Universidad Zaragoza

Trabajo Fin de Grado

JAGE (JustAnotherGameEngine): Creación de un motor de videojuegos 2D multiplataforma de código abierto

Autor/es

Rubén Tomás Gracia

Director/es

Eduardo Mena Nieto

Grado en Ingeniería Informática
Escuela de Ingeniería y Arquitectura
2016

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Rubén Tomás Gracia,

con nº de DNI 18174166A en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) JAGE(JustAnotherGameEngine): Creación de un motor de videojuegos 2D multiplataforma de código abierto,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 20 de Junio del 2016

Fdo: Rubén Tomás Gracia

Para mi abuelo Juan y mi tío abuelo Mateo,
por ser quienes más influyeron en mi interés por los videojuegos

Agradecimientos

A mi familia, por nunca rechazar mi afición por los videojuegos.
A mi novia, por apoyarme siempre y ayudarme a seguir trabajando.
A mis amigos, por ayudarme en las pruebas del proyecto.

JAGE (JustAnotherGameEngine): Creación de un motor de videojuegos 2D multiplataforma de código abierto

Resumen

Hoy en día los videojuegos están más presentes en la vida diaria de lo que nunca han estado. Son mucho más accesibles para la gente con pocos medios, Asimismo, la extensión de la Informática ha propiciado el hecho de que mucha gente haya podido adoptar la programación como hobby. Esto ha desencadenado un estallido en el mundo de los videojuegos, de gente que programa sus propios videojuegos con bajo (o ningún) presupuesto para luego publicarlos en Internet: conocidos como *juegos indie*, están ahora en su mejor momento. Todo esto ha llevado a cabo una retroalimentación por la que cada vez los videojuegos están más aceptados, y más gente quiere hacer su propio juego. Muchas personas no quiere un videojuego comercial, solo explotar su lado artístico en un mundo que disfrutan, y por ello cada vez han surgido más herramientas que permiten hacer juegos con un esfuerzo reducido: los motores.

En este proyecto se implementa, usando Java, desde cero y en código abierto, un motor de videojuegos 2D con la flexibilidad como máxima prioridad. Para demostrar dicha flexibilidad, se han creado tres juegos completamente distintos con él, que han permitido explorar todas las posibilidades que da el motor, así como servir de retroalimentación para el motor y saber qué era necesario cambiar en éste para cumplir los objetivos propuestos.

El proyecto pretende ser una ayuda a todas las personas que quieren probar el desarrollo de videojuegos, tanto para aquellas que no saben ni quieren saber nada de programación, como para las que quieren desarrollar un motor propio, puesto que el proyecto se ha desarrollado bajo una licencia de código abierto.

Índice de contenido

1. INTRODUCCIÓN

Un motor de videojuegos es un programa que se encarga de las tareas más complicadas y repetitivas, como por ejemplo el renderizado de gráficos, y los cálculos físicos. A partir de ello, el usuario puede simplemente dedicarse a definir las escenas que luego el motor se encargará de hacer funcionar y de mostrar por pantalla.

Sin embargo, la mayoría de motores que existen actualmente, o son demasiado complejos para un juego 2D, demasiado sencillos, o demasiado caros. Motores como Unreal Engine[1] y Cryengine[2] son motores completamente enfocados a juegos 3D y tienen un reducido soporte 2D. Unity Engine[3] cada vez es mejor para los juegos 2D, sin embargo, sigue siendo más enfocado al 3D y es demasiado general para juegos en 2D. Game Maker[4] es un buen motor 2D, sin embargo su precio para poder lanzar un juego en Linux ya asciende a los $149.99, y su lenguaje de scripting es bastante limitado. Y RPG Maker[5], un motor para videojuegos 2D RPG con el que se pueden hacer juegos decentes sin saber programar nada, y que sabiendo programar puedes modificarlo como quieras, debido a que todo el motor está escrito en Ruby[12], tiene muchas versiones distintas, incompatibles entre sí, a un precio demasiado elevado para alguien que crea videojuegos simplemente como hobby.

Con el mercado de motores que hay actualmente, este proyecto quería proponer una alternativa totalmente gratuita para iniciar a aquellas personas que crean videojuegos como disfrute personal, pero a la vez permitiendo crear juegos perfectamente comerciales.

1.1. Objetivos

El objetivo es desarrollar un motor de videojuegos 2D completamente gratuito, que permita a los usuarios crear videojuegos sin tener por qué saber programar, que funcione tanto en Windows como en Linux, y que permita la fácil creación y distribución de extensiones del motor para permitir la creación de juegos completamente diferentes con el mismo motor.

1.2. Metodología y herramientas

Este proyecto se ha llevado a cabo en Java 1.7, usando la librería Slick2D[6], que facilita el renderizado por pantalla encapsulando las llamadas a OpenGL[9] en una API similar a la de Java2D[5].

Para eliminar la necesidad de que el usuario busque y enlace las librerias al proyecto, así como de crear los archivos ejecutables finales, se ha usado Gradle[7] como gestor de dependencias y de compilación.

1 Unreal Engine. [Citado el 19/06/2016] https://www.unrealengine.com
2 CryEngine. [Citado el 19/06/2016] https://www.cryengine.com
3 Unity 3D. [Citado el 19/06/2016] https://unity3d.com
4 Game Maker. [Citado el 19/06/2016] http://www.yoyogames.com/gamemaker
5 RPG Maker. [Citado el 19/06/2016] http://www.rpgmakerweb.com
6 Slick2D. [Citado el 19/06/2016] http://slick.ninjacave.com
7 Gradle. [Citado el 19/06/2016] http://gradle.org

Por último, se ha utilizado GitHub[8] como plataforma de gestión de configuraciones y de distribución, debido a su gran uso en el software de código abierto por sus posibilidades de participación pública en dichos proyectos, así como de su uso como copia de seguridad en caso de que haya algún problema con el computador usado.

1.3. Contenido de la memoria

Esta memoria se divide en tres partes claramente diferenciadas. Primero, se detallará el progreso de análisis del entorno de los motores y lenguajes de programación que conllevó a la elección de Java como lenguaje de desarrollo, y de las influencias que ha tenido el proyecto de otros motores y videojuegos. Segundo, se detallará cómo se ha implementado el motor y sus características más importantes que han permitido cumplir los objetivos propuestos. Tercero, se explicará cómo se ha extendido el motor para los distintos juegos que se han creado para demostrar todas sus capacidades.

1.4. Licencia usada

Para este proyecto, se ha optado por una licencia **MIT**[9], debido a que es una licencia permisiva, que impone pocas restricciones, y que permite una gran compatibilidad con el resto de licencias.

8 GitHub-JAGE(JustAnotherGameEngine). [Citado el 19/06/2016] https://github.com/valarion/JAGE
9 The MIT License (MIT).[Citado el 19/06/2016] https://opensource.org/licenses/MIT

2. TECNOLOGÍA UTILIZADA

Las tecnologías usadas en el proyecto no han sido elegidas por casualidad, sino que han pasado un proceso de selección para elegir las tecnologías más adecuadas para el tipo de programa a realizar.

2.1. Por qué Java

En el mercado actual de motores de videojuegos se usan muchos lenguajes de programación distintos para permitir la extensión del motor. Unity Engine usa C#[1] y Javascript[7], Unreal Engine usa UnrealScript[13] y C++[11], GameMaker usa GML (Game Maker Languaje)[10], RPG Maker usa Ruby y Javascript, y Cryengine usa Lua[8].

Para todo motor de videojuegos, es importante la implementación de un lenguaje de scripting que permita extender o modificar las funcionalidades básicas del mismo, para permitir la creación de elementos y efectos completamente nuevos y específicos para las necesidades de cada usuario. Para ello, hay que tener varios conceptos en cuenta: la facilidad para el usuario de aprender el lenguaje, la facilidad de uso del mismo, y la eficiencia en la ejecución de dicho lenguaje.

Para ello, se investigó los lenguajes de scripting más comunes actualmente de fácil implementación en un programa. Se busco un lenguaje de scripting que tuviera todas o la mayoría de las siguientes características: ha de ser multiplataforma, eficiente, orientado a objetos y compilable.

Los siguientes lenguajes contenían todas o la mayoría de las características requeridas: Lua, Ruby, Python[10] y Javascript. Se descartaron C# por ser propietario de Windows, y C++ por que el código compilado no es multiplataforma. De ellos, Ruby y Python prometían bastante, sin embargo, no son lenguajes fácilmente interoperables[11] [12], además, Ruby no es muy eficiente[14]. Lua y Javascript hubieran sido perfectos, si hubiera estado orientado a objetos enteramente, pero su orientación a objetos es compleja y nada intuitiva[13] [14]. Y Javascript, las librerías que existen no facilitan mucho la interoperabilidad entre los dos lenguajes.

Tras el primer análisis, se investigó qué más lenguajes de programación y scripting se podían usar para el motor de un videojuego. Entonces se penso en la ejecución de una máquina virtual de java desde C, y añadir interoperabilidad mediante JNI (Java Native Interface)[6]. Sin embargo, se llegó a la conclusión de que

10 GML Overview. [Citado el 19/06/2016] https://docs.yoyogames.com/source/dadiospice/002_reference/001_gml%20language%20overview/index.html

11 Ruby embedded into c++. [Citado el 19/06/2016] http://aeditor.rubyforge.org/ruby_cplusplus/index.html#id2866647

12 Embedding python in another application. [Citado el 19/06/2016] https://docs.python.org/2/extending/embedding.html#pure-embedding

13 OOP In JavaScript: What You NEED to Know. [Citado el 19/06/2016] http://javascriptissexy.com/oop-in-javascript-what-you-need-to-know

14 Programming in Lua. [Citado el 19/06/2016] https://www.lua.org/pil/16.html

no era necesario la ejecución de la máquina virtual de Java desde C, cuando se puede implementar todo en Java directamente.

Java es un lenguaje eficiente, orientado a objetos, compilado y multiplataforma, que ya está instalado en la mayoría de computadores del mundo. Tiene una fuerte comunidad detrás, por lo que cualquier problema encontrado se puede resolver fácilmente, existen muchas librerías para poder extender sus funcionalidades, y mucha gente ya ha aprendido a programar en Java.

Además, existe un motor conocido gratuito y de código abierto implementado en Java: jMonkeyEngine[15]; sin embargo no es un motor enfocado a usuarios con pocos conocimientos de programación, sino a desarrolladores experimentados (Ver Texto 1). Asimismo, es un motor de juegos 3D, por lo que su uso en juegos 2D es más complicado debido a la necesidad de cambiar la cámara de perspectiva a ortogonal, fijarla a un eje y dibujar sobre los otros dos.

> "It's not a visual RPG Maker or an FPS modder. You'll get the most out of the engine if you bring some programming aptitude to the table."
>
> Traducción:
> No es un RPG Maker visual in una herramienta para modificar juegos FPS.
> Conseguirás lo máximo del motor si tienes aptitudes de programación.

Texto 1: Cita de la página web del motor jMonkeyEngine y traducción.

Por todas estas características, Java era el lenguaje perfecto para el proyecto que se quería realizar.

2.2. Por qué Gradle

Uno de los grandes problemas de C y Java son las librerías. Se necesita la librería compilada correcta para el sistema operativo usado, o compilarla. Para ello, es necesario descargarla e incluirla en el proyecto. Esto conlleva dos problemas: si el proyecto en GitHub contiene las librerías quedando listo para descargar y compilar, se malgasta espacio en GitHub, costando, sobre todo, mayores tiempos de descarga por parte de los usuarios. Y si no se incluye en GitHub, se obliga al usuario a buscarla y enlazarla por su cuenta, lo que conlleva una molestia.

Este problema es difícil de solucionar en C. Recientemente, cerró Biicode[16], que pretendía resolver este problema en C y de momento no existe ninguna buena alternativa. En Java, sin embargo, este problema se resolvió hace mucho tiempo con Maven[17] y Ant[18], sin embargo, su sintaxis es algo más compleja si se quiere estructurar la organización de un proyecto. Por ello se optó por Gradle.

