ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

TURING'S ADVENTURE:
Videojuego educativo para aprender a desarrollar videojuegos multiplataforma en Java con LibGDX

Javier Villegas Gómez

13 de septiembre de 2013

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

TURING'S ADVENTURE:
Videojuego educativo para aprender a desarrollar videojuegos multiplataforma en Java con LibGDX

- Departamento: Lenguajes y Sistemas informáticos
- Departamento: Ingeniería Informática
- Autor del proyecto: JAVIER VILLEGAS GÓMEZ
- Director del proyecto: MANUEL PALOMO DUARTE

Cádiz, 13 de septiembre de 2013

Fdo: Javier Villegas Gómez

Agradecimientos

A mis padres, por todo el apoyo durante la realización del proyecto.

A Jose Antonio, por su confianza y la magnífica oportunidad que me ha concedido.

A todos mis amigos y compañeros de la carrera por sus ánimos e inspiración para el diseño y gráficos.

A mi tutor, por su paciencia y consejos.

Licencia

Resumen

Cada vez el ser humano dedica más tiempo y recursos económicos al ocio. Entre las diferentes posibilidades destacan los videojuegos, que en los últimos años han pasado de ser un entretenimiento para jóvenes a usarse por personas de un rango de edades mucho más amplio. Gran parte de este cambio se debe al aumento de las posibilidades de los teléfonos móviles, que permiten jugar en prácticamente cualquier lugar.

El desarrollo de videojuegos es una actividad informática compleja debido a la cantidad de campos que confluyen en ella: programación, simulación física, gestión de sonido y música, etc. Si además queremos que el videojuego pueda usarse en diversas plataformas (tanto ordenadores de sobremesa como dispositivos móviles), se complica aún más.

El objetivo de este Proyecto Fin de Carrera es desarrollar un videojuego con fines educativos que permita a alumnos de Ingeniería Informática aprender a programar videojuegos con la biblioteca libre LibGDX. Para ello el videojuego tendrá 4 niveles en que se aplicarán las principales técnicas de desarrollo de videojuegos progresivamente:

- Primer nivel: aprenderemos a mostrar imágenes por pantalla, a gestionar la entrada de ratón/pantalla táctil, y a crear unas colisiones básicas en cuanto a captar el puntero y chocar con los bordes de pantalla.
- Segundo nivel: gestión de animaciones, entrada por teclado, botones en dispositivos Android y la lógica de pulsación.
- Tercer nivel: gestión de física y colisiones avanzadas.
- Cuarto nivel: gestión del mundo (elementos coleccionables, vidas, etc).
- General: guardado de datos permanentes y gestión de usuarios.

Índice general

Índice de figuras

Parte I

Prolegómenos

Capítulo 1

Introducción

1.1. Motivación

La idea del Videojuego Turing's Adventure surgió con la preparación del taller "Introducción a la programación de Videojuegos con **Java** y **LibGDX**", en el que realicé los materiales necesarios para explicar las nociones básicas para que cualquier persona con un mínimo de conocimientos de programación pudiese crear un videojuego. Dichos materiales no podían considerarse un videojuego como tal, apenas eran un pequeño nivel en el que debíamos dotar a nuestro personaje de física y colisiones, explicando dicho proceso paso a paso.

El taller tuvo muy buena aceptación, pero aún así, en tan poco tiempo solo pudimos dar un par de pinceladas a la estructura general de los videojuegos, la impresión de imágenes por pantalla, el movimiento y una breve introducción a las colisiones.

Pude comprobar que algunos alumnos se quedaron con ganas de mas información con respecto a la librería y al desarrollo de videojuegos. En concreto por su facilidad para generar juegos multiplataformas, ya que es poco frecuente encontrar librerías de este tipo. Y a pesar de que hay buena documentación sobre LibGDX, se echan un poco en falta manuales o tutoriales de introducción, que permitan a programadores noveles iniciarse en esta disciplina.

Es verdad que existen sistemas que nos permiten aprender la realización de juegos multiplataformas en Java como JavaSoccer o Robocode (para este último incluso realicé un taller previo al de LibGDX), pero no es exactamente lo que buscamos.

Robocode es un videojuego ya realizado, basado en combate de tanques, en el que aprenderemos a insertar un comportamiento específico en cada uno de ellos (su movimiento general, sus reacciones al recibir daño, chocar con otro tanque o con la pared, sus métodos de ataque y defensa...), partiendo desde instrucciones básicas hasta llegar a la realización de sistemas expertos complejos.

JavaSoccer sigue el mismo patrón pero basado en el entorno de un partido de fútbol en el que deberemos desarrollar el comportamiento de los jugadores.

Nosotros en cambio buscamos la forma de poder crear un juego desde cero, a nuestro antojo, sin depender de un juego con un objetivo ya establecido.

Por ello me pareció buena idea aprovechar el material del taller, convertirlo en un videojuego, pero enfocándolo al aprendizaje. Todo explicado de manera sencilla para que aquellos que decidan adentrarse en este mundo tengan algo de donde sacar ideas.

Así pues, Turing's Adventure es un videojuego que, siguiendo los tutoriales proporcionados y los comentarios del propio código, se puede tomar como referencia para iniciar un nuevo proyecto y aprender las nociones básicas del funcionamiento de un videojuego.

1.2. Objetivo

Como ya hemos dicho, el objetivo del proyecto no es la creación de un videojuego en sí, sino la realización de una guía para que cualquier programador novato (o con poca experiencia en el desarrollo de videojuegos), pueda crear un videojuego desde cero.

Todo se irá explicando de forma progresiva, desde lo mas básico, como mostrar elementos por pantalla, pasando por una introducción a físicas y colisiones hasta el almacenamiento de datos externos al juego y la forma de implementar distintos usuarios.

Esto lo haremos usando los niveles del propio juego, ya que cada uno será distinto al anterior, mejorando los conceptos explicados en el nivel anterior o introduciendo nuevos conceptos.

1.3. Ambientación

¿Un videojuego ambientado en la vida de Alan Turing?

El motivo es que en 2012 se conmemoró el centenario del nacimiento de Alan Turing, con motivo de tal evento se financió el taller mencionado anteriormente, por lo que era de esperar que el resultado estuviera ambientando en nuestro informático preferido.

Por supuesto, el juego no se puede considerar como una biografía, ya que el objetivo es la enseñanza y no la redacción de las memorias de Turing, pero si que se han utilizado algunos de los momentos mas significativos de su vida para ambientar los niveles que componen el videojuego.

Capítulo 2

Conceptos Básicos

2.1. Sistemas Similares

Antes de realizar este proyecto, se ha investigado sobre sistemas similares sobre los que tomar un modelo de referencia. La idea era buscar otras herramientas que permitan el aprendizaje de desarrollo de videojuegos, a ser posible multiplataformas, e intentar aprovechar la metodologia de enseñanza empleada adaptandola a nuestras necesidades.

Primero se investigó sobre entornos cerrados: videojuegos ya realizados que pretenden enseñar a desarrollar alguna parte concreta del mismo, como por ejemplo:

- **Robocode** (Fig. 2.1)(`http://robocode.sourceforge.net/`): es un juego de programación multiplataforma, donde el objetivo es desarrollar un robot-tanque para combatir contra otros tanques en Java o .NET. En el, el objetivo es aprender a insertar un comportamiento específico en cada uno de los tanques: los dotaremos de un patrón de movimiento general, podemos establecer mediante instrucciones propias del sistema sus reacciones al recibir daño, chocar con otro tanque o con la pared, sus métodos de ataque y defensa, etc, de manera que crearemos una inteligencia o comportamiento especifico para cada uno de ellos, pudiendo complicarlo tanto como queramos, partiendo desde instrucciones básicas hasta llegar a la realización de sistemas expertos complejos. Para este sistema incluso se desarrolló un taller previo al de LibGDX. (RoboUca).

- **JavaSoccer** (`http://eia.udg.es/~iitap/cursjava/javasoccer.html`): Este juego sigue una estructura muy similar al anterior, también desarrollando en Java, solo que en lugar de ambientarse en combate, lo hace en un juego de fútbol. Aprenderemos a programar el comportamiento de jugador, hacerlo mas agresivo o defensivo según su equipo vaya ganando o perdiendo, sus acciones una vez en posesión del balón, etc.

