escenarios y personajes descritos por el archivo de configuración, animándolos automáticamente para andar, hablar y cualquier otra animación adicional que desee el usuario. Es posible crear objetos en el escenario con los que interactuar, tanto estáticos como animados. Estos objetos pueden ser, además, recolectados y almacenados en el inventario por el personaje principal. Entre dos personajes puede establecerse un dialogo, definido por un árbol de decisión en el archivo de configuración. Este dialogo es conducido por el jugador seleccionando opciones en pantalla.

El otro objetivo importante del proyecto era mantener el código libre y gratuito. Habiendo utilizado todas las librerías de terceros con este tipo de licencias, se ha conseguido cumplir también con este punto.

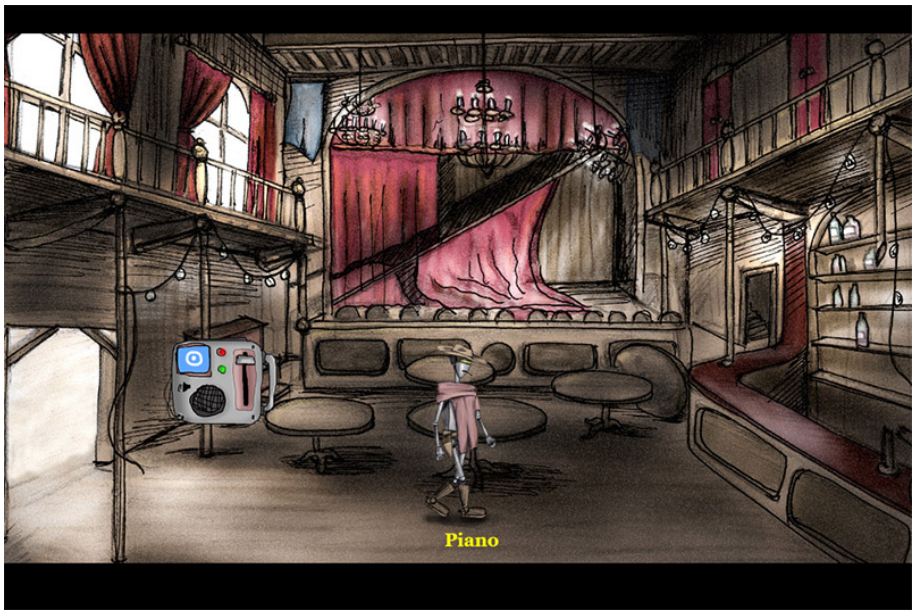

Figura 6.1: Juego creado usando el motor M4.

Como ejemplo, una demostración completa totalmente jugable puede encontrarse en `http://screen13.net/beta/m4`, y el código completo y su documentación de uso en el repositorio de *software* `https://github.com/Screen13/mmmm`.

6.2 Trabajo futuro

Como motor para videojuegos de tipo aventura gráfica, este proyecto puede ampliarse y mejorarse en muchas de sus facetas, según los requisitos concretos que un usuario pueda desear para un desarrollo en particular. Al margen de características particulares, para que los resultados obtenidos se aproximen un poco más a los videojuegos comerciales actuales hay una serie de requisitos deseables:

- Traducciones: Aunque todas las cadenas de texto están definidas en el archivo config.js, sería una mejora considerable poder tener los textos destinados a aparecer en pantalla en archivos separados. De este modo se facilitaría la traducción a varios idiomas sin interferir con la lógica del juego.
- Iluminación: Pese a ser un juego en dos dimensiones, existen técnicas de iluminación que permiten mejorar la apariencia gráfica final. Desde sombreados simples hasta uso de luces dinámicas.
- Editor: Una mejora importante en cuanto a usabilidad sería la implementación de un editor visual para modificar el archivo de configuración o manipular los archivos multimedia. Al tratarse de una estructura estándar JSON, es relativamente sencillo implementar un editor que cumpla con este requisito.
- Sonido: En la actual versión del motor, es posible añadir música o efectos sonoros a través de la inserción de código JavaScript en el archivo de configuración. Esto podría simplificarse y automatizarse para que el usuario no tuviese que utilizar programación para añadir el sonido.
- Guardar/cargar: Una característica interesante para el motor sería la capacidad de guardar el estado en un momento concreto del juego para poder recuperarlo más adelante. De esta forma podrían crearse historias más largas sin la necesidad de que el jugador las termine de una sola vez.

Aunque estos requisitos no formaban parte de los objetivos iniciales, esperamos poder seguir trabajando y mejorando el proyecto mas allá de la versión expuesta en esta memoria e implementar estas y mas mejoras en un futuro próximo. El código actualizado se mantendrá en https://github.com/Screen13/mmmm.