15 jMonkeyEngine. [Citado el 19/06/2016] http://jmonkeyengine.org
16 Biicode (just the company) post mortem. [Citado el 19/06/2016] http://blog.biicode.com/biicode-just-the-company-post-mortem
17 Maven. [Citado el 19/06/2016] https://maven.apache.org
18 Ant. [Citado el 19/06/2016] http://ant.apache.org

Gradle es una herramienta que facilita la construcción de código, no solo permitiendo indicar fácilmente cómo (mediante Groovy[4]), sino basándose en la estructura existente de Maven para la gestión de dependencias. De esta forma, se facilita la separación en distintos proyectos más pequeños, además de eliminar la necesidad de guardar las librerías en GitHub o de obligar al usuario a buscarlas y enlazarlas por su cuenta.

3. IMPLEMENTACIÓN DE LA BASE DEL MOTOR

La implementación básica del motor se basa en dos partes: primero, la extensibilidad del motor; segundo, la base del motor que buscará las extensiones , las añadirá, y comenzará la ejecución. Además, implementa la posibilidad de cargar fácilmente mapas creados con la herramienta gratuita y de código abierto Tiled[19].

3.1. Requisitos funcionales

La base del motor es el núcleo sobre el que se centra el proyecto y desde el que van a partir el resto de extensiones, por ello debe cumplir los siguientes requisitos funcionales:

- Debe ser capaz de cargar extensiones en tiempo de ejecución e interactuar con ellas.
- Debe mantener un tamaño mínimo indispensable, puesto que será una parte no modificable por las extensiones.

Sin embargo, el motor no se compone unicamente de la base. Puesto que para obtener la máxima flexibilidad es necesario mantener la base en el tamaño mínimo posible, la mayor parte de requisitos propios del motor se implementarán en la extensión principal que se explicará más adelante. Estos requisitos son:

Requisitos generales:

- Debe permitir el uso de mapas ortogonales creados con el programa editor de mapas Tiled.
- Debe permitir el uso de distintos mapas y la transición entre ellos de forma fluida.

Requisitos de modos de juego:

- Debe tener una pantalla de menu principal que permita al jugador elegir si jugar o salir del juego.
- Debe tener un modo de juego basado en los mapas anteriormente requeridos.
- Debe existir un sistema de combate que resulte en victoria o derrota.

Requisitos de movimiento:

- Debe ser capaz de ajustar el movimiento del jugador y computador a casillas, pudiendo moverse una cada vez, y sin poder moverse en diagonal.
- Debe calcular colisiones en el movimiento, si el jugador o computador desean moverse a una casilla ocupada.
- Debe ser capaz de mantener un sprite controlado por el jugador.
- Debe permitir la existencia de sprites controlados por el computador.
- Debe permitir al computador tomar el control del sprite del jugador, y devolverlo.
- Debe mantener al sprite del jugador centrado en pantalla en todo momento, salvo que se desee mostrar otra zona del mapa, o no haya sprite de jugador.

19 Tiled Map Editor. [Citado el 19/06/2016] http://www.mapeditor.org

Requisitos de menús:

- Debe permitir la interrupción del juego mediante menús de pausa.
- Debe permitir el guardado del juego en un momento deseado.
- Debe permitir el cargado de un juego anteriormente guardado y continuar la ejecución como si no hubiera habido interrupción.
- Debe permitir al jugador salir del juego de forma cómoda mediante menús dentro del propio juego.

Requisitos de interacción jugador/computador:

- Debe permitir mostrar y ejecutar diversos comportamientos según el movimiento e interacciones del sprite del jugador con el mapa y los sprites del computador.
- Debe permitir crear árboles de diálogo.

3.2. Extensibilidad del motor

Uno de los objetivos principales del proyecto era facilitar la extensibilidad del motor mediante el lenguaje de programación Java. Para ello, se disponía de distintas alternativas: Crear el motor como una librería que un usuario puede importar y usar con sus propias implementaciones, o crear el motor de forma que se puede añadir cualquier librería a él que extienda la funcionalidad.

La primera opción es mucho más sencilla de crear, pero más limitada. Es difícil permitir que haya distintas extensiones funcionando a la vez. Por ello, se optó por la segunda posibilidad. Para ello, Se decidió basar el sistema en el usado por Minecraft Forge[20] para permitir la extensión del juego Minecraft[21]. Este sistema consiste en poder crear un archivo jar autoencapsulado que simplemente copiándolo a una carpeta, sea cargado por el programa[22].

Sin embargo, Minecraft Forge es mucho más complejo que todo eso: permite la modificación del bytecode compilado del programa original[23]. Esto es demasiado complicado de realizar en un proyecto tan sencillo; sin embargo, existen alternativas, pues esta aproximación existe únicamente porque el juego original no permitía la existencia de extensiones. Sin embargo, el motor del proyecto está orientado desde el comienzo a la creación de extensiones. De esta forma, se ideó el sistema de extensiones usado.

El sistema de extensiones es un cargador de archivos jar en tiempo de ejecución, al que se le pasan las interfaces que se desean buscar, y devuelve todas las clases que implementan dichas interfaces. De esta forma, simplemente se desarrolla una extensión de una interfaz, se encapsula en un archivo jar, y se copia a la carpeta necesaria, y el motor lo encontrará y lo cargará al iniciarse. Para ello, se basa en el patrón de diseño *Bridge*[2] (ver Ilustración 1), declarando una abstracción para que los usuarios creen la implementación necesaria para cada ocasión. Posteriormente,

20 Minecraft Forge. [Citado el 19/06/2016] http://files.minecraftforge.net
21 Minecraft. [Citado el 19/06/2016] https://minecraft.net
22 How to install mods for Minecraft Forge. [Citado el 19/06/2016] http://www.minecraftmods.com/how-to-install-mods-for-minecraft-forge
23 Core Mod – Minecraft Forge. [Citado el 19/06/2016] http://www.minecraftforge.net/wiki/Core_Mod

se crearan instancias de las clases según su nombre (ver Texto 2), por lo que para sobrescribir la funcionalidad de una clase principal, simplemente se crea otra clase con el mismo nombre que remplace a la original. Además, como mecanismo de seguridad para evitar el remplazo de una clase inesperada, se puede implementar un método que decida si la clase debe reemplazar o no a la ya existente. Para ello, basta con crear un método con la anotación @ClassOverrider, y que reciba y devuelva un Objeto de tipo *Class<?>* (ver Texto 3). Recibirá la clase que existe actualmente, y devolverá la clase que debe remplazarla.

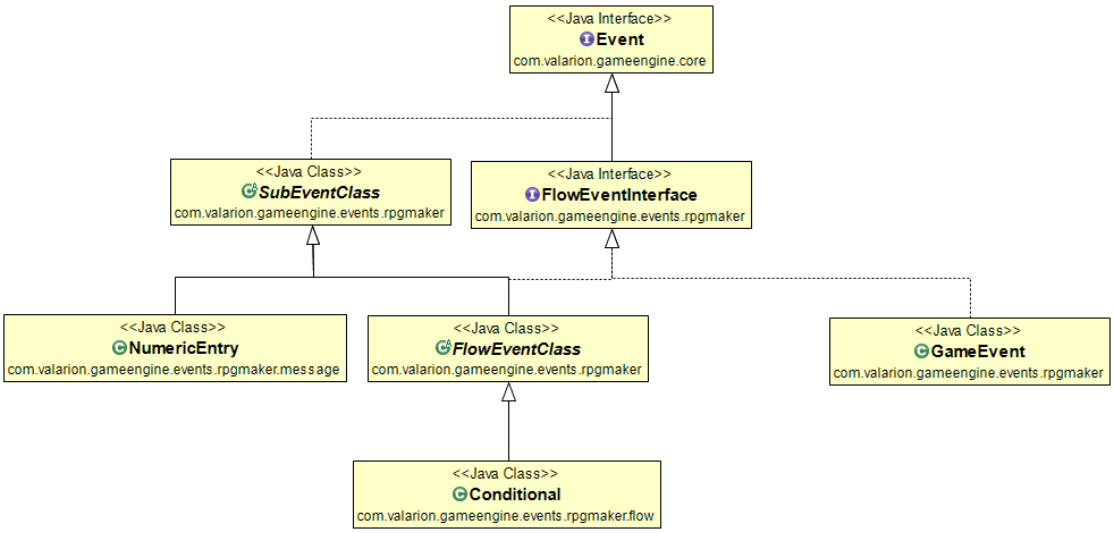


Ilustración 1: Ejemplo de uso del patrón Bridge

```
Map<String, Class<?>> map = game.getSets().get(Event.class);
Class<?> c = map.get(n.getNodeName());
Event e = (Event) c.newInstance();
```

Texto 2: Instanciación de una clase.

```
@ClassOverrider
public static Class<?> override(Class<?> c) {
	if(c.equals(com.valarion.gameengine.util.Camera.class)){
		return Camera.class;
	}
	else {
		return c;
	}
}
```

Texto 3: Método que decide qué clase debe reemplazar a la existente.

3.3. Base del motor

Tras desarrollar la extensibilidad, era necesario desarrollar la base del motor que cargaría las extensiones. Por ello, se optó por crear una base sencilla y crear una extensión enfocada hacia juegos RPG que serviría como base para el resto de extensiones. Sin embargo, esta extensión se verá en el siguiente capítulo, puesto que estrictamente hablando, es una extensión del motor.

La base del motor consiste en dos clases principales y diversas clases auxiliares. Una clase principal, es una ventana de Java Swing que permite seleccionar como se va a ejecutar el juego. Esta idea proviene de Unity Engine (Ver Ilustración 2). La ventana usada es una ventana mucho más simplificada que solo permite seleccionar la resolución y si se ejecuta a pantalla completa o no (Ver Ilustración 3).

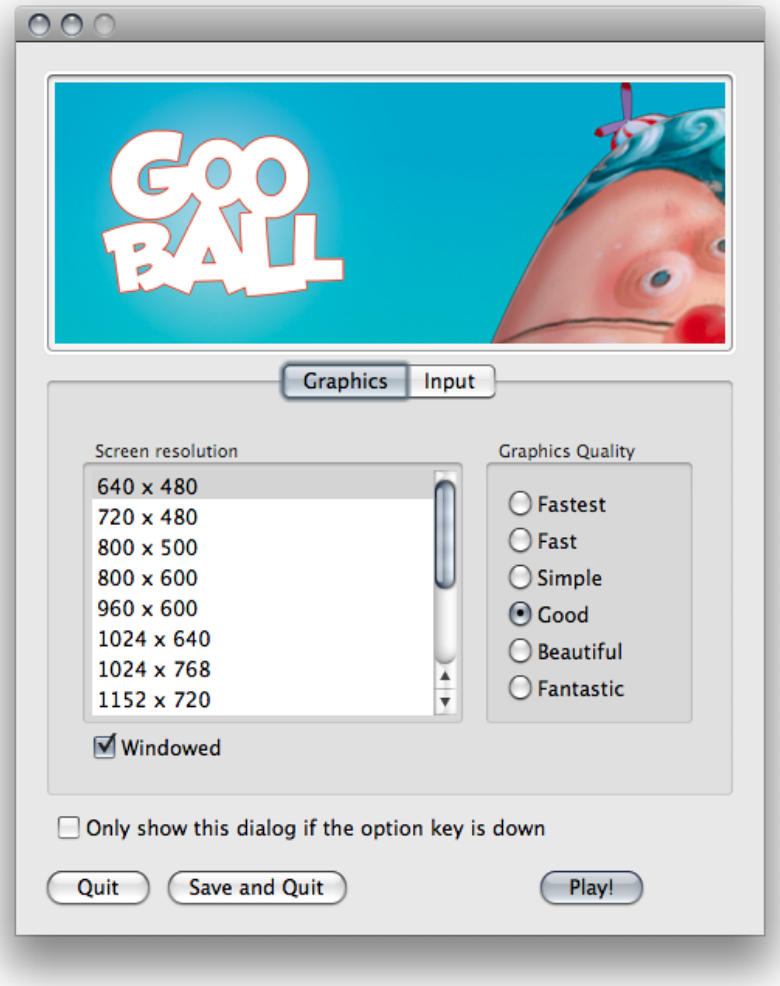


Ilustración 2: Ventana de configuración de Unity

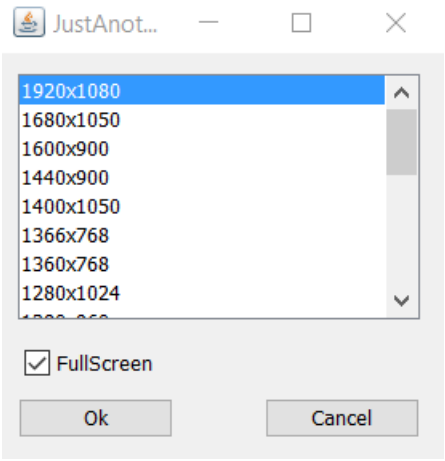


Ilustración 3: Ventana de configuración de JAGE

La segunda clase principal, es la clase que crea la ventana correspondiente dada la información de la clase anterior, carga las extensiones disponibles en la carpeta *modules* y crea una instancia de la clase llamada *StartState*, subclase de *GameState*, que se encargará de ser el motor en si mismo. Desde este momento en adelante, la clase principal guardará una instancia a un objeto del tipo *GameState* y se encargará de decirle cuándo debe actualizarse, y cuándo debe renderizarse.