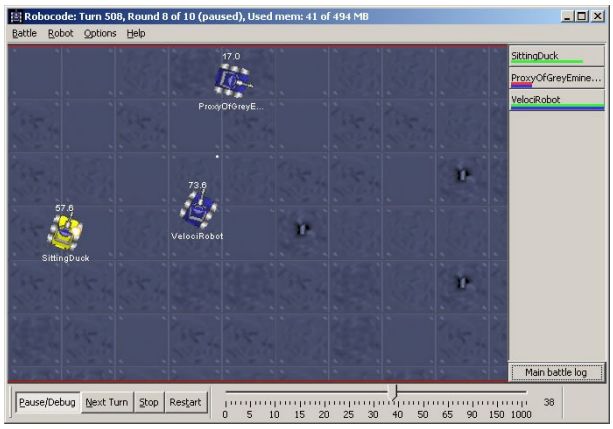


Figura 2.1: Ejemplo de sistemas similares: Robocode

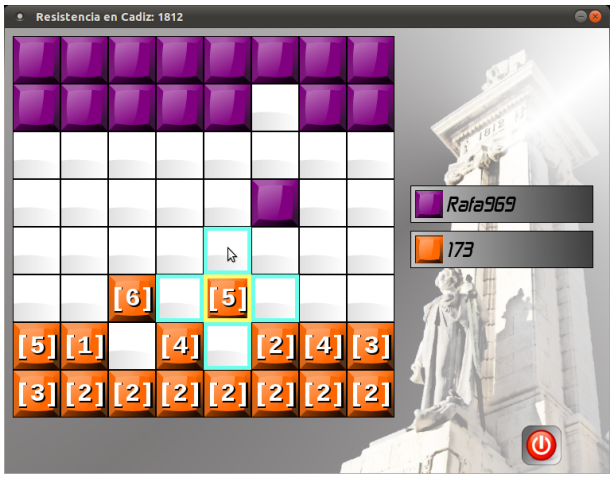


Figura 2.2: Ejemplo de sistemas similares: GadesSiege

– **Gades Siege** (Fig. 2.2) (`https://code.google.com/p/gsiege/`): Desarrollado en Python. También sigue la misma metodología, esta vez basado en un juego de estrategia similar al ajedrez, tendremos fichas numeradas de manera que cada ficha puede comer a otra ficha enemiga de igual o menor valor, pero no podremos ver el valor de las fichas enemigas y seremos derrotados cuando perdamos la ficha de valor '1'. Aprenderemos por lo tanto a diseñar una estrategia para nuestro modo de juego. Siendo mas arriesgados avanzando con las fichas superiores corriendo el riesgo de perderlas, adoptando una actitud defensiva y defendiendo nuestro '1' o de la manera que queramos.

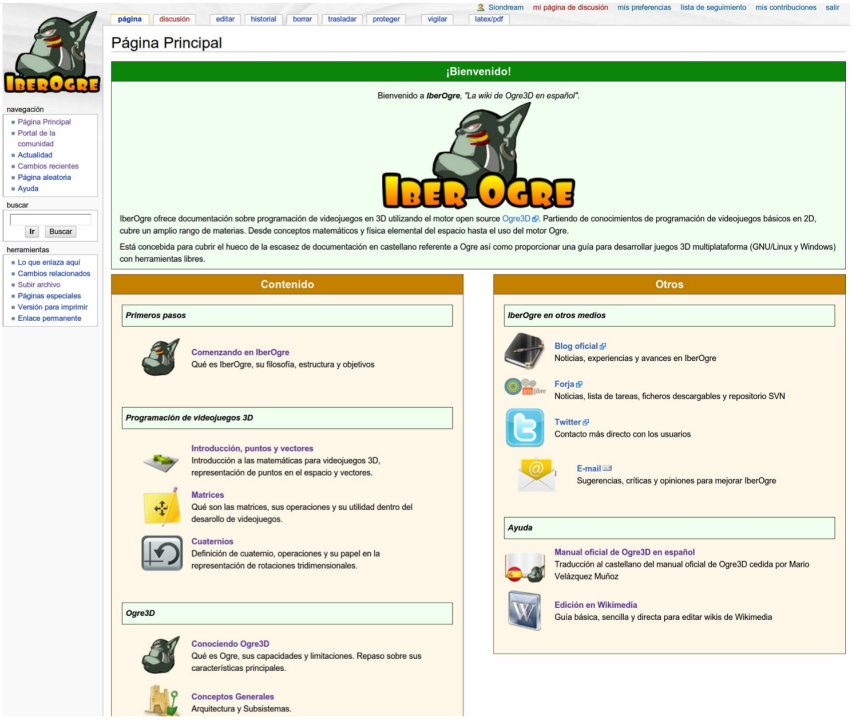


Figura 2.3: Ejemplo de sistemas similares: IberObre, Wiki sobre Ogre3D

Son herramientas muy útiles, ya que permiten al programador novato tener un primer contacto con el desarrollo de videojuegos, empezando por instrucciones básicas y avanzando en la complejidad de las mismas.

Pero en cambio no cumplen con el objetivo que estamos buscando. Mediante estos sistemas el programador no puede hacer más que diseñar mas tanques, jugadores, o elementos pertenecientes al juego y nosotros queremos poder crear un videojuego desde cero. Con nuestras reglas y personajes. Sea de estrategia, de combate de tanques, plataformas, de disparos o como queramos. Por lo que continuamos con la investigación.

Después se buscó información sobre sistemas abiertos, que proporcionen algun manual o wiki sobre el manejo de librerias de desarrollo de videojuegos. Algo que permitiera el desarrollo desde el comienzo del juego, algunas de las encontradas fueron:

– **IberOgre** (Fig. 2.3)(`http://osl2.uca.es/iberogre`): Es una Wiki sobre Ogre3D, una librería de desarrollo de videojuegos en 3D con C++. Desarrollada como proyecto de fin de carrera por David Saltares, un compañero de Ingeniería Informática. En ella se detalla de forma excelente la instalación y manejo básico de la librería, la creación de escenas, la realización de animaciones, definición materiales, gestión de nodos, particulas, luces, sombras, entorno y audio.

El resultado es una buena forma de aprender a desarrollar videojuegos en 3D con Ogre, aunque tal vez algo mas enfocada a usuarios mas experimentados, ya que para alguien que lleve poco tiempo en el mundo de la programación podría resultarle abrumadora la gestión de un sistema 3D y preferir centrarse en aspectos más básicos (como el movimiento y las colisiones) que permitan la creación de un juego simple, teniendo luego por supuesto la opción de seguir avanzando.

– **Tutorial de libSDL para la programación de videojuegos**(http://wikis.uca.es/wikijuegos/w): Mediante su proyecto de fin de carrera, nuestro compañero Antonio García nos enseña a realizar videojuegos con SDL en C++. En este tutorial se centra principalmente en la descricpcion exhaustiva de la libreria y de todos sus componentes (instalación, subsistemas, tipos de datos, animaciones...) y terminando con un ejemplo de un nivel, describiendo paso a paso todos los elementos necesarios para realizarlo.

Ambos ejemplos se acercan ya bastante a nuestro objetivo, pero aún entontramos algunas cosas que se podrían adaptar mejor a nuestros propósitos. Por ejemplo, a pesar de que ambas librerias descritas arriba (Ogre3D y libSDL) son, al estar realizadas en C/C++, la instalación y ejecución en distintos sistemas requiere de procesos distintos, lo que nos obliga a crear una forma diferente de instalación para cada sistema, corriendo así el riesgo de que por razones de arquitectura en alguno de ellos falle o no funcione como esperamos. Por lo tanto nosotros intentamos buscar una herramienta que si que nos garantice la portabilidad directa entre sistemas.

Ademas una carencia que encontramos en los ejemplos es que no admite la portabilidad a sistemas móviles. La inclusión de los dispositivos móviles al mercado de los videojuegos es relativamente reciente, por lo que hasta ahora ha sido algo mas complicado el desarrollo de videojuegos para ellos. Es por lo tanto una razon mas para buscar una forma de realizar un juego en el que solo haga falta desarrollarlo una vez para que nos sirva en cualquier plataforma, incluida móvil.

2.2. Libreria utilizada: LibGDX

Debido a los motivos descritos anteriormente, hemos decidido elegir LibGDX ya que al estar basado en Java nos permite abstraernos completamente del sistema operativo, llegando así al mayor número de personas, centrándonos en el aprendizaje sin tener que complicarnos pensando en los requisitos para programar, la versión del sistema operativo, o del propio sistema operativo en si, ya que al utilizar Java, que es completamente multiplataforma, el ejecutable generado servirá igualmente para cualquier sistema.

Pero libGDX va incluso un poco mas allá, pues mediante cambios muy simples y reutilizando prácticamente todo el código original es posible importar el juego para Android, y es que el mercado de los videojuegos se dirige cada vez mas hacia los móviles, por lo que con esta librería, con programar una sola vez el código obtendremos un juego funcional en cualquier sistema operativo ademas de en cualquier dispositivo con Android.

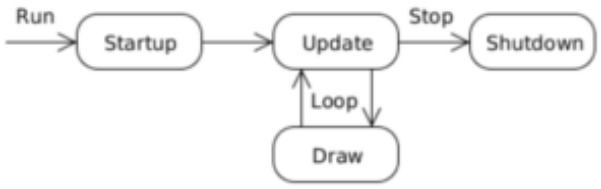


Figura 2.4: Estructura general de un Videojuego

Esto también es posible con librerías que usen C++, pero por lo general es una tarea mas complicada y puede desanimar si para empezar a programar debemos tener un sistema operativo o versión especifica.