Bibliografía

[1] Arora, Sumeet *WebGL Game Development* PACKT Publishing, 2014.

[2] Documentación API de ECMAScript (JavaSript), versón 6. Consultado en https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia.

[3] Documentación API de la librería Pixi.JS, versón 3. Consultado en http://pixijs.github.io/docs.

[4] Fulton, Steve; Fulton Jeff *HTML5 Canvas* O'Reilly Media, 2013.

[5] Jerz, Dennis G. *Colossal Cave Adventure (c. 1975)* Seton Hill University, 2004.

[6] Kent, Allen; Williams, James G *Encyclopedia of Microcomputers 3* CRC Press, Enero, 1989.

[7] Moss, Richard *A truly graphic adventure: the 25-year rise and fall of a beloved genre*. Ars Technica, Enero, 2011.

[8] Ray Rischpater *JavaScript JSON Cookbook* Packt Publishing, 2015.

[9] Rodríguez Echeverrría, Roberto; Sosa Sánchez, Encarna; Prieto Ramos, Álvaro *Programación Orientada a Objetos* Librería Papelería Álvaro, 2004.

[10] Rollings, Andrew; Ernest Adams *Fundamentals of Game Design* Prentice Hall, 2006.

[11] The Circle of Life: An Analysis of the Game Product Lifecycle. Consultado en http://www.gamasutra.com/view/feature/1453/the_circle_of_life_an_analysis_of_.php.

APÉNDICE A
Configuración de *software*

Listado y descripción del *software* utilizado durante el desarrollo del proyecto y sugerencias de aplicaciones a utilizar como complemento al motor descrito. Se ha buscado que, al igual que el objeto de esta memoria, todas las aplicaciones utilizadas sean también gratuitas y de código abierto.

- Navegador: El presente proyecto esta construido como una aplicación web, con el objetivo de poder portarse de forma inmediata a cualquier dispositivo actual. Por este motivo, un requisito para lanzar un producto creado a partir del motor es disponer de un navegador web.

  Durante el desarrollo se ha utilizado principalmente Firefox, por dos motivos: Ser *software* libre y disponer de una gran cantidad de complementos avanzados destinados al desarrollo. Para comprobar el correcto funcionamiento en ortos navegadores también se han utilizado: Google Chrome, Microsoft Internet Explorer, Microsoft Edge, Safari, Safari Mobile y Chrome Mobile.

- Depuración: JavaScript es un lenguaje de programación interpretado que no cuenta con compilador o intérprete oficiales, por lo que puede ser complicado depurar el código sin las herramientas adecuadas. Por suerte, la mayoria de navegadores incorporan ya estas herramientas de serie, aunque en este caso se ha utilizado un complemento externo para Firefox llamado Firebug, algo mas completo y potente que la herramienta por defecto.

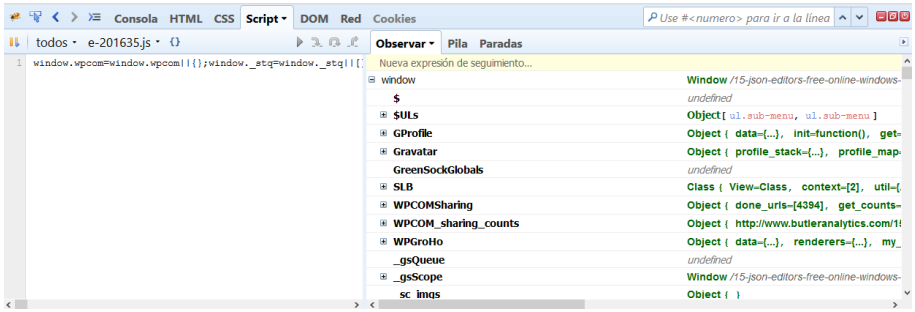

Figura A.1: Captura de pantalla del complemento Firebug.

- Editor: Todos los lenguajes implicados en el desarrollo, tanto de programación como estructurales, se codifican como texto plano y pueden ser creados

y editados con cualquier editor de texto que admita este formato. En este caso se ha utilizado Brackets, un editor sencillo pero potente orientado a la edición de código, desarrollado por Adobe, pero distribuido bajo licencia de código abierto.