Puesto que esta clase contiene la información de las extensiones cargadas, y para facilitar el acceso a las mismas, la clase implementa el patrón de diseño *Singleton*[3]. Para ello, el constructor de la clase es protegido, y solo se puede crear una instancia del mismo mediante su método *main*.

Las clases auxiliares son interfaces que definirán la parte abstracta del patrón de diseño *Bridge*, y que es son necesarias para que, al cargar las extensiones, se busquen las clases que implementan dicha abstracción (Ver Texto 4).

```
public interface Condition{
	/**
	 * Evaluate condition
	 * @param e Event in to which evaluate the condition.
	 * @return true if condition checks, false otherwise.
	 */
	public boolean eval(Event e, GameContainer container,
SubTiledMap map);

	/**
	 * Load the condition from an xml node.
	 * @param node Node to load.
	 * @param context Object of context. Usually the parent event.
	 * @throws SlickException
	 */
	public void load(Element node, Object context) throws
SlickException;
}
```

Texto 4: Ejemplo de interfaz como parte abstracta del patrón de diseño Bridge.

4. IMPLEMENTACIÓN DE LAS EXTENSIONES

Para mantener abiertas las posibilidades de extensión del motor, era necesario que la base del mismo fuera muy ligera, puesto que es inmutable. Por ello, es necesario crear extensiones para añadir la funcionalidad propia del motor. Por ello, se han creado tres extensiones: una extensión base, que consiste en crear un motor de juegos 2D RPG parecido a RPG Maker; y dos extensiones de la base que crean dos juegos diferentes usando los conceptos creados por ésta, modificándolos y ampliándolos dadas las necesidades de cada juego. Los juegos creados son: un juego de tipo puzle, clon del Tetris[24], y un juego de acción, clon del Skramble[25] (así mismo, un clon del Scramble[26] original de 1981).

4.1. Por qué un RPG, Puzle y Acción

Los tres tipos de juegos seleccionados tienen poco en común, y por eso han sido elegidos. El propósito es mostrar de qué es capaz la extensibilidad del motor. Sin embargo, si que tienen cosas en común que les permite reutilizar elementos, facilitando así la creación del juego (Ver Ilustración 4).

	RPG	PUZLE (Tetris)	ACCIÓN (Scramble)
Movimiento	Por casillas	Por casillas	Libre
Controles	Arriba, abajo, izquierda, derecha, aceptar, cancelar	Izquierda, derecha, girar, acelerar bajada, bajada instantánea	Arriba, abajo, acelerar, decelerar, disparar
Victoria	Resolver todos los puzles y ganar todas las batallas hasta el final	No hay victoria	Acabar todos los niveles y matar al jefe
Derrota	Perder una batalla	Amontonar piezas hasta que no puedan aparecer más	Chocar con un enemigo o terreno de frente
Personaje	Personaje con el que te puedes identificar	Pieza tetromino	Nave de ciencia ficción
Incentivo	Resolver puzles y ganar batallas	Conseguir puntos, eliminar lineas y subir niveles	Conseguir puntos y acabar niveles
Colisiones	Cuadrado ocupado / desocupado	Cuadrado ocupado / desocupado	Collisiones calculadas matematicamente
Movimiento automatico	Ninguno	Constante hacia abajo	Constante hacia la derecha
Rejugabilidad	Limitada: siempre mismos mapas y objetivos	Infinita: cada partida es distinta	Limitada: siempre mismos mapas y objetivos
Historia	Elaborada	Ninguna	Simple

Ilustración 4: Comparativa entre un juego RPG, Tetris y Scramble

Una de las principales cosas que tienen en común, es que los tres juegos tienen mapas estáticos sobre los que se mueven ciertos objetos, que en el motor son llamados eventos. Esto permite usar para los tres el mismo editor de mapas implementado en la base del motor.

4.2. Extensión RPG: JAGEmodules

La extensión JAGEmodules es la extensión base sobre la que se van a basar el resto de extensiones. Ésta implementa lo necesario para crear un juego RPG al más puro estilo RPG Maker. Para ello, define cómo es la pantalla de inicio del juego, la pantalla del juego de movimiento a través de los mapas, pantalla de batalla, cámara, menús, sistema de guardado de partidas, gestor de recursos usados en el juego y muchas implementaciones de las abstracciones definidas por la base del motor.

24 Tetris - Videogame by Atari Games [Citado el 19/06/2016] http://www.arcade-museum.com/game_detail.php?game_id=10081

25 Skramble. [Citado el 19/06/2016] https://www.c64-wiki.com/index.php/Skramble

26 Scramble - Videogame by Konami [Citado el 19/06/2016] http://www.arcade-museum.com/game_detail.php?game_id=9447

4.2.1. Patrón Composite

Pese a crear diversas implementaciones para cada abstracción definida en la base del motor, lo más importante es la implementación del patrón *Composite* en dos de ellas, permitiendo crear árboles. Éste se usa tanto en condicionales (para crear los operadores lógicos como AND, OR y NOT) (Ver Ilustración 5 más adelante) como en eventos.

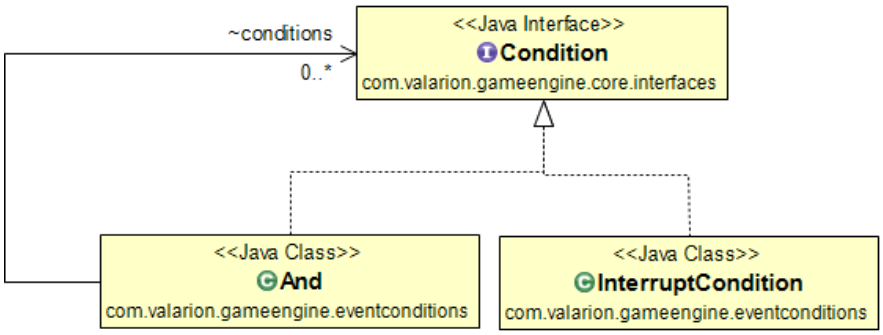


Ilustración 5: Ejemplo de patrón Composite aplicado a condicionales booleanos

Esto permite crear árboles de eventos y condiciones que se ejecutarán según sea necesario, creando un lenguaje de scripting mediante eventos que un usuario sin conocimientos de programación puede usar (Ver Texto 5). La idea era crear un sistema similar al de programación de eventos de RPG Maker (Ver Ilustración 6 más adelante)

```
<Conditional>
	<conditions>
		<SwitchCondition interrupt="81"/>
	</conditions>
	<true>
		<Dialog position="bot">
			<Text>You opened the jail with the jail keys.
			</Text>
		</Dialog>
		<Switch switch="82" action="true"/>
	</true>
	<false>
		<Dialog position="bot">
			<Text>The door is closed.</Text>
		</Dialog>
	</false>
</Conditional>
```

Texto 5: Ejemplo de estructura condicional en el patrón Composite creado

```
@>Conditional Branch: Switch [0081] == ON
   @>Text: -, -, Normal, Bottom
   :      : You opened the jail with the jail keys.
   @>Control Switches: [0081] = ON
   @>
: Else
   @>Text: -, -, Normal, Bottom
   :      : The door is closed
   @>
: Branch End
@>
```

Ilustración 6: Ejemplo de estructura condicional en RPG Maker.

De esta forma, se consigue replicar y ampliar las funcionalidades: Se pueden añadir más condiciones y se puede crear ramas para una de las dos condiciones (cierto o falso) o para ambas a la vez.

4.2.2. Gestor de recursos

El gestor de recursos es una clase que implementa el patrón *Singleton* y que mantiene en memoria todos los archivos de sonido, imágenes y animaciones necesarias para todo el juego. Lo único que no mantiene en memoria son los tilesets usados por cada mapa, pues éstos se cargan al cargar el mapa. La base de datos carga la información de un archivo XML con un formato concreto (Ver Texto 6).

```
<database>
	<spritesheet name="Emotes" image="emote.png"
		tilewidth="32" tileheight="32">
		<interrogation>
			<tile x="0" y="0"></tile>
			<tile x="1" y="0"></tile>
			<tile x="2" y="0"></tile>
		</interrogation>
	</spritesheet>
	<window name="defaultwindow" image="Window.png" border="2">
		<sub name="background" x="0" y="0" w="64" h="64"></sub>
		<sub name="decoration" x="0" y="64" w="64" h="64"></sub>
	<window>
	<image name="princessface" src="face/Actor4.png" x="288" y="96"

			w="96" h="96"></image>
	<sound name="door" src="025-Door02.ogg"></sound>
	<music name="title" src="Rain.ogg"></music>
</database>
```

Texto 6: Ejemplo de definición de recursos

4.2.3. Datos de la partida

El modulo principal implementa un sistema de guardado de partidas sencillo mediante la serialización propia de Java. Para ello, toda la información de la partida está contenida en una clase que implementa la interfaz *Serializable* y cuyos campos necesariamente implementan dicha interfaz. La clase guarda variables, interruptores y objetos globales, estadísticas (como el número de pasos), el nombre del mapa en el que se encuentra el usuario (el mapa se carga a partir del nombre, cualquier modificación en los eventos del mapa se pierde) y el evento que funciona como controlador del usuario. Sin embargo, este evento contiene campos que no se deberían guardar (las imágenes). Para lograrlo, es necesario modificar la forma en que los objetos de la clase se serializan y deserializan. Para ello, se ha de marcar los campos que no se deben guardar como *transient*, y crear dos métodos privados que modifiquen el comportamiento para que, en lugar de guardar las imágenes, se guarden los nombres de las mismas (Ver Texto 7).

```
private void writeObject(ObjectOutputStream stream) throws IOException
{
	stream.defaultWriteObject();
	if (sprite != null) {
		stream.writeBoolean(true);
		stream.writeObject(sprite.getName());
		stream.writeFloat(sprite.getSpritespeed());
		stream.writeFloat(sprite.getMovingspeed());
		stream.writeInt(sprite.getDirection());
	} else {
		stream.writeBoolean(false);
	}
}
private void readObject(ObjectInputStream stream) throws IOException,
ClassNotFoundException {
	stream.defaultReadObject();
	if (stream.readBoolean()) {
		sprite = Database.instance().getSprites().get((String)
		(stream.readObject())).createSprite(stream.readFloat(),
		stream.readFloat());
		sprite.setDirection(stream.readInt());
	}
}
```

Texto 7: Métodos modificados de serialización y deserialización.