De esta manera no hay excusas, puedes aprender a hacer un videojuego tengas el sistema operativo que tengas.

2.3. Estructura de un Videojuego

Antes de comenzar con aspectos concretos de la librería, vamos a dar un repaso a la estructura general de un videojuego. En cualquier videojuego podemos encontrar multitud de elementos funcionando al mismo tiempo e interactuando entre si. Por ejemplo, en un videojuego tipo First-Person-Shooter tenemos a nuestro propio personaje, las balas, los enemigos, todos los elementos del escenario, etc. En un juego de carreras hay muchos contrincantes contra los que competir, o incluso armas y objetos que pueden ser usados en beneficio propio o en contra del rival. Incluso en uno de lucha, que en principio los únicos elementos que influyen son los dos rivales, estos deben funcionar a la vez ya que el tiempo de reacción es decisivo para saber quien golpea primero.

Asi pues, necesitamos que todos los elementos de nuestro juego funcionen al mismo tiempo. Una opción lógica para ello es usar la concurrencia, es decir, múltiples hilos de ejecución y que cada elemento funcione por separado. Pero esta opción requiere de una potencia inmensa (sobre todo si tenemos mucho elementos de juego) además de que debe estar todo muy bien hilado para que no haya conflicto de prioridades.

Descartando la concurrencia, ya que ésta es muy poco usada en videojuegos, la opción mas básica y que se usa en la inmensa mayoría de los juegos es usar un bucle principal o Game Loop. De esta forma, ejecutaremos el juego secuencialmente. Los cambios de cada elemento se producirán uno detrás de otro, pero serán tan pequeños y se ejecutarán tan rápido (conocido como Frames por Segundo, normalmente 30 o 60) que dará la sensación de que todo se ejecuta al mismo tiempo.

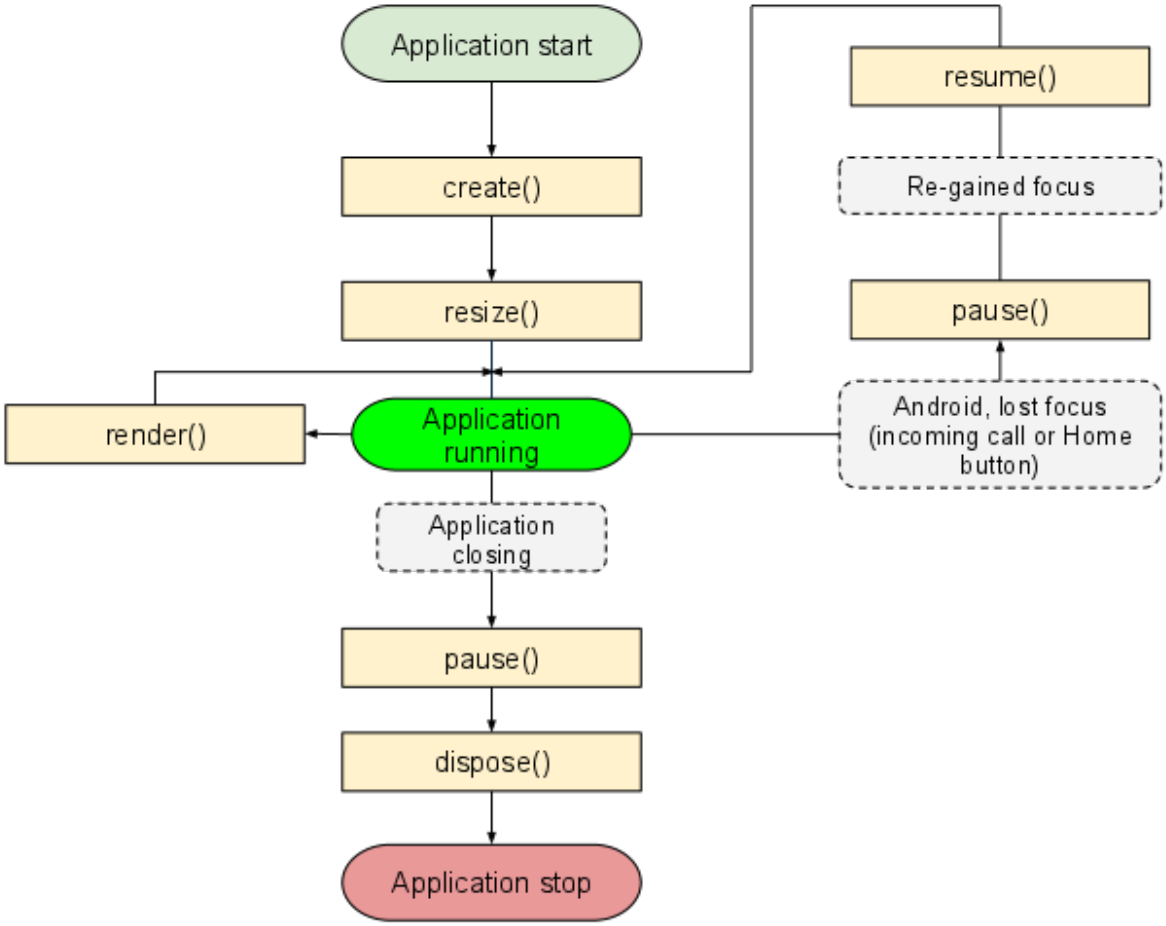


Figura 2.5: Estructura de LibGDX

2.4. Estructura de LibGDX

Tal y como hemos explicado, todos los videojuegos siguen el mismo esquema: Utilizar un Game Loop como bucle principal de juego. Pero cada librería tiene sus particularidades y utilizan distintas funciones para lograr este objetivo. Esta es la estructura que sigue LibGDX, la librería que hemos elegido para el desarrollo de nuestro videojuego: (Fig. 2.5)

Como vemos, la clase principal se divide en 6 funciones.

La función **create()** servirá para inicializar todas las variables y elementos del juego. Es lo primero que se ejecuta al iniciar el juego.

Posteriormente pasará a ejecutarse la función **render()**, que sera la función principal. Su funcionamiento consiste simplemente en ejecutarse todas las veces posible por segundo, por lo que en esta función es donde tendremos que hacer que se ejecuten todas las acciones del juego: dibujar los elementos por pantalla, procesar la entrada y salida, mover los personajes, detectar colisiones, etc.

La función **dispose()** es la ultima en ejecutarse, se encarga de liberar recursos y dejar la memoria limpia antes de cerrar el juego. Debemos tener siempre en cuenta esta función, ya que en LibGDX debemos controlar la liberación de los recursos que asignemos, a diferencia de lo que Java nos tiene acostumbrados (qué el mismo se encarga del recolector de basura). En esta función debemos liberar los recursos de todos los elementos externos que carguemos en el sistema, que principalmente serán las texturas, sonidos y demás multimedia.

Y por ultimo las funciones restantes que se ejecutan durante el juego: **resize()**, que sirve para recalcular el tamaño de los elementos cuando se modifica la pantalla, y **resume()** y **pause()**, que son funciones que se ejecutan en Android cuando salimos de la aplicación o se interrumpe la ejecución de la misma y volvemos a ella.

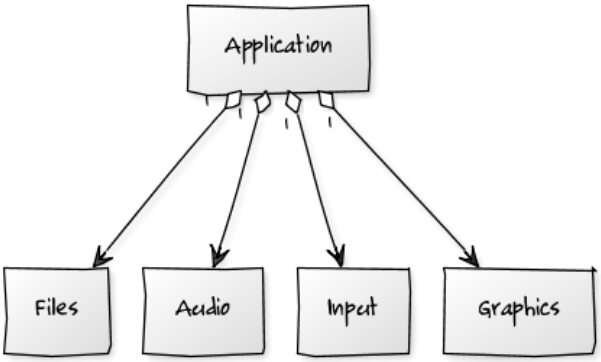


Figura 2.6: Módulos de LibGDX

LibGDX se divide en 4 módulos, agrupados dentro del Marco de aplicación:

- Marco de aplicación: maneja el bucle principal y el ciclo de vida
- Componente de gráficos: gestiona imágenes y objetos gráficos en pantalla
- Componente de audio: acceso a sonido y música de la aplicación
- Componente de E/S (files): lee y escribe en los ficheros de datos (imágenes, configuración, sonido...)
- Componente de Entrada (input): gestiona la entrada por teclado, pantalla táctil, acelerometro...

Capítulo 3

Planificación

En este capítulo describiremos la planificación temporal del proyecto, el período de duración, las fases en las que se ha compuesto, y el trabajo realizado en cada una de ellas.