Otras aplicaciones complementarias recomendadas, pero no obligatorias, a la hora de crear un videojuego utilizando este motor son:

- Edición del archivo config.js: El archivo de configuración de cada proyecto, config.js, esta pensado para poder ser editado con un editor de texto plano aunque, al estar construido como un objeto JSON, puede utilizarse cualquiera de los múltiples programas que existen para editar este tipo de documentos. Destacamos el editor en línea `http://www.jsoneditoronline.org/` y la aplicación de escritorio Visual JSON Editor.

- Archivos de imagen: Para la edición de imagen también existe una amplia gama de aplicaciones de todas clases y para todos los sistemas operativos. Como herramienta gratuita podemos destacar el editor GIMP, disponible para Windows, MacOS y Linux.

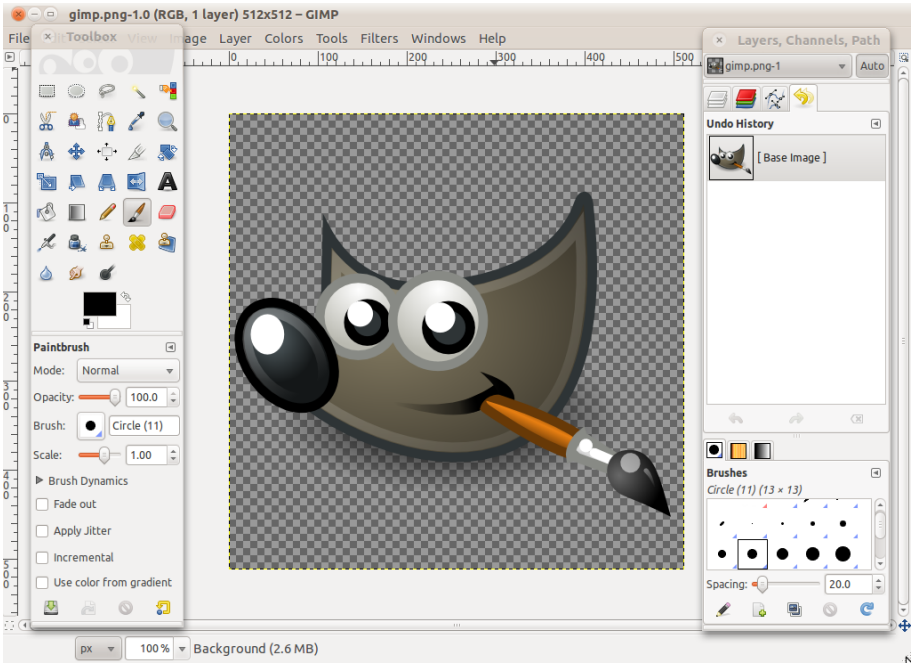

Figura A.2: Interfaz del editor de imágenes GIMP.

- Hojas de *sprites*: El motor admite archivos de imagen por separado, aunque puede mejorarse el rendimiento y velocidad de carga, sobre todo en aplicaciones en línea, si se agrupan varias imágenes en una sola. Este proceso puede hacerse con cualquier editor de archivos de imagen o utilizando software especifico. Entre estos últimos, podemos destacar TexturePacker, aunque sólo está disponible para sistemas Windows.