4.2.4. Las implementaciones de GameState: modos de juego

La abstracción GameState está inspirada en Unreal Engine. En dicho motor, existen los modos de juego, que definen el numero de jugadores, cómo los jugadores entran en el juego, si se puede pausar el juego o no, y las transiciones entre niveles[27]. Ésto permite la creación de distintos modos de juego para distintas partes, como los menús y el propio juego en sí. En este aspecto, la extensión principal del motor implementa tres modos de juegos, llamados estados de juego en el motor.

El primero, define el funcionamiento del menú principal, y es el punto de entrada del motor (Ver Ilustración 7). El segundo estado de juego, define la funcionalidad propia de un RPG, es decir, el movimiento por casillas a través de los mapas (Ver Ilustración 9). Por último, el tercer estado compone un sistema de batalla por turnos estilo piedra-papel-tijera con ambientación de duelos de espada y escudo (Ver Ilustración 8).

Para transicionar entre los tres estados, cada uno tiene distintos métodos. El estado de menú principal cambia al estado de RPG cuando el usuario crea una nueva partida, o carga una existente. El estado RPG cambia a la pantalla principal o a la pantalla de combate mediante el uso de los eventos *QuitToMainMenu* y *Battle* respectivamente. Y el estado de combate cambia al RPG cuando se acaba la batalla, independientemente del resultado de la misma.

27 https://docs.unrealengine.com/latest/INT/Gameplay/Framework/GameMode

Ilustración 7: Pantalla principal

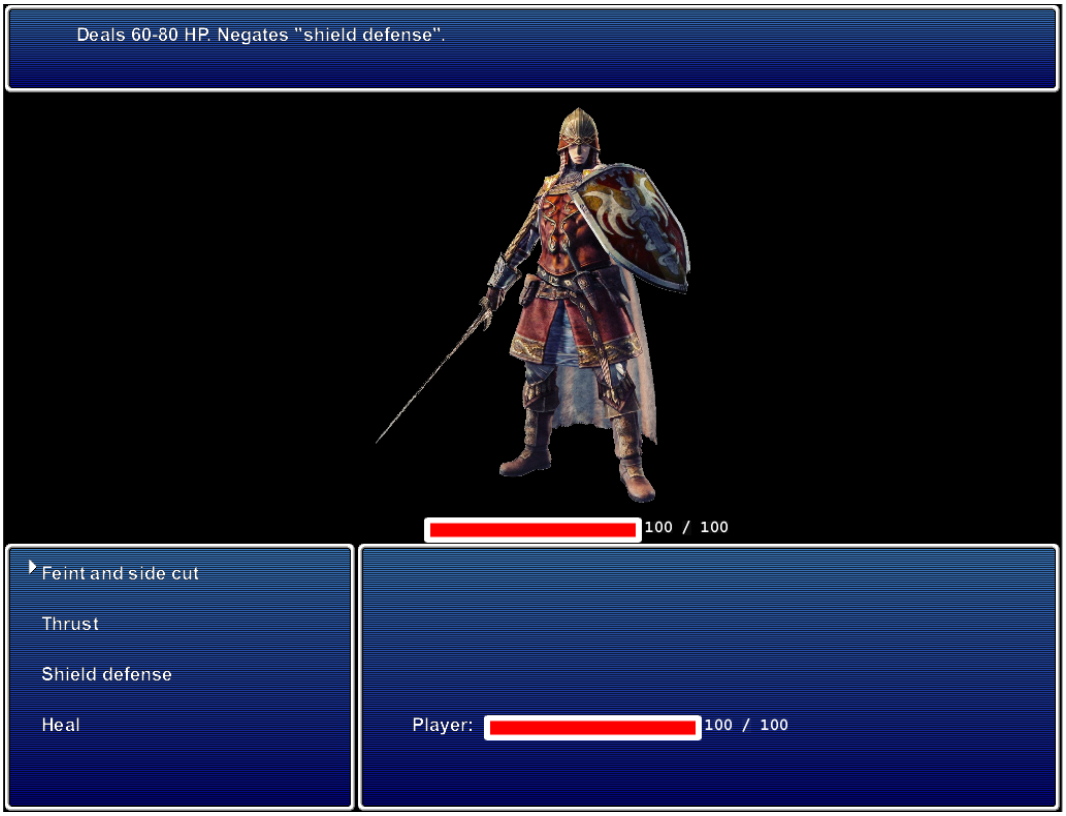


Ilustración 8: Pantalla de combate por turnos

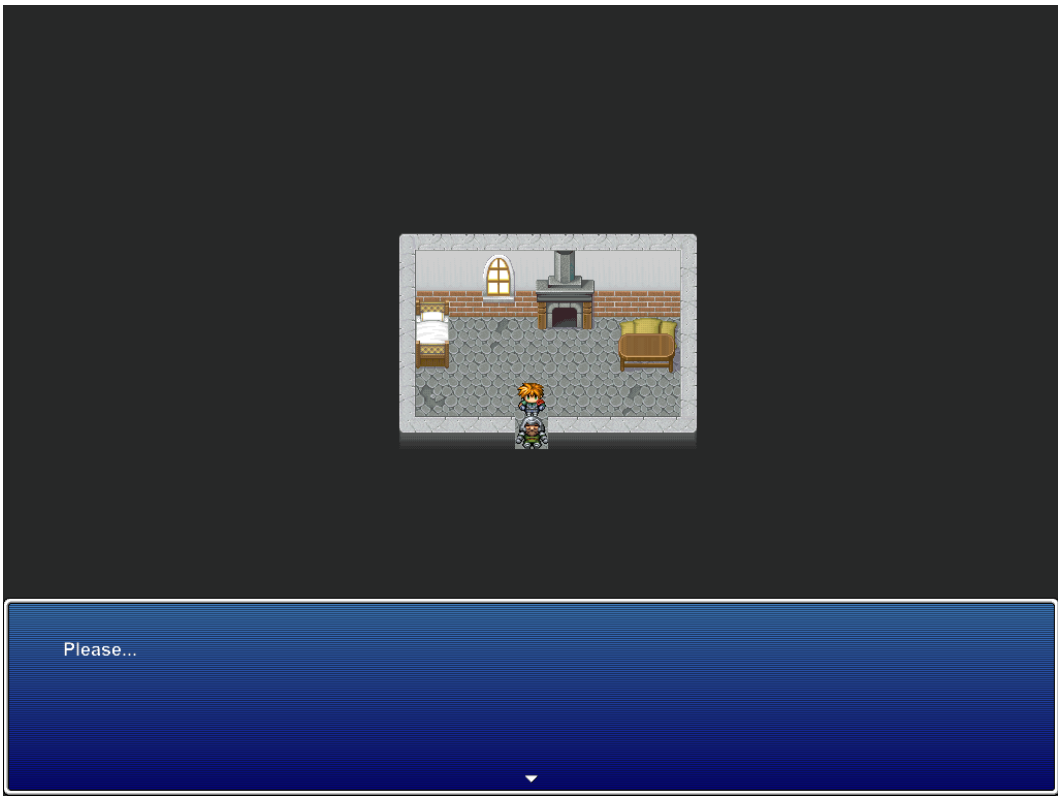


Ilustración 9: Pantalla de RPG

4.3. Extensión de puzle: JAGEtetrisplugin

La extensión de puzle permite crear un clon del Tetris original. Es una extensión bastante sencilla, puesto que aprovecha el movimiento por casillas de la extensión base. Uno de los principales cambios es la modificación del controlador de jugador implementando un sistema de estados (Ver Imagen 10).

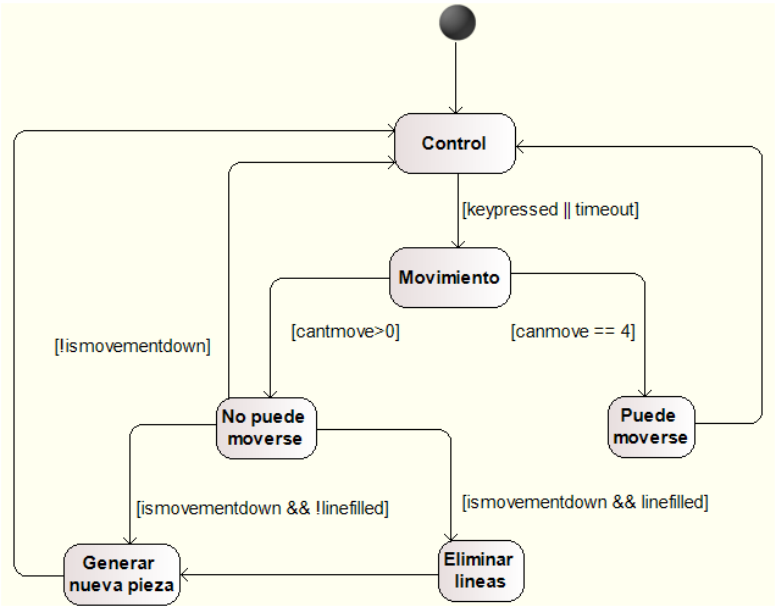


Ilustración 10: Diagrama de estados de tetris

Además de eso, crea dos estados de juego: uno que permite al jugador escribir un nombre con el que se guardará su puntuación, y otro que muestra las mejores puntuaciones. También modifica el menú principal y de juego para eliminar las opciones de guardar y cargar partida. Por último, implementa el comportamiento de las distintas piezas de Tetris mediante eventos individuales para cada uno de los cuatro cuadrados de cada pieza tetromino, de tal forma que cada cuadrado funciona teóricamente de forma independiente, pero con el sistema de estados diseñado, en realidad se mueven sincronizadamente, obteniendo la sensación de formar una única pieza. Es necesario definirlos independientes puesto que en caso de llenar lineas, se eliminan cuadrados, o piezas.

4.4. Extensión de acción: JAGEscrambleplugin

Por último, la extensión de acción permite crear un clon del Scramble original, y es la que más modifica las funcionalidades básicas. Primero de todo, no funciona mediante colisión por casillas, sino con formas geométricas. Por ello, todo el sistema de movimiento ha de ser cambiado, y se añaden figuras geométricas a cada evento para poder comprobar colisiones. Además, los mapas usan la funcionalidad del propio editor de mapas Tiled para indicar donde está cada evento de cada tipo, y las zonas en las que la nave no puede estar (ver Ilustración 11).

A la inclusión de formas que permitan la detección de colisiones, hay que añadir la propia detección. En el juego, esto lo hacen las clases del jugador, que comprueba si colisiona con el terreno o los enemigos, si detecta terreno, intenta moverse arriba, abajo izquierda o derecha según corresponda por la colisión, y si no puede muere, y en el caso de los enemigos, muere siempre que colisiona. Además, los proyectiles detectan si colisionan con los enemigos y los terrenos. Si colisiona con enemigos los destruye, y añade al jugador la puntuación y fuel, y en el caso de colisionar con terreno, simplemente se destruye a si mismo.

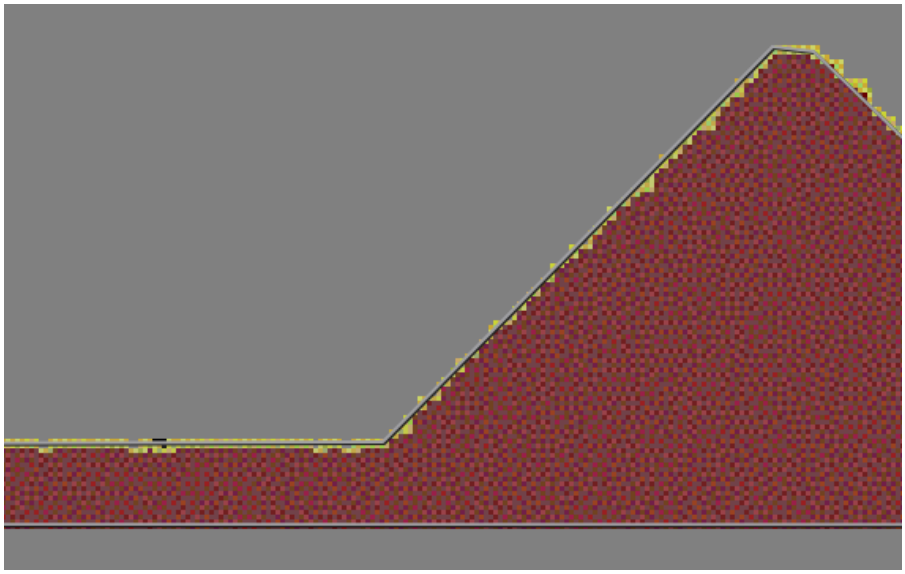


Ilustración 11: ejemplo de polígono que marca la parte colisionable de un mapa

Además, crea clases independientes para cada tipo de evento, puesto que su comportamiento es repetido en todos los eventos del mismo tipo, y es distinto para cada tipo de eventos, por lo que no tiene sentido crearlos individualmente como en el caso de los mapas RPG. También se ha de cambiar la cámara para convertirlo en un scroller lateral y que permita el dibujado de las estadísticas en pantalla.