Finalmente se adjunta un diagrama de Gantt (Pág. 14), realizado con GanttProject, un programa libre y multiplataforma muy fácil de usar, donde podremos ver la planificación de manera gráfica.

3.1. Fases

El proyecto puede dividirse en tres períodos principales:

3.1.1. Fase 1

El proyecto comenzó en diciembre de 2012, aunque realmente en aquel momento no era siquiera consciente de ello, ya que el objetivo era distinto.

Me encontraba preparando el taller de Introducción a la Programación de Videojuegos con Java y LibGDX (Fig. 3.1) que fue presentado el 21 de Marzo de 2013.

Para dicho taller el objetivo no era crear un videojuego, sino preparar una serie de ejercicios que permitieran a los alumnos aprender en apenas un par de horas la estructura y conceptos básicos para realizar un videojuego.

El taller me permitió realizar una pequeña investigación sobre la librería, adaptarme a ella y ver de que era capaz. A través de los ejercicios realizados para el taller se obtiene lo que ahora sería el primer nivel del juego, ya que en el mismo se explicó la mecánica general de un videojuego, el funcionamiento concreto de la librería LibGDX, el proceso de renderizado (mostrar imágenes por pantalla) y una breve introducción a la entrada de teclado y colisiones.

Figura 3.1: Cartel del taller 'Introducción a la Programación de Videojuegos con Java y LibGDX'

3.1.2. Fase 2

Después del taller tiene lugar un período de estancamiento, ya que debido a exámenes me vi obligado a centrar mis esfuerzos en los mismos. Ademas, en ese punto aún no pensaba en el videojuego como proyecto de fin de carrera. Fué al finalizar ese período (finales de Abril) cuando por fín empecé a pensar en el proyecto, y se me ocurrió aprovechar los materiales desarrollados, la investigación sobre la librería y la estructura de los videojuegos.

Por ello continué trabajando en el mismo, comenzando por completar el primer nivel utilizando lo ya existente.

De nuevo, la progresión se vio interrumpida, esta vez por una causa aún mas importante :El día 20 de Mayo empecé a trabajar en la empresa Alcatel-Lucent Technologies en Sevilla, lo que me hizo frenar el avance en el proyecto durante un par de semanas, tiempo que tuve que utilizar para mi adaptación al trabajo y a mi nueva vida.

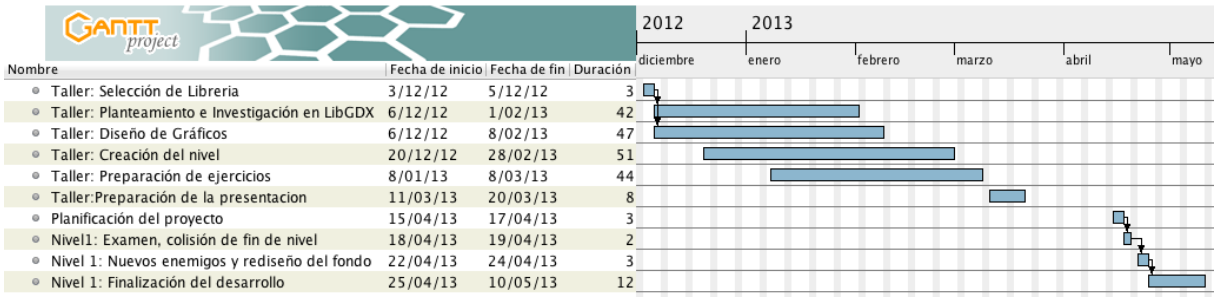
3.1.3. Fase 3

Y es finalizado este período de adaptación, a principios de Junio, cuando por fín retomo el proyecto, aprovechando las tardes entre semana y los fines de semana enteros para ir avanzando todo lo posible e ir cumpliendo la planificación establecida.

3.2. Diagrama de Gantt

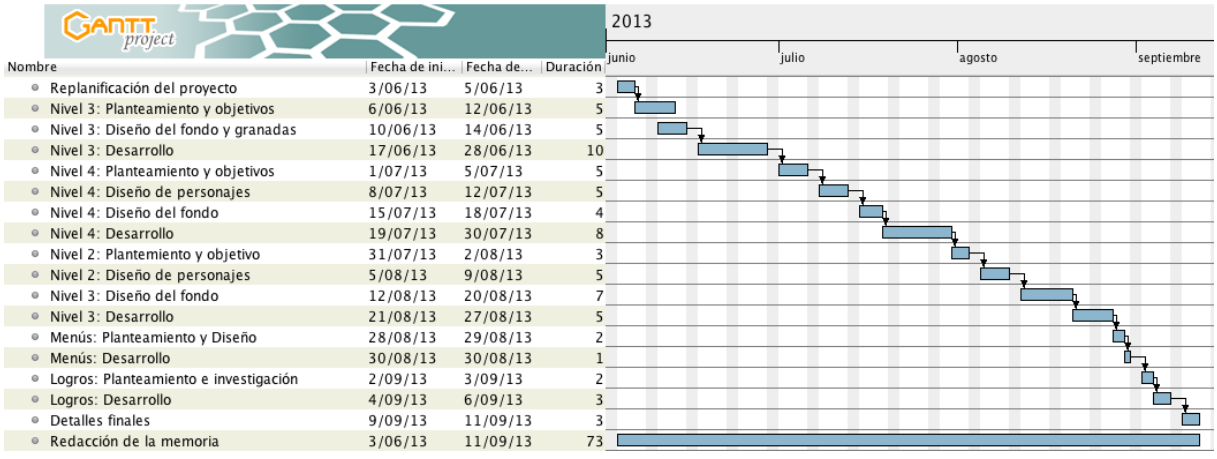
Nombre	Fecha de inicio	Fecha de fin	Duración
Taller: Selección de Librería	3/12/12	5/12/12	3
Taller: Investigación en LibGDX	6/12/12	1/02/13	42
Taller: Diseño de Gráficos	6/12/12	8/02/13	47
Taller: Creación del nivel	20/12/12	28/02/13	51
Taller: Preparación de ejercicios	8/01/13	8/03/13	44
Taller:Preparación de la presentacion	11/03/13	20/03/13	8
Planificación del proyecto	15/04/13	17/04/13	3
Nivel1: Examen, colisión de fin de nivel	18/04/13	19/04/13	2
Nivel 1: Rediseño de enemigos y fondo	22/04/13	24/04/13	3
Nivel 1: Finalización del desarrollo	25/04/13	10/05/13	12
Replanificación del proyecto	3/06/13	5/06/13	3
Nivel 3: Planteamiento y objetivos	6/06/13	12/06/13	5
Nivel 3: Diseño del fondo y granadas	10/06/13	14/06/13	5
Nivel 3: Desarrollo	17/06/13	28/06/13	10
Nivel 4: Planteamiento y objetivos	1/07/13	5/07/13	5
Nivel 4: Diseño de personajes	8/07/13	12/07/13	5
Nivel 4: Diseño del fondo	15/07/13	18/07/13	4
Nivel 4: Desarrollo	19/07/13	30/07/13	8
Nivel 2: Planteamiento y objetivo	31/07/13	2/08/13	3
Nivel 2: Diseño de personajes	5/08/13	9/08/13	5
Nivel 3: Diseño del fondo	12/08/13	20/08/13	7
Nivel 3: Desarrollo	21/08/13	27/08/13	5
Menús: Planteamiento y Diseño	28/08/13	29/08/13	2
Menús: Desarrollo	30/08/13	30/08/13	1
Logros: Planteamiento e investigación	2/09/13	3/09/13	2
Logros: Desarrollo	4/09/13	6/09/13	3
Detalles finales	9/09/13	11/09/13	3
Redacción de la memoria	3/06/13	11/09/13	73

Figura 3.2: Diagrama de Gantt



Nombre	Fecha de inicio	Fecha de fin	Duración
Taller: Selección de Libreria	3/12/12	5/12/12	3
Taller: Planteamiento e Investigación en LibGDX	6/12/12	1/02/13	42
Taller: Diseño de Gráficos	6/12/12	8/02/13	47
Taller: Creación del nivel	20/12/12	28/02/13	51
Taller: Preparación de ejercicios	8/01/13	8/03/13	44
Taller:Preparación de la presentacion	11/03/13	20/03/13	8
Planificación del proyecto	15/04/13	17/04/13	3
Nivel1: Examen, colisión de fin de nivel	18/04/13	19/04/13	2
Nivel 1: Nuevos enemigos y rediseño del fondo	22/04/13	24/04/13	3
Nivel 1: Finalización del desarrollo	25/04/13	10/05/13	12