APÉNDICE B
Ejemplo config.js completo

```
config={
  "name": "Matt Murphy",
  "folder": "juego",
  "assets": [
    "matt.png",
    "exterior_despacho.png"
  ],
  "width": 640,
  "height": 360,
  "textHeight": 20,
  "textPadding": 10,
  "textSize": 16,
  "colorOsdLo": "#f6a800",
  "colorOsdHi": "#F5CE7A",
  "verbs": [
    {
      "name": "coger",
      "defaultText": "No puedo coger eso."
    },
    {
      "name": "mirar",
      "defaultText": "No le veo nada raro."
    },
    {
      "name": "hablar",
      "defaultText": "No sabria que decir."
    }
  ],
  "characters": [
    {
      "name": "Matt Murphy",
      "color": "#c03e57",
      "images": [
        "matt_quieto.png",
        "matt_ojos_1.png",
        "matt_ojos_2.png",
        "matt_anda_1.png",
        "matt_anda_2.png",
        "matt_anda_3.png",
        "matt_anda_4.png",
        "matt_anda_5.png",
        "matt_anda_6.png",
        "matt_coge_000.png",
```

```
        "matt_coge_001.png",
        "matt_coge_002.png",
        "matt_coge_003.png",
        "matt_coge_004.png",
        "matt_coge_005.png",
        "matt_coge_006.png",
        "matt_coge_007.png",
        "matt_coge_008.png",
        "matt_coge_009.png",
        "matt_coge_010.png",
        "matt_coge_011.png",
        "matt_boca_cerrada.png",
        "matt_boca_a.png",
        "matt_boca_o.png",
        "matt_boca_e.png",
        "matt_boca_ts.png",
        "matt_boca_ln.png",
        "matt_boca_mbp.png",
        "matt_boca_fv.png"
      ],
      "animations": {
        "stop": [1,1,1,1,1,1,2,1],
        "walk": [3,4,5,6,7,8],
        "talk": [21,22,23,24,25,26,27,28],
        "pick": [9,10,11,12,13,14,15,16,17,18,19,20]
      }
    }
  ],
  "inventory": [
    {
          "name": "chicle",
          "images": [
            "inventario_chicle.png"
          ],
          "actions": {
            "coger": []
            "mirar": []
          }
        },
        {
          "name": "botella",
          "images": [
            "inventario_botella.png"
          ],
          "actions": {
            "coger": [],
            "mirar": []
          }
        }
  ],
  "scenes": [
    {
      "name": "Exterior despacho",
      "img": "capa_exterior_despacho_3.png",
      "areas": [
        [30,347,1820,347,1820,347,30,347]
      ],
      "items": [
        {
```

```
          "name": "farola",
          "x": 612,
          "y": 36,
          "width": 37,
          "height": 327,
          "actions": []
        },
        {
          "name": "chicle",
          "images": [
            "obj_chicle.png"
          ],
          "x": 623,
          "y": 200,
          "actions": []
        }
      ],
      "characters": [
        {
          "name": "hombre enfadado",
          "color": "#416cc2",
          "images": [
            "enfadado_quieto.png",
            "enfadado_anda_1.png",
            "enfadado_anda_2.png",
            "enfadado_anda_3.png",
            "enfadado_anda_4.png",
            "enfadado_anda_5.png",
            "enfadado_anda_6.png",
            "enfadado_e.png",
            "enfadado_fv.png",
            "enfadado_ln.png",
            "enfadado_mbp.png",
            "enfadado_ojos2.png",
            "enfadado_ojos1.png",
            "enfadado_mano1.png"
          ],
          "animations": {
            "stop": [0,0,0,13,0,13,0,0,0,0,0,0,0,0,0,12,0,0,13,
            0,0,11,11,11,11,12,11,0],
            "walk": [1,2,3,4,5,6],
            "talk": [7,8,9,10,7,8,9,10,7]
          },
          "x": 500,
          "y": 345
        }
      ]
    }
  ],
  "dialogs": [
    {
      "name": "hablar con hombre enfadado",
      "lines": [
        {
          "id": 0,
          "say": [
            {
              "character": "Matt Murphy",
              "text": "Disculpe."
```

```
            },
            {
              "character": "hombre enfadado",
              "text": "Que que quiere usted"
            }
          ],
          "goto": 1
        },
        {
          "id": 1,
          "options": [
            {
              "text": "Tengo que solucionar unos asuntos en esas
                 oficinas.",
              "goto": 2
            },
            {
              "text": "Parece enfadado.",
              "goto": 3
            },
            {
              "text": "Digame que no esta buscando a Matt Murphy.",
              "goto": 4
            }
          ]
        },
        {
          "id": 2,
          "say": [
            {
              "character": "Matt Murphy",
              "text": "Tengo que solucionar unos asuntos en esas
                 oficinas."
            },
            {
              "character": "hombre enfadado",
              "text": "Nadie va a entrar ni salir de este edificio
                 hasta que no acabe lo que tengo que hacer."
            }
          ],
          "goto": 1
        },
        {
          "id": 3,
          "say": [
            {
              "character": "Matt Murphy",
              "text": "Parece enfadado."
            },
            {
              "character": "hombre enfadado",
              "text": "Y usted huele muy mal."
            }
          ],
          "goto": 1
        },
        {
          "id": 4,
          "say": [
```

```
            {
              "character": "Matt Murphy",
              "text": "Digame que no esta buscando a Matt Murphy."
            },
            {
              "character": "hombre enfadado",
              "text": "Es exactamente lo que estoy haciendo."
            }
          ],
          "goto": 5
        },
        {
          "id": 5,
          "options": [
            {
              "text": "Vayase a casa, yo le hago el relevo.",
              "goto": 5
            },
            {
              "text": "Hasta luego.",
              "goto": 6
            }
          ]
        },
        {
          "id": 6,
          "say": [
            {
              "character": "Matt Murphy",
              "text": "Hasta luego."
            },
            {
              "character": "hombre enfadado",
              "text": "Adios."
            }
          ],
          "goto": -1
        }
      ]
    }
  ]
}
```