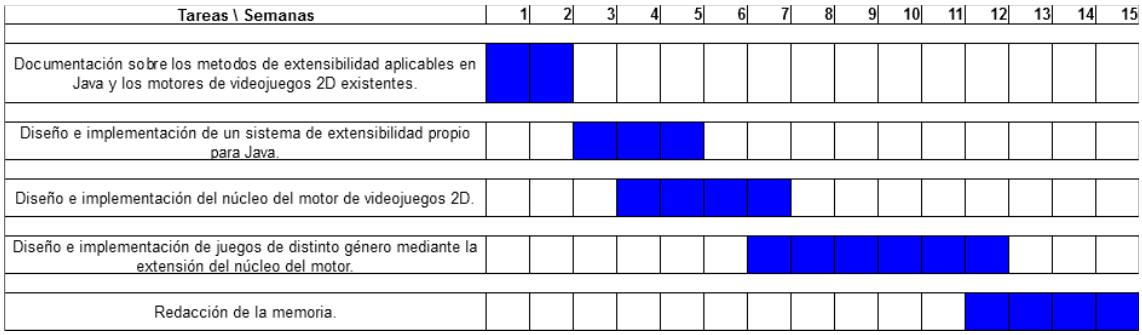
Por último, incorpora las pantallas de escribir un nombre para la puntuación y la pantalla de máximas puntuaciones de la extensión tetris, con ligeros cambios, puesto que el Scramble es un juego que se puede ganar o perder, mientras que en el tetris no.

5. CONCLUSIONES

Se ha implementado un motor 2D completo y muy flexible, demostrándolo mediante la creación de tres juegos con unas mecánicas muy diversas de movimiento, cámara y físicas. Sin embargo, implementar dicha flexibilidad ha ocasionado muchos problemas que en caso de crear un motor de propósito único no se hubieran dado, por lo que se ha aprendido mucho a depurar y encontrar errores en código que son difíciles de encontrar en compilación, porque dependen de cómo se haya creado el juego fuera del motor, bugs en casos especiales que no se manifiestan hasta que se dan unas condiciones muy concretas. Se ha dado el momento, por ejemplo, en el que la corrección de un bug ha creado dos, que se manifestaban los mismos problemas (En este caso, la posibilidad de moverse durante un diálogo), por lo que depurar para encontrar un bug, solucionarlo y descubrir que seguía existiendo y que no era causa de todo, fue realmente difícil.

5.1. Cronograma

En el cronograma proporcionado se puede observar que se esperaba que la mayor parte del tiempo se empleara en el diseño e implementación de juegos de distinto género. El tiempo final empleado en este apartado ha resultado ser mayor de lo esperado, teniendo en cuenta que la extensión base es la creación de un juego RPG. Sin embargo, de igual manera, se podría considerar que la extensión base es parte del núcleo del motor. En dicho caso, el tiempo del diseño e implementación de los juegos sería mucho menor (2-3 semanas) quedando el diseño e implementación del núcleo en unas 7 semanas de trabajo.



5.2. El futuro del proyecto

Este proyecto ha sido llevado a cabo por iniciativa propia, y por ello se desea seguir trabajándolo y ampliándolo en un futuro para que llegue a ser un motor que se pudiera incluso comercializar. Para ello, hay diversas ideas y proyectos a llevar a cabo:

1. Crear un editor de eventos que permita la fácil creación y edición de eventos sin tener que modificar archivos XML.
2. Proponer una alternativa a XML que permita no guardar todos los datos como archivos de texto, de forma que pueden ocupar menos y introducir algún método de seguridad que impida el robo de ideas por parte de otros usuarios creadores de juegos.
3. Crear una modificación del editor de mapas de código abierto Tiled que permita la creación de eventos, y por extensión de juegos completos, dentro del propio editor de mapas.
4. Implementar más tipos de eventos de los que ya hay, con el objetivo de conseguir la variedad que existe en RPG maker.
5. Crear un sistema de componentes de equipo, armas, habilidades y un sistema de batalla que implemente dichos componentes.
6. Crear un sistema de niveles con mejora de características por nivel.

5.3. Opinión personal

Desarrollar un motor de videojuegos de propósito general ha sido una experiencia completamente distinta a todo lo vivido anteriormente. Más que a la creación de un programa, se parece a la creación de una librería general que pueda ser modificada por el usuario para adaptarla a sus propias necesidades. Esto ha sido un verdadero choque frente a la filosofía que se aprende en la actualidad en la que se propone un problema y se busca la solución, y en la que se tienen casos de prueba muy definidos. En este caso, era necesario desarrollar teniendo en cuenta todas las posibilidades, y pensar todos los errores que puede cometer el usuario.

Por todo esto, ha sido una experiencia muy placentera, puesto que ha sido la primera vez que he sentido que creaba un programa completamente útil tanto ahora como en un futuro, y no algo que solo sirve en el momento de crearlo. He sentido que por fin he creado algo de lo que estoy orgulloso de llamar mío, puesto que puede ayudar a gente tanto en la creación de su propio motor, puesto que es software de código abierto, como de su propio juego.

Bibliografía

1. Hejlsberg, A., Wiltamuth, S., & Golde, P. (2003). *C# language specification*. Addison-Wesley Longman Publishing Co., Inc..

2. Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

3. Nystrom, R. (2014). Game programming patterns. Genever Benning.

4. Koenig, D., Glover, A., King, P., Laforge, G., & Skeet, J. (2007). *Groovy in action* (Vol. 1). Manning.

5. Knudsen, J. (1999). *Java 2D graphics*. " O'Reilly Media, Inc.".

6. Stearns, B. (2004). Java native interface. *Online Document*.

7. Flanagan, D. (2006). JavaScript: the definitive guide. " O'Reilly Media, Inc.".

8. Miller, F. P., Vandome, A. F., & McBrewster, J. (2009). Lua (programming language).

9. Shreiner, D., & Bill The Khronos OpenGL ARB Working Group. (2009). OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1. Pearson Education.

10. Van Rossum, G. (2007, June). Python Programming Language. In USENIX Annual Technical Conference (Vol. 41).

11. Stroustrup, B. (1986). The C++ programming language. Pearson Education India.

12. Flanagan, D., & Matsumoto, Y. (2008). *The ruby programming language*. " O'Reilly Media, Inc.".

13. Sweeney, T., & Hendriks, M. (1998). UnrealScript language reference. Epic MegaGames, Inc.

14. Sasada, K. (2005, October). YARV: yet another RubyVM: innovating the ruby interpreter. In Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (pp. 158-159). ACM.

ANEXO I: Descarga, compilación y ejecución.

El proyecto se puede descargar ya compilado o descargarlo y compilarlo. Para ambas opciones, primero es necesario ir a la página de GitHub del proyecto (https://github.com/valarion/JAGE). Para el primer caso, simplemente hay que ir a *releases* y descargar *JAGErpg*, *JAGEtetris* o *JAGEscramble* de la versión que se quiera, descomprimirlo y ejecutar JAGEcore-linux o JAGEcore-windows.bat según la plataforma en la que nos encontremos. Para ejecutarlo es necesario tener instalado Java 1.7.

En el caso de querer compilarlo, es necesario clonar el repositorio o descargar el código fuente. Ambas cosas se pueden hacer desde la página principal, en el botón verde "*Clone or download*"(Ver Ilustración 12).

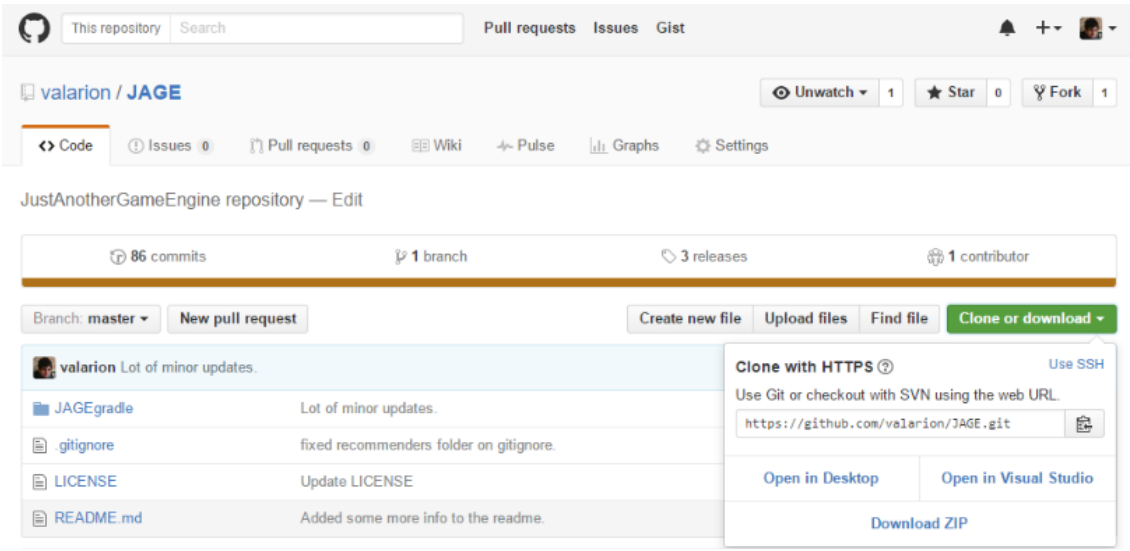


Ilustración 12: Descarga del código fuente

Una vez descargado el código fuente, si se tiene Gradle instalado, se necesita abrir una consola de comandos, dirigirse a la carpeta JAGEgradle y ejecutar por consola Gradle indicando el nombre de la tarea que queremos ejecutar (Ver Ilustración 13).

Las distintas tareas que existen son: *runrpg*, *runtetris*, *runscramble* permiten ejecutar cada juego directamente (si se añade *debug*, se ejecutará con depuración remota activada en el puerto 8080 a la espera de la conexión de un depurador para la ejecución del programa); *releaserpg*, *releasetetris* y *releasescramble* crea una carpeta que se puede distribuir directamente en JAGEgradle/build; *releaseAll* crea archivos zip comprimidos de los tres proyectos por separado, que son los que se pueden descargar desde la página de *relesases*. Además, existe la tarea *clean* que elimina todos los archivos creados por el proceso de compilación.

```
C:\Users\Ruben\Documents\GitHub\JAGE\JAGEgradle>gradle releaserpg
com.stehno.gradle.natives.NativesPluginExtension_Decorated@267f9765
:JAGEPluginSystem:compileJava UP-TO-DATE
:JAGEPluginSystem:processResources UP-TO-DATE
:JAGEPluginSystem:classes UP-TO-DATE
:JAGEPluginSystem:jar UP-TO-DATE
:JAGEcore:compileJava UP-TO-DATE
:JAGEcore:processResources UP-TO-DATE
:JAGEcore:classes UP-TO-DATE
:JAGEcore:jar UP-TO-DATE
:JAGEcore:startScripts UP-TO-DATE
:JAGEcore:installDist UP-TO-DATE
:copyCoreLibs
:copyCoreShortcuts
:JAGEmodules:compileJava UP-TO-DATE
:JAGEmodules:processResources UP-TO-DATE
:JAGEmodules:classes UP-TO-DATE
:JAGEmodules:jar UP-TO-DATE
:copyModules
:JAGEcore:distTar UP-TO-DATE
:JAGEcore:distZip UP-TO-DATE
:JAGEcore:assemble UP-TO-DATE
:JAGEcore:compileTestJava UP-TO-DATE
:JAGEcore:processTestResources UP-TO-DATE
:JAGEcore:testClasses UP-TO-DATE
:JAGEcore:test UP-TO-DATE
:JAGEcore:check UP-TO-DATE
:JAGEcore:build UP-TO-DATE
:JAGEcore:unpackNatives
:copyNatives
:copyRpgResources
:releaserpg

BUILD SUCCESSFUL

Total time: 8.481 secs
```

Ilustración 13: Ejemplo de compilación por consola de comandos.