Figura 3.3: Fases 1 y 2 de la planificación



Nombre	Fecha de ini...	Fecha de...	Duración
Replanificación del proyecto	3/06/13	5/06/13	3
Nivel 3: Planteamiento y objetivos	6/06/13	12/06/13	5
Nivel 3: Diseño del fondo y granadas	10/06/13	14/06/13	5
Nivel 3: Desarrollo	17/06/13	28/06/13	10
Nivel 4: Planteamiento y objetivos	1/07/13	5/07/13	5
Nivel 4: Diseño de personajes	8/07/13	12/07/13	5
Nivel 4: Diseño del fondo	15/07/13	18/07/13	4
Nivel 4: Desarrollo	19/07/13	30/07/13	8
Nivel 2: Plantemiento y objetivo	31/07/13	2/08/13	3
Nivel 2: Diseño de personajes	5/08/13	9/08/13	5
Nivel 3: Diseño del fondo	12/08/13	20/08/13	7
Nivel 3: Desarrollo	21/08/13	27/08/13	5
Menús: Planteamiento y Diseño	28/08/13	29/08/13	2
Menús: Desarrollo	30/08/13	30/08/13	1
Logros: Planteamiento e investigación	2/09/13	3/09/13	2
Logros: Desarrollo	4/09/13	6/09/13	3
Detalles finales	9/09/13	11/09/13	3
Redacción de la memoria	3/06/13	11/09/13	73

Figura 3.4: Fase 3 de la Planificación

Parte II

Desarrollo

Capítulo 4

Requisitos del Sistema

En este capitulo veremos las herramientas necesarias para empezar a realizar un videojuego.

4.1. Software

Antes de la instalación de la propia librería necesitaremos el Software Eclipse (también vale con algún otro entorno de desarrollo similar, pero la integración con Eclipse es realmente buena y es la que he utilizado para el proyecto). Podremos descargarlo de su pagina principal: Eclipse Bajamos el principal, Eclipse IDE for Java EE Developers.

Tras descargarlo obtendremos una carpeta llamada eclipse y haciendo click el archivo del mismo nombre ejecutaremos Eclipse, para mayor comodidad se puede crear un acceso directo al escritorio, o copiar dicha carpeta en la carpeta Aplicaciones si estamos en OS X. También deberemos tener Java instalado, si no es así, es muy sencillo realizarlo desde su pagina oficial: `http://www.java.com/es/download/`

4.2. Crear un nuevo proyecto

Para ello deberemos descargarnos la librería de LibGDX, esto lo podemos hacer desde su web: `https://code.google.com/p/libgdx/downloads/list`.

Descargaremos el mas reciente y al descomprimirlo obtendremos una carpeta llamada libgdx-nightly-latest en la que tendremos todos los archivos que componen la librería.

Para crear un nuevo proyecto ejecutamos el archivo gdx-setup-ui.jar (si no se abre al hacer doble click podemos acceder por terminal y ejecutar "java -jar gdx-setup-ui.jar") el cual nos abrirá la ventana de creación de proyectos.

Aquí rellenamos el nombre de nuestro proyecto y demás parámetros. El único requisito antes de crearlo es tener instalada la librería, para ello hacemos click aquí (Fig. 4.1) y cuando esté instalada nos aparecerá el nombre en verde.

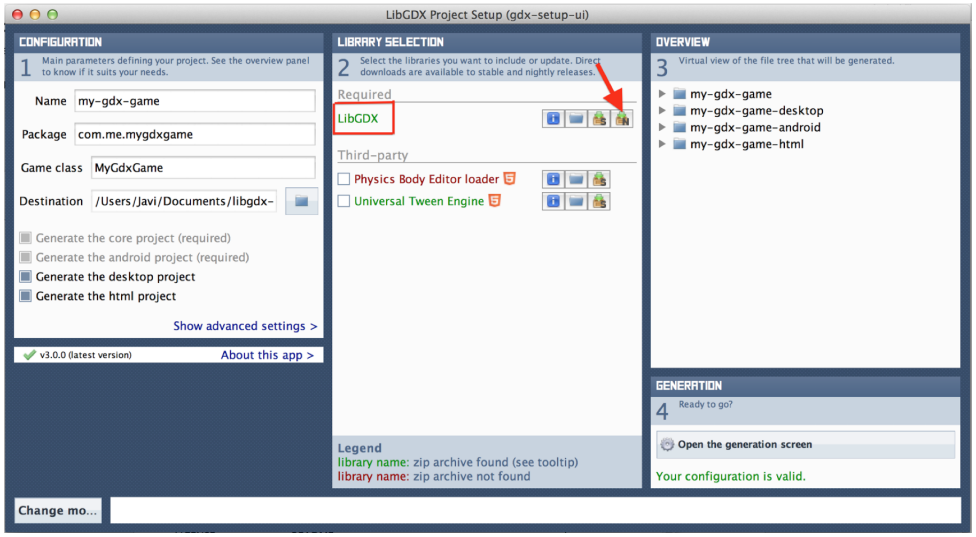
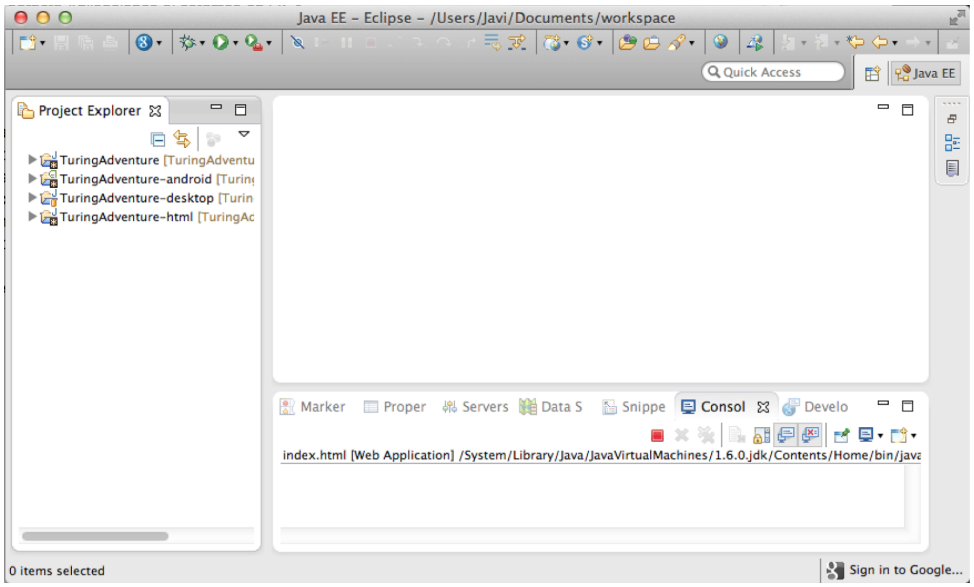


Figura 4.1: Instalación de LibGDX

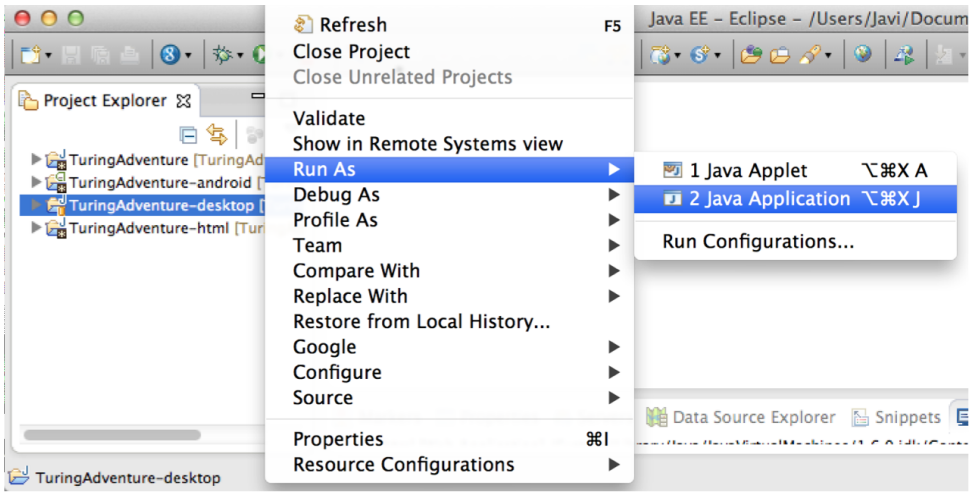
4.3. Importar un proyecto

Luego debemos hacer es importar nuestro proyecto en Eclipse. Esto servirá tanto para proyectos nuevos como para los ya existentes. Importar un proyecto es tan sencillo como entrar en Eclipse, hacer click en la pestaña **File/Import**, seleccionamos la carpeta **General** y dentro de ella **Existing Projects into Workspace**.

Pulsamos en Next y tenemos que seleccionar el proyecto a importar (seleccionamos la carpeta principal). Nos saldrán varios subproyectos para seleccionar, pertenecientes al objetivo de cada uno (desktop, Android, HTML...), seleccionamos los que nos interesan, que son el principal (siempre), el de desktop y el de android. Por último pulsamos Finish y obtendremos en Eclipse un entorno parecido a esto:



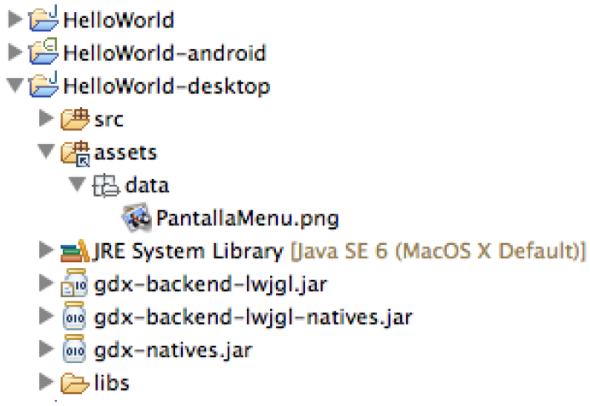
Para ejecutar nuestro proyecto solo tenemos que hacer click derecho en la carpeta desktop y seleccionamos **Run As/Java Application**



A la hora de programar nos centraremos en la carpeta general, todas están enlazadas, así que lo que modifiquemos en ella servirá para desktop, android y las demás que tengamos.

En concreto los archivos a modificar son los que se encuentran en la carpeta src, el resto son archivos de librerías y configuración interna de LibGDX:

Pero para insertar las imágenes y demás elementos externos lo haremos en la carpeta correspondiente: dentro de **assest/data**, aunque basta con hacerlo en una sola de ellas (la de desktop por ejemplo), todo se copia automáticamente en las demás.



Capítulo 5

Análisis del Sistema

En este capítulo mostraremos el modelo de los casos de uso con los que el usuario tendrá que interactuar, analizando cada uno de ellos.

5.1. Modelo de Casos de Uso

Primero veremos el modelo general de casos de uso: (Pág. 24)
Y ahora pasamos a analizar cada uno de ellos:

5.1.1. Caso de uso: Cambiar Usuario

Escenario principal:

- Se muestra el menú principal.
- El jugador selecciona el botón de Cambiar Usuario.
- El jugador Introduce su usuario (Ya existente) y pulsa el botón Aceptar.
- El sistema cambia el usuario al introducido por el usuario.
- Se muestra de nuevo el menú principal con el nombre del usuario en pantalla.

Escenario alternativo 1:

- Se muestra el menú principal.
- El jugador selecciona el botón de Cambiar Usuario.
- El jugador Introduce un nuevo usuario y pulsa el botón Cancelar.
- El sistema crea el nuevo usuario.
- Se muestra de nuevo el menú principal con el nombre del usuario en pantalla.

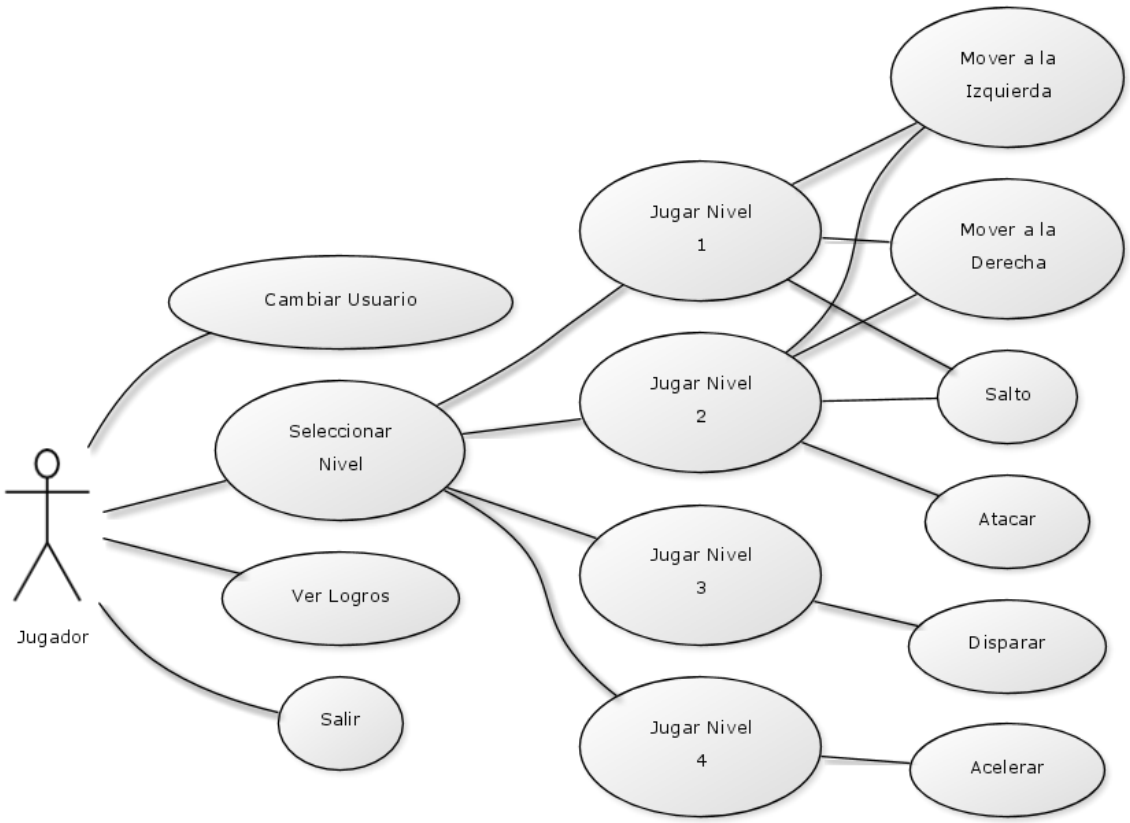


Figura 5.1: Modelo de Casos de Uso

Escenario alternativo 2:

- Se muestra el menú principal.
- El jugador selecciona el botón de Cambiar Usuario.
- El jugador Introduce su usuario (Ya existente) y pulsa el boton Cancelar.
- Se muestra de nuevo el menú principal sin cambiar el nombre del último usuario.

5.1.2. Caso de uso: Ver Logros

Escenario principal:

- Se muestra el menú principal.
- El jugador selecciona el botón de Logros.
- Se muestra la Pantalla de Logros, mostrando los logros obtenidos por el usuario actualmente logueado.
- El usuario selecciona el botón Volver.
- Se muestra el Menú principal.

5.1.3. Caso de uso: Salir

Escenario principal:

- Se muestra el menú principal.
- El jugador selecciona el botón de Salir.
- Se finaliza el juego y se cierra la aplicación.

5.1.4. Caso de uso: Seleccionar Nivel

Escenario principal:

- Se muestra el menú principal.
- El jugador selecciona el botón de Seleccionar Nivel.
- Se muestra la pantalla de Selección de Nivel.
- El jugador selecciona el primer nivel.
- El sistema muestra la pantalla de introducción del nivel.
- El usuario pulsa en la pantalla o cualquier tecla para seguir adelante.

- El sistema lanza la pantalla del nivel 1 y se inicia el juego.

Escenario alternativo 1:

- Se muestra el menú principal.
- El jugador selecciona el botón de Seleccionar Nivel.
- Se muestra la pantalla de Selección de Nivel.
- El jugador selecciona el segundo nivel.
- El sistema muestra la pantalla de introducción del nivel.
- El usuario pulsa en la pantalla o cualquier tecla para seguir adelante.
- El sistema lanza la pantalla del nivel 2 y se inicia el juego.

Escenario alternativo 2:

- Se muestra el menú principal.
- El jugador selecciona el botón de Seleccionar Nivel.
- Se muestra la pantalla de Selección de Nivel.
- El jugador selecciona el tercer nivel.
- El sistema muestra la pantalla de introducción del nivel.
- El usuario pulsa en la pantalla o cualquier tecla para seguir adelante.
- El sistema lanza la pantalla del nivel 3 y se inicia el juego.

Escenario alternativo 3:

- Se muestra el menú principal.
- El jugador selecciona el botón de Seleccionar Nivel.
- Se muestra la pantalla de Selección de Nivel.
- El jugador selecciona el cuarto nivel.
- El sistema muestra la pantalla de introducción del nivel.
- El usuario pulsa en la pantalla o cualquier tecla para seguir adelante.
- El sistema lanza la pantalla del nivel 4 y se inicia el juego.

Escenario alternativo 4:

- Se muestra el menú principal.
- El jugador selecciona el botón de Seleccionar Nivel.
- Se muestra la pantalla de Selección de Nivel.
- El jugador selecciona el botón Volver.
- Se muestra el Menú principal.

5.1.5. Caso de uso: Mover a la Izquierda

Escenario principal:

- Se muestra la pantalla de juego del Nivel 1 o 2.
- El jugador presiona la tecla 'A' o selecciona el botón de movimiento a la izquierda y no hay ningún obstáculo a la izquierda del personaje.
- El personaje se mueve a la izquierda.