Se pueden ejecutar varias tareas y Gradle decidirá el orden en el que deben ejecutarse. Ésto es muy útil sobre todo para limpiar los archivos de compilación y compilar de nuevo, para evitar errores de archivos incorrectos. Además, es necesario en el caso de querer depurar el código de forma remota (Ver Ilustración 14).

```
C:\Users\Ruben\Documents\GitHub\JAGE\JAGEgradle>gradle clean runtetris debug
com.stehno.gradle.natives.NativesPluginExtension_Decorated@267f9765
:clean
:JAGEPluginSystem:clean
:JAGEcore:clean
:JAGEeditor:clean
```

Ilustración 14: Ejemplo de limpieza y depuración de código.

En el caso de querer compilarlo en eclipse, se debe importar la carpeta JAGEgradle como proyecto Gradle, y ello importará todos los subproyectos. Una vez hecho esto, se debe crear una configuración de ejecución de tipo *"Gradle Project"*, usando como espacio de trabajo la propia carpeta JAGEgradle (Ver Ilustración 15) y usando como tareas las definidas anteriormente.

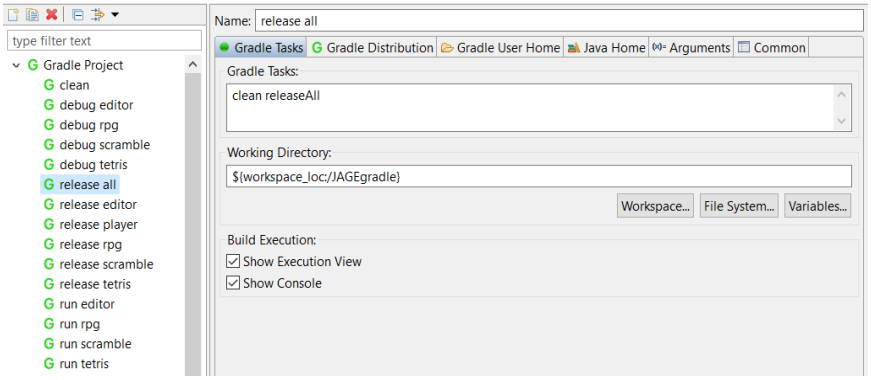


Ilustración 15: Ejemplo de configuración de ejecución de eclipse.

ANEXO II: Manual de usuario de RPG

Para ejecutar el juego RPG, es necesario descargar, o compilar, y ejecutar JAGErpg, mediante el archivo correspondiente al sistema operativo en el que se vaya a ejecutar. Una vez hecho, se mostrará la pantalla de selección de resolución en la que ejecutar el juego y, una vez seleccionada, se verá la pantalla principal del juego como la mostrada en la Ilustración 16, y se oirá el sonido de lluvia correspondiente a dicha pantalla.

Ilustración 16: Pantalla principal RPG

En esta pantalla existen 3 opciones, aunque en la primera ejecución solo se pueden usar dos: *nueva partida*, *cargar partida* y *salir*. *Salir* permite cerrar el juego y salir al escritorio. *Cargar partida* permite seleccionar una partida guardada anteriormente para continuar jugando, usando mediante el menú mostrado en la ilustración 17.

Por último, *nueva partida* permite iniciar un juego nuevo. Una vez iniciada, podremos ver la primera pantalla del juego, una pantalla negra en la que van apareciendo letras (Ver Ilustración 18) Los controles en los diálogos se pueden observar en la Tabla 1.

Una vez acabado el diálogo inicial, se podrá ver en pantalla el interior de una casa, con un personaje dentro de la cama (Ver Ilustración 19). Éste es el que controla el jugador. En esta pantalla, los controles usados se pueden ver en la Tabla 2.

Ilustración 17: Pantalla de cargar partida

Control	Acción
Intro (Pulsar)	Aceptar
Intro (Mantener)	Aumentar velocidad de diálogo
Escape	Cancelar

Tabla 1: Controles juego RPG diálogos

Control	Acción
Flecha arriba	Mover una casilla hacia arriba
Flecha abajo	Mover una casilla hacia abajo
Flecha izquierda	Mover una casilla hacia la izquierda
Flecha derecha	Mover una casilla haca la derecha
Intro	Aceptar/interactuar
Escape	Cancelar
Shift izquierdo	Andar más rápido

Tabla 2: Controles juego RPG mapas

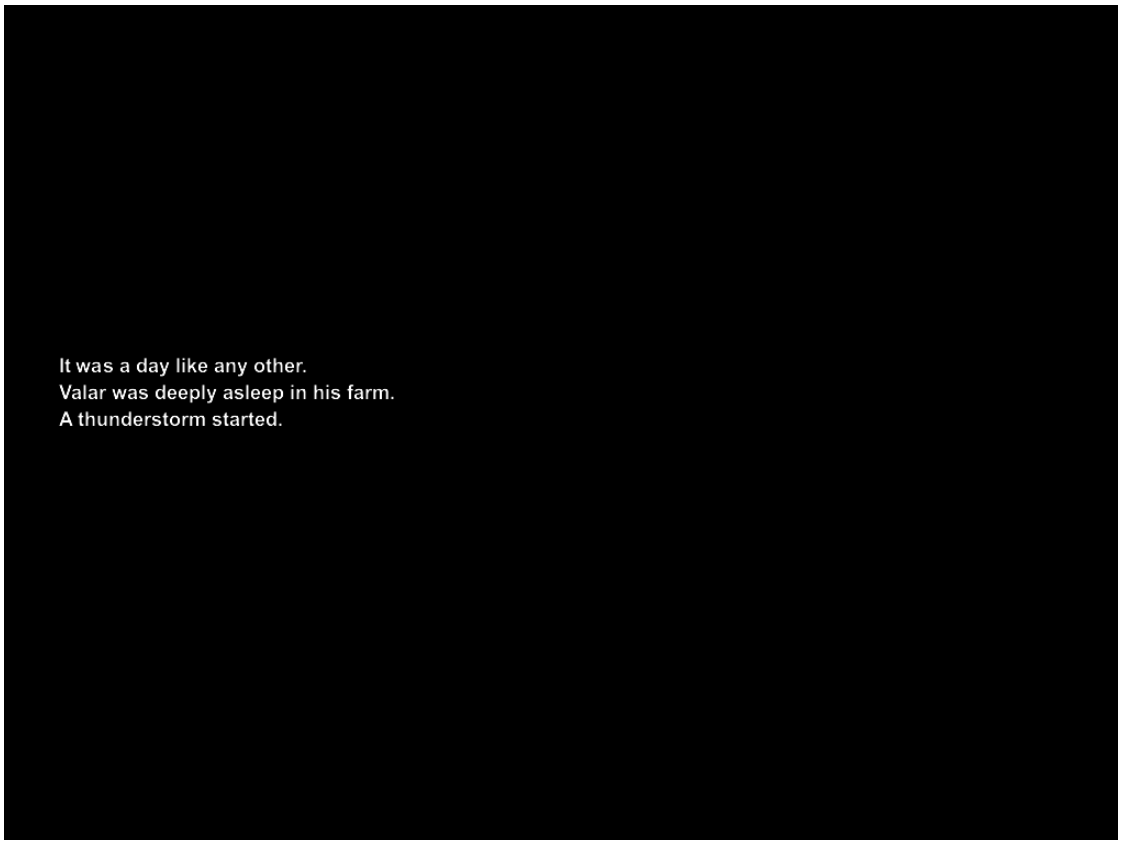


Ilustración 18: Juego iniciado.

Ilustración 19: Interior

Mientras se juega, llegará un momento en el que será necesario luchar. Para ello, se mostrará la pantalla de batalla (Ver Ilustración 20). En ella, el usuario puede elegir entre cuatro acciones, que en español significan: *Finta y corte lateral*, *estocada*, *defensa con escudo*, y *curar*. Tanto el jugador como la máquina elegirán una acción, que se ejecutaran simultáneamente. Las tres primeras tienen una mecánica de piedra-papel-tijera: f*inta y corte lateral* niega la acción de *defensa con escudo*, *defensa con escudo* niega la acción de *estocada*, y *estocada* niega la acción de *finta y corte lateral*. Además, *finta y corte lateral* hará entre 60 y 80 de daño siempre que no sea negada; *estocada* hará entre 50 y 60 siempre que no sea negada, y *defensa con escudo* hará entre 10 y 30 de daño siempre que haya negado una acción enemiga (es decir, siempre que el enemigo haya hecho una *estocada*). Por último, *curar* regenera entre 55 y 65 puntos de daño independientemente de lo que haga el contrario.