Escenario Alternativo:

- Se muestra la pantalla de juego del Nivel 1 o 2.
- El jugador presiona la tecla 'A' o selecciona el botón de movimiento a la izquierda y hay un obstáculo a la izquierda del personaje.
- El personaje no se mueve.

5.1.6. Caso de uso: Mover a la Derecha

Escenario principal:

- Se muestra la pantalla de juego del Nivel 1 o 2.
- El jugador presiona la tecla 'D' o selecciona el botón de movimiento a la derecha y no hay ningún obstáculo a la derecha del personaje.
- El personaje se mueve a la derecha.

Escenario Alternativo:

- Se muestra la pantalla de juego del Nivel 1 o 2.
- El jugador presiona la tecla 'D' o selecciona el botón de movimiento a la derecha y hay un obstáculo a la derecha del personaje.
- El personaje no se mueve.

5.1.7. Caso de uso: Saltar

Escenario principal:

- Se muestra la pantalla de juego del Nivel 1 o 2.
- El jugador presiona la tecla 'W' o selecciona el botón de salto y el personaje no se encuentra saltando.
- El personaje salta.

Escenario Alternativo:

- Se muestra la pantalla de juego del Nivel 1 o 2.
- El jugador presiona la tecla 'W' o selecciona el botón de salto y el personaje ya esta en el aire.
- El personaje continua con su salto sin realizar uno nuevo.

5.1.8. Caso de uso: Atacar

Escenario principal:

- Se muestra la pantalla de juego del Nivel 2.
- El jugador presiona la tecla 'J' o selecciona el botón de ataque y el personaje tiene munición superior a 0.
- El personaje dispara.

Escenario Alternativo:

- Se muestra la pantalla de juego del Nivel 2.
- El jugador presiona la tecla 'J' o selecciona el botón de ataque y el personaje tiene 0 de munición.
- El personaje no dispara.

5.1.9. Caso de uso: Disparar

Escenario principal:

- Se muestra la pantalla de juego del Nivel 3.
- El jugador hace click o pulsa la pantalla sobre una granada.
- Se realiza el disparo y se destruye la granada.

Escenario Alternativo:

- Se muestra la pantalla de juego del Nivel 3.
- El jugador hace click o pulsa la pantalla en una zona vacía.
- Se realiza el disparo pero no ocurre nada.

5.1.10. Caso de uso: Acelerar

Escenario principal:

- Se muestra la pantalla de juego del Nivel 4.
- El jugador pulsa las teclas derecha e izquierda o los botones derecho e izquierdo en Android consecutivamente a la velocidad establecida.
- El personaje acelera, hasta llegar a la velocidad máxima.

Escenario Alternativo:

- Se muestra la pantalla de juego del Nivel 4.
- El jugador pulsa las teclas derecha e izquierda o los botones derecho e izquierdo en Android consecutivamente a una velocidad menor a la establecida.
- El personaje decelera hasta pararse.

5.2. Diagrama de clases

Ahora veremos el Diagrama de Clases del videojuego, donde tenemos las clases utilizadas en la implementación y la relación entre ellas. Para una mayor comprensión de las mismas y ver de donde procede cada una de forma individual es recomendable su visualización junto al diagrama de **Jerarquía de Clases** (Pág. 71) y a su posterior explicación junto de los atributos de cada una de ellas. Además tendremos una descripción mas detallada gracias a la generación de la documentación del código en **Javadoc** (Pág. 51).

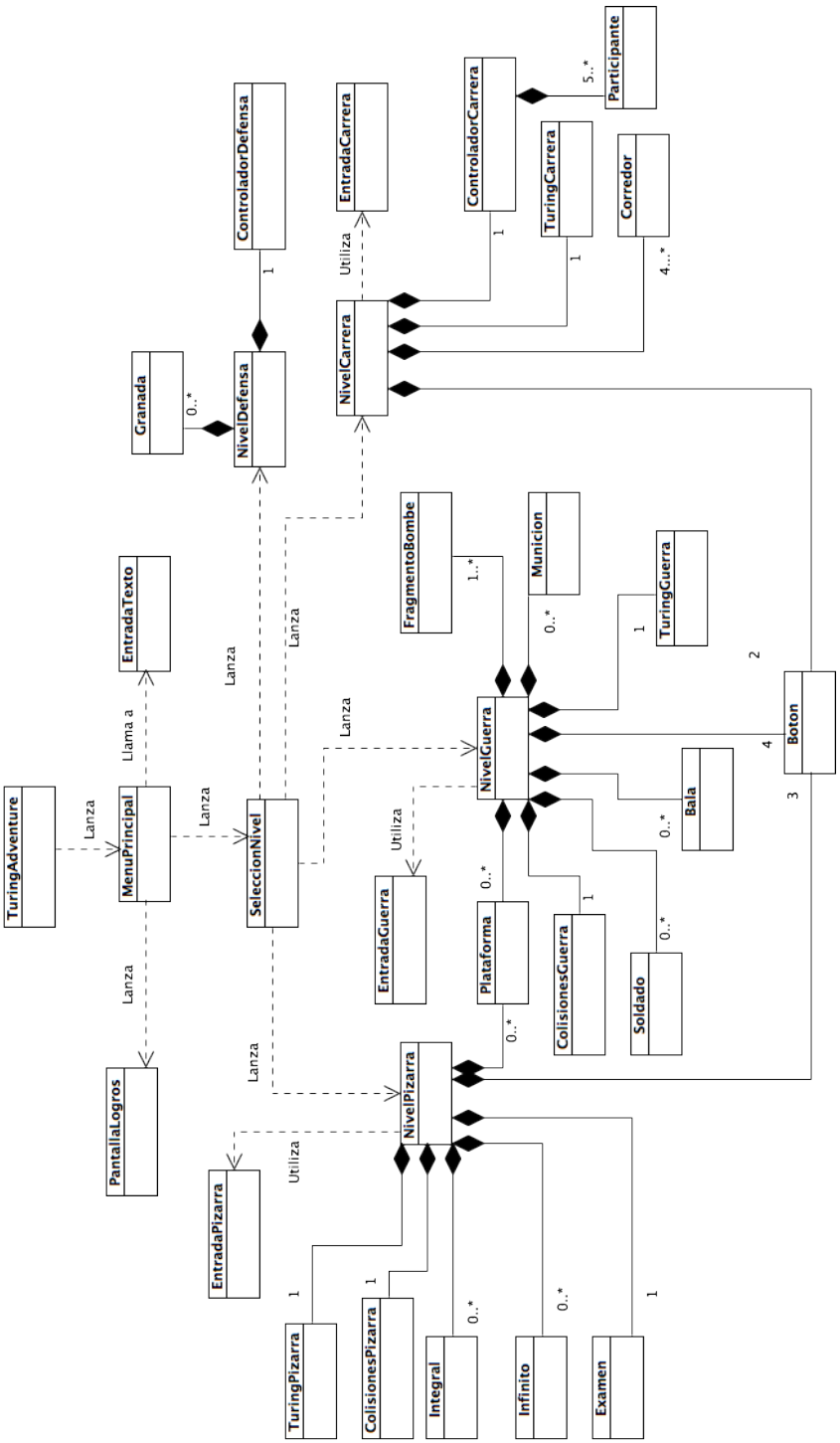


Figura 5.2: Diagrama de clases

Capítulo 6

Diseño del Sistema

En este capítulo recorreremos todas las posibles pantallas del juego y veremos todos los elementos gráficos que se han utilizado, así como una breve explicación de las herramientas y los métodos utilizados para desarrollarlos. Todas las imágenes que no han sido creadas desde cero han sido obtenidas de la web Openclipart

Para los sonidos y música en cambio no poseo las herramientas ni conocimientos necesarios para su desarrollo, por lo que han sido todos obtenidos de forma libre, en las webs FreeSound y Jamendo.

Todos los elementos multimedia empleados en la realización del proyecto estan incluidos dentro del proyecto de Eclipse, en concreto dentro del directorio TuringsAdventure/media: (`http://turing.forja.rediris.es/CodigoFuente.zip`).

6.1. Herramientas utilizadas

Ya que el objetivo del proyecto es el aprendizaje de la estructura y la lógica de juego, se ha procurado usar herramientas e imágenes libres para el diseño gráfico, para buscar una manera simple de realizar imágenes y animaciones y que cualquiera pueda tener acceso de una forma fácil y rápida.

6.1.1. Inkscape

Es una herramienta libre de creación de gráficos vectoriales. Se ha usado principalmente para la creación de los personajes y enemigos, a partir de formas geométricas simples. Es un software muy útil, y aunque al principio resultó extraño de utilizar, debido a mi falta de experiencia en diseño gráfico, me adapte muy rápido y fui capaz de crear personajes simples para el juego.