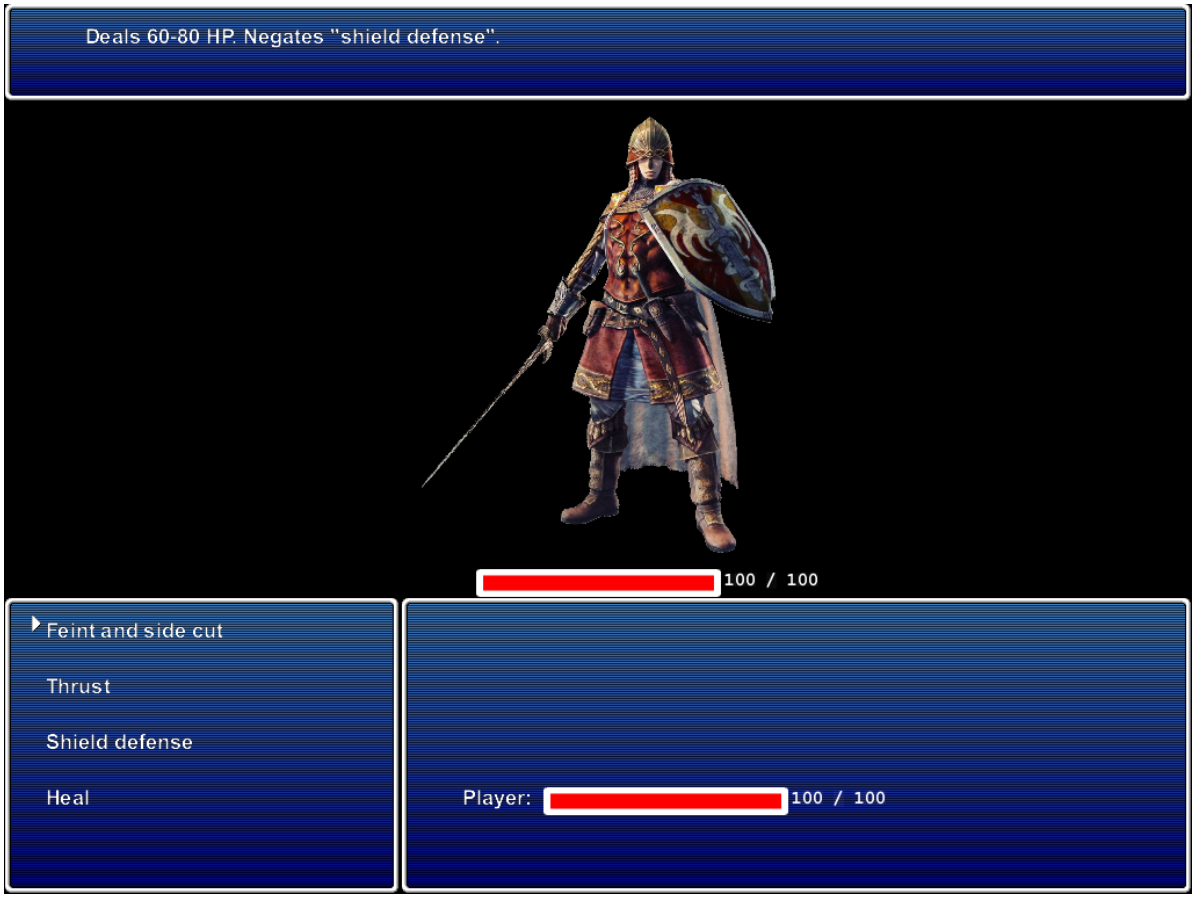


Ilustración 20: Pantalla de batalla

Una vez elegida una acción, se podrá ver por pantalla qué ha elegido el ordenador, y cual ha sido el resultado, incluyendo qué acciones se han negado, cuanto daño se ha recibido y realizado, y cuanto daño se ha curado cada uno (Ver Ilustración 21). En el caso de ganar la batalla, el enemigo desaparecerá del mapa y se podrá continuar el juego. En el caso de perder la batalla, se pueden elegir dos opciones (ver Ilustración 22): continuar el juego, en cuyo caso el jugador aparecerá en el mapa anterior y tendrá la posibilidad de volver a luchar contra el mismo enemigo; o salir a la pantalla de inicio y perder todo el proceso no guardado.

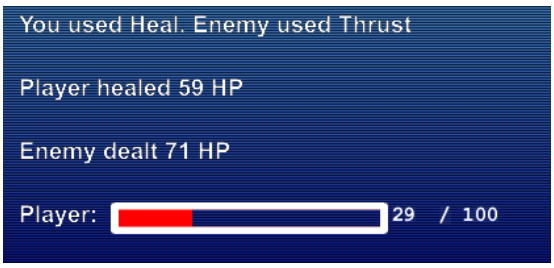


Ilustración 21: Resultado de la acción escogida.

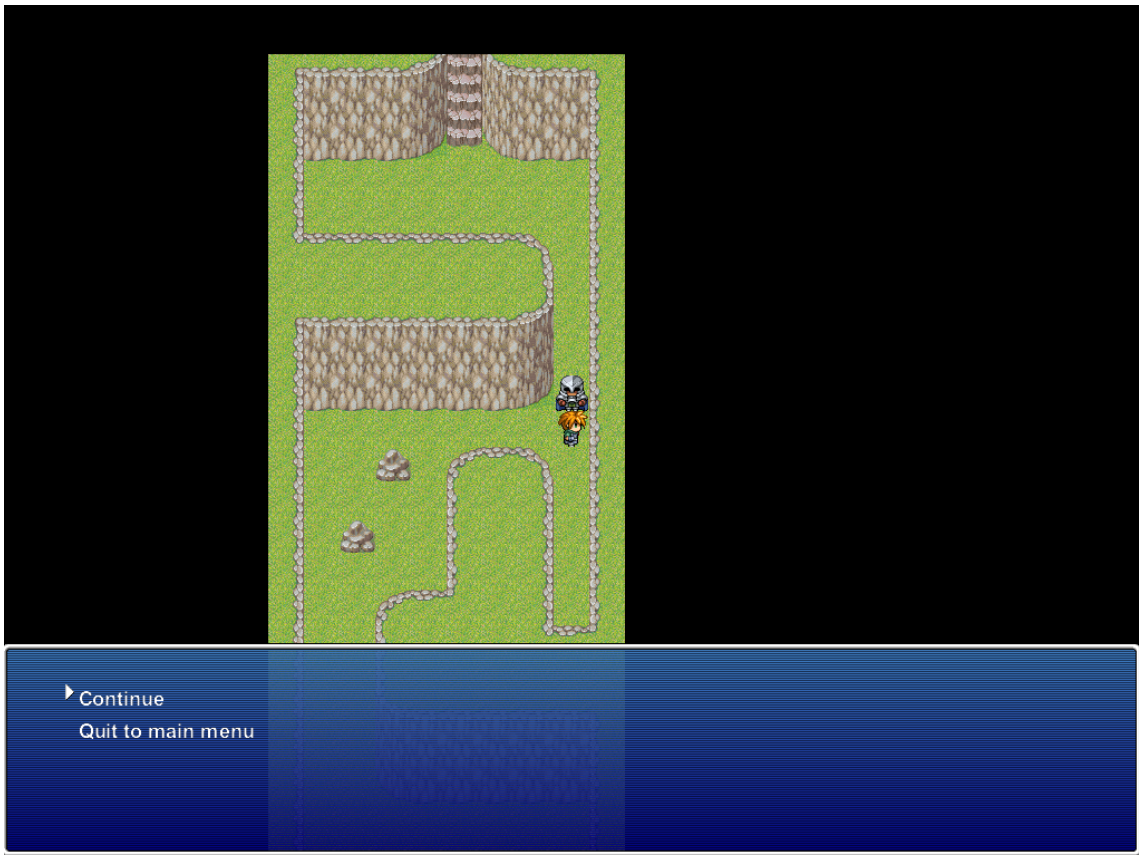


Ilustración 22: Opciones en caso de derrota

ANEXO III: Manual de usuario Tetris

Para ejecutar el juego Tetris, es necesario descargar, o compilar, y ejecutar JAGEtetris, mediante el archivo correspondiente al sistema operativo en el que se vaya a ejecutar. Una vez hecho, se mostrará la pantalla de selección de resolución en la que ejecutar el juego y, una vez seleccionada, se verá la pantalla principal del juego como la mostrada en la Ilustración 23.

Ilustración 23: Pantalla principal Tetris

En esta pantalla existen 3 opciones: *nueva partida*, *ver puntuaciones* y *salir*. *Salir* permite cerrar el juego y salir al escritorio. *Ver puntuaciones* permite ver una lista ordenada de las mejores puntuaciones obtenidas en el juego (Ver Ilustración 24).

3480	win	Valarion
0	win	Anonymous

Ilustración 24: Pantalla de puntuaciones

Por último, *nueva partida* permite iniciar un juego nuevo. Una vez iniciada, podremos ver la primera pantalla del juego, una pantalla con un espacio vacío en medio donde aparecerá una pieza, a la izquierda, la próxima pieza a salir, y a la derecha la estadísticas de la partida (Ver Ilustración 25). Los controles se pueden ver reflejados en la tabla 3.

El juego consiste en conseguir la máxima puntuación con el mínimo número de filas, pues cuantas más filas se consiguen, más rápido caen las piezas. Se consiguen filas llenando una fila entera con piezas, de forma que no quede ningún cuadrado blanco. Cuando una fila se llena, se elimina. Se pueden llenar hasta cuatro filas a la vez, cuantas más filas a la vez, mayor es la puntuación que se consigue. El juego termina cuando la siguiente pieza no puede aparecer.

Control	Acción
Flecha izquierda	Mover pieza hacia la izquierda
Flecha derecha	Mover pieza hacia la derecha
Flecha abajo	Bajar pieza más rápido
Flecha arriba	Girar pieza
Intro	Bajar pieza hasta abajo
Escape	Mostrar el menú

Tabla 3: Controles Tetris

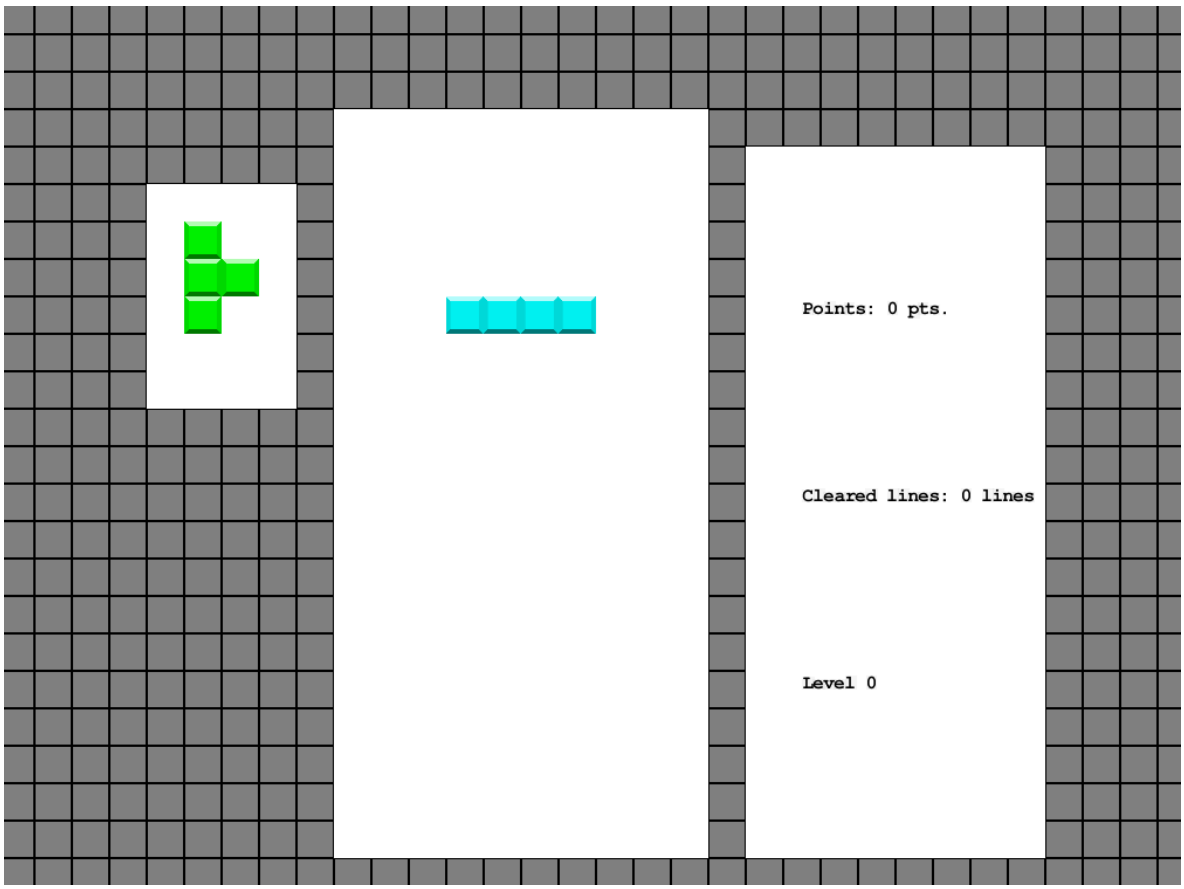


Ilustración 25: Pantalla de juego Tetris

ANEXO IV: Manual de usuario Scramble

Para ejecutar el juego Scramble, es necesario descargar, o compilar, y ejecutar JAGEscramble, mediante el archivo correspondiente al sistema operativo en el que se vaya a ejecutar. Una vez hecho, se mostrará la pantalla de selección de resolución en la que ejecutar el juego y, una vez seleccionada, se verá la pantalla principal del juego como la mostrada en la Ilustración 26.

Ilustración 26: Pantalla principal Scramble

En esta pantalla existen 3 opciones: *nueva partida*, *ver puntuaciones* y *salir*. *Salir* permite cerrar el juego y salir al escritorio. *Ver puntuaciones* permite ver una lista ordenada de las mejores puntuaciones obtenidas en el juego (Ver Ilustración 27).

18860	loose	Anonymous
120	loose	Anonymous

Ilustración 27: Pantalla puntuaciones Scramble

Por último, *nueva partida* permite iniciar un juego nuevo. Una vez iniciada, podremos ver la primera pantalla del juego, una pantalla con una nave espacial, controlada por el usuario, y estadísticas en las partes superior e inferior de la pantalla (Ver Ilustración 39). Los controles de juego se pueden apreciar en la Tabla 4.

Control	Acción
Flecha arriba	Moverse hacia arriba
Flecha abajo	Moverse hacia abajo
Flecha derecha	Acelerar
Flecha izquierda	Frenar
X	Disparar láser
Z	Lanzar bomba
Intro	Aceptar
Escape	Cancelar

Tabla 4: Controles Scramble

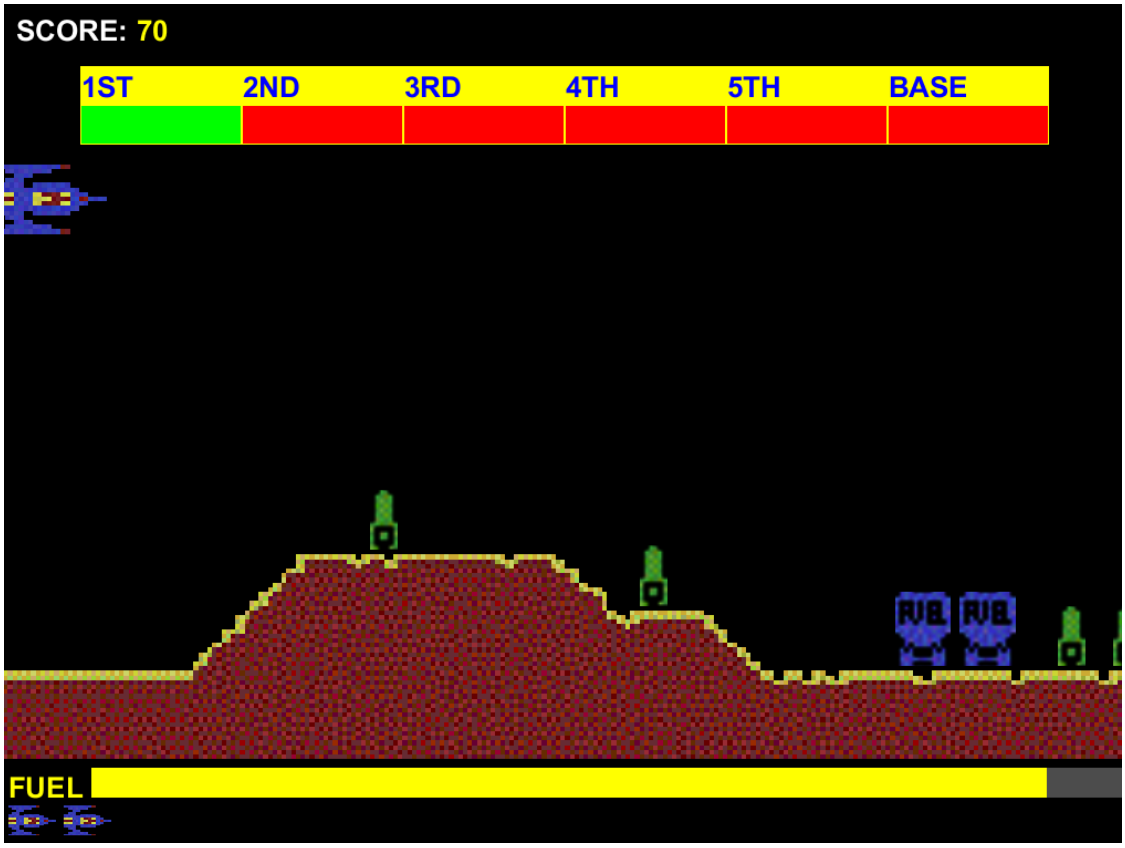


Ilustración 28: Pantalla de juego Scramble

Todos los enemigos del juego pueden ser destruidos a excepción de uno. En el tercer nivel existen unas bolas de fuego (Ver Ilustración 29), que se generan en una altura aleatoria y que se mueven hacia el jugador que son imposibles de destruir mediante láser y bombas, por lo que será necesario esquivarlos.

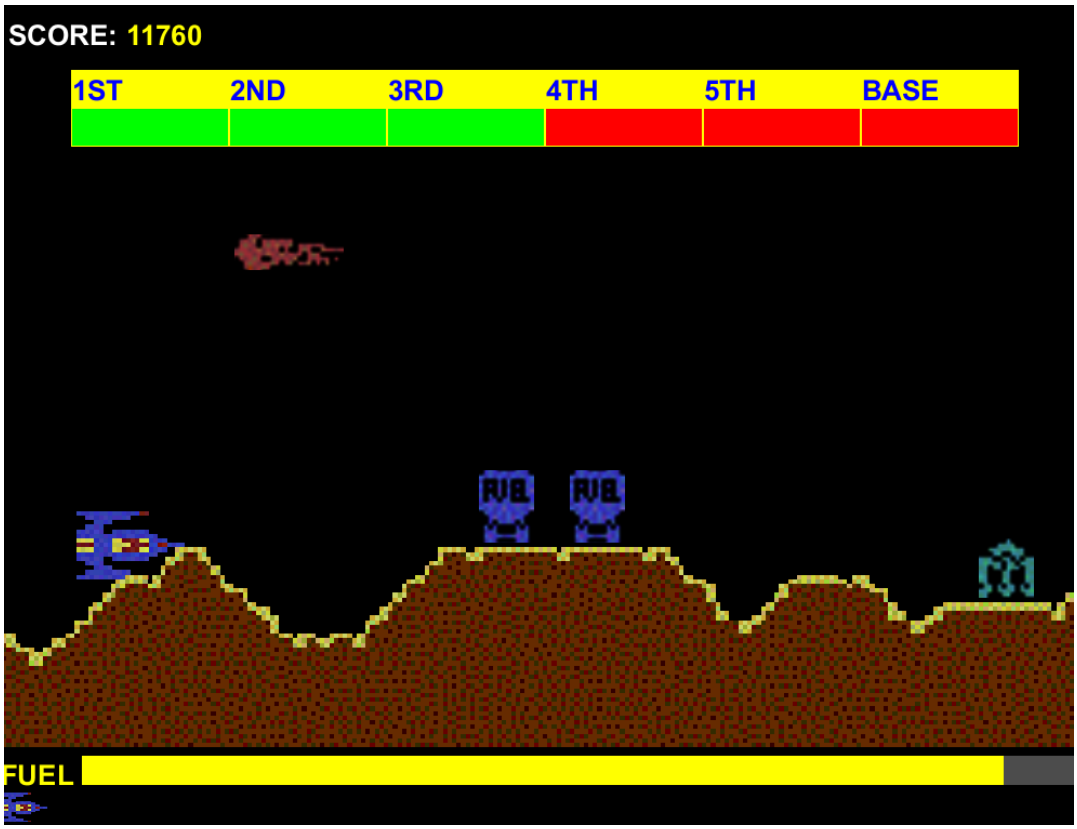


Ilustración 29: Ejemplo de enemigo indestructible

El objetivo del juego es llegar al último nivel, "BASE", y destruir uno de los tres enemigos finales que existen en dicho nivel (Ver Ilustración 30). A la vez, el objetivo secundario es conseguir la máxima puntuación posible. El juego termina cuando el jugador muere tres veces, cada vez que muere, se reinicia desde el nivel en el que se encontraba.

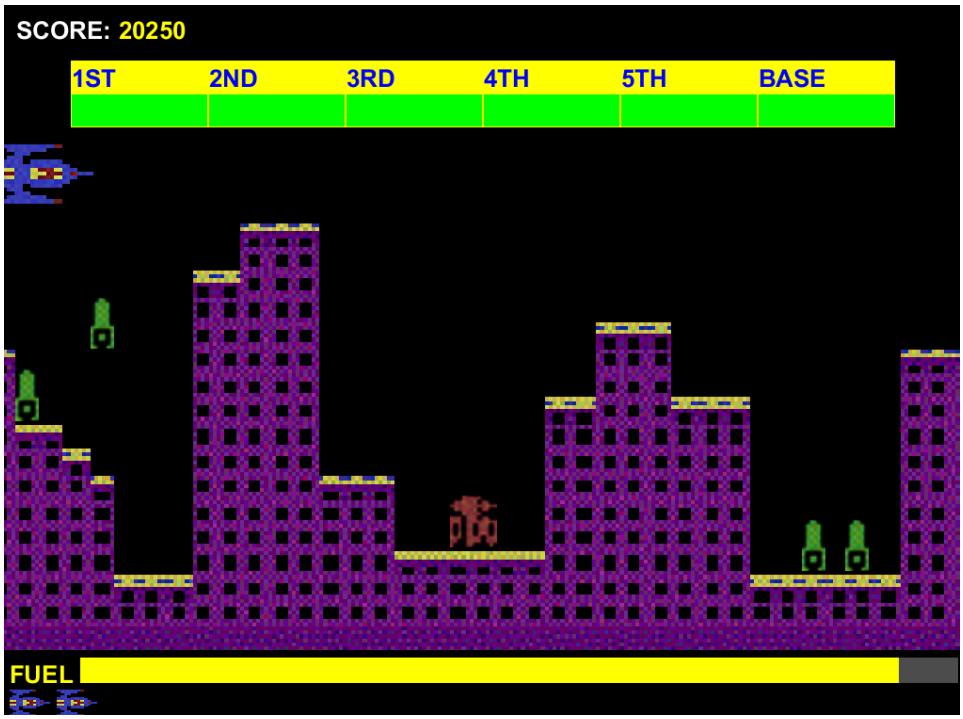


Ilustración 30: Ejemplo de enemigo final (en rojo)

El primer nivel del juego sienta las bases del funcionamiento del resto del juego (cohetes verdes, fuel y enemigos misteriosos que dan una puntuación aleatoria), pero cada nivel Añade una dificultad a ésto y única para cada nivel (es decir, no se acumulan). En el segundo nivel, existen unos ovnis que se mueven rápidamente y constantemente en vertical. En el tercero, existen bolas de fuego indestructibles que se generan aleatoriamente. En el cuarto, la dificultad está en el reducido espacio para maniobrar. Y por último en el quinto, la dificultad añadida consiste en conseguir despejar el camino a tiempo. El último nivel, la base, no tiene ninguna dificultad añadida, únicamente un objetivo distinto, destruir el enemigo final. Cuando la pantalla cambie de color (Ver Ilustración 31) significará que se ha cambiado de nivel y que es necesario cambiar de táctica.

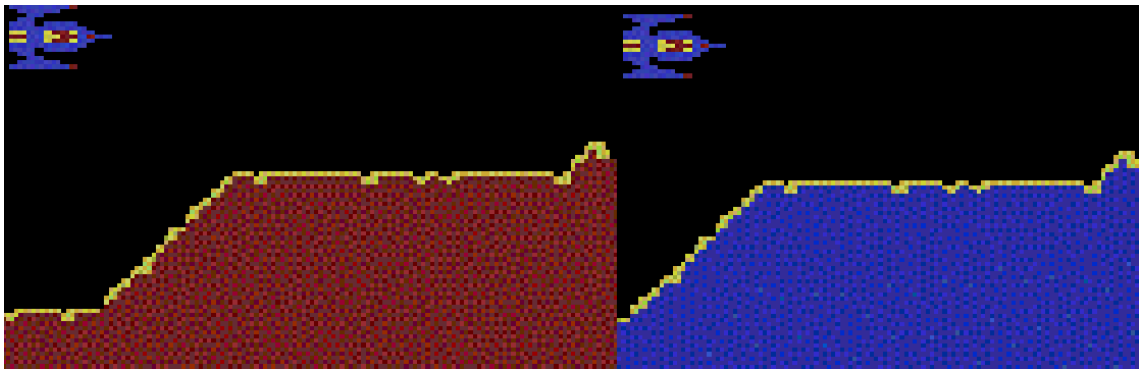


Ilustración 31: Cambio de color de la pantalla