6.1.2. Gimp

Es también un herramienta libre, mas enfocada al retoque de imágenes que a la creación de éstas. Se ha usado para el montaje de los fondos combinando diversas imágenes, así como algunos de los botones y escalado de imágenes.

6.1.3. TexturePacker

Como su nombre indica es un empaquetador de texturas. No ha hecho falta mas que la versión de prueba, que aunque incluye menos características, ha servido para el objetivo requerido: agrupar los sprites de las animaciones creando una sola imagen con todos ellos. De esta forma para crear las animaciones no tenemos que cargar tantas imágenes, sino que en una sola almacenamos la animación completa.

6.1.4. FS Resizer

Figura 6.1: Pantalla del Menu Principal

Esta herramienta ha facilitado el trabajo de TexturePacker, ya que nos ha permitido reducir y ajustar el tamaño de múltiples imágenes al mismo tiempo, por lo que nos hemos ahorrado el trabajo de tener que ajustar el tamaño de todas la imágenes de una animación una a una.

6.2. Menús

Aquí vemos la pantalla del menú principal (Fig. 6.1). Es simplemente un fondo sobre el que se han dibujado los botones, a los que pulsado llegamos a las distintas pantallas del juego o salimos de el.

Al pulsar sobre el botón logros entramos en la pantalla de logros (Fig. 6.2), donde podremos ver los que hemos conseguido y los que no, junto con nuestra progresión en los mismos. Se ha utilizado el mismo método que para el menú principal, solo que añadimos la marca de logro desbloqueado delante de la imagen de cada logro si lo hemos obtenido.

Al pulsar sobre el botón Selección de Nivel, entramos en la correspondiente pantalla (Fig. 6.3), donde aplicando de nuevo el mismo método, podemos acceder a la pantalla de cada nivel.

Figura 6.2: Pantalla de visulización de Logros

Figura 6.3: Pantalla de selección de Nivel

Figura 6.4: Turing en el primer nivel

6.3. Nivel 1

6.3.1. Personajes

Aquí veremos los personajes incluidos en el primer nivel.
Turing (Fig. 6.4) es nuestro protagonista. Ha sido creado con Inkscape a partir de formas geométricas básicas. Gracias a este programa se puede dibujar al personaje por partes y así ir modificándolo para ir creando cada sprite de la animación. Finalmente la animación fue montada con FS Resizer y TexturePacker. Los enemigos son la Integral (Fig. 6.5) y el Infinito (Fig. 6.6), los cuales han sido creados de la misma forma.

Figura 6.5: Integral: Enemigo del primer nivel

Figura 6.6: Infinito: Enemigo del primer nivel

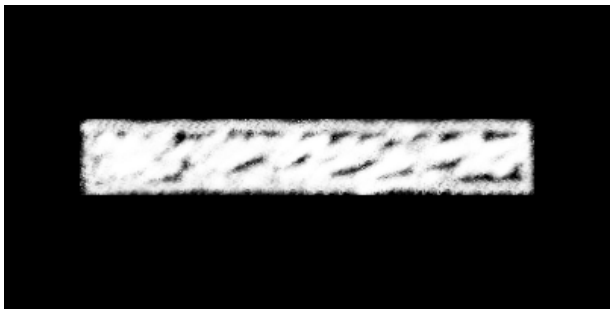


Figura 6.7: Plataforma flotante plana

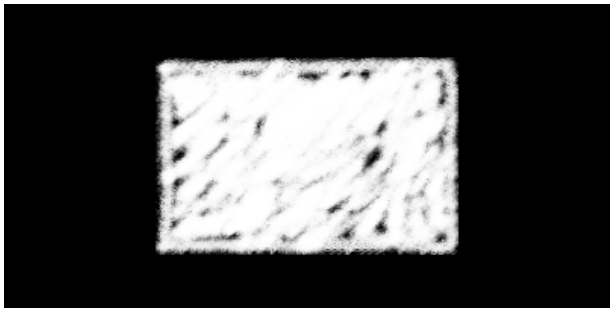


Figura 6.8: Plataforma cuadrada

6.3.2. Objetos

Para las plataformas (Fig. 6.7 y Fig. 6.8) se ha utilizado directamente Gimp, usando un pincel tipo tiza y darle ese efecto tan curioso.

Y por último el Examen (Fig. 6.9), que marcará el fin de nivel cuando lo recojamos, que también ha sido creado a partir de Gimp.

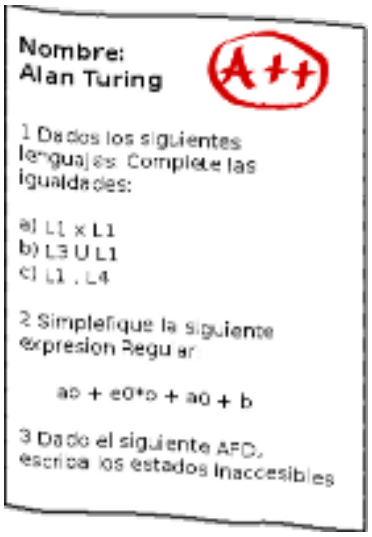


Figura 6.9: Examen: Objeto de fin del nivel 1

6.3.3. Fondo

Para el fondo (Fig. 6.10) se ha utilizado el mismo fondo que para los menús, solo que se ha dividido en varias partes obteniendo un gran fondo panorámico, y con Gimp se ha recortado en partes iguales, ya que LibGDX no permite cargar imágenes de tales dimensiones, por lo que hay que cargar el fondo por partes.

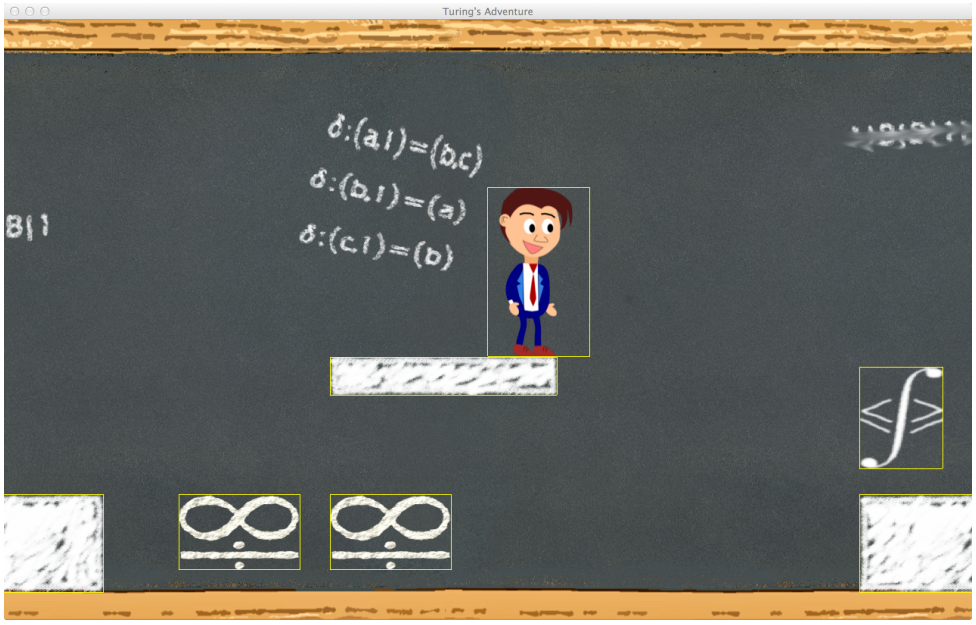


Figura 6.10: Pantalla del Nivel 1

6.4. Nivel 2

6.4.1. Personajes

Los personajes que usaremos aquí se han creado de la misma forma que los del primer nivel. Mediante Inkscape y agrupando la animación con FS Resizer y TexturePacker. Ademas se han usado algunas imágenes libres para el casco de los enemigos, la pistola, el fusil y las balas.

El personaje principal seguirá siendo Turing (Fig. 6.11), pero con un nuevo look ya que es un nivel ambientado en una escena de la segunda guerra mundial, con soldados alemanes, que serán nuestros enemigos (Fig. 6.12).

6.4.2. Objetos

Las plataformas en este nivel serán un tanque inutilizado (Fig. 6.13) creado a partir de un modelo distinto de tanque obtenido de Openclipart, una viga (Fig. 6.14), que hemos obtenido de la misma web, y un bunker (Fig. 6.15) que si se ha creado desde cero usando Inkscape.

Ademas, tendremos dos objetos adicionales, uno será el paquete de munición (Fig. 6.16), que podremos recoger para aumentar nuestra munición, y el otro objeto serán diversos fragmentos de la máquina Bombe (Fig. 6.17), que deberemos recoger todos para completar el nivel.

El paquete de munición se ha realizado mezclando imágenes obtenidas de la web, mientras que los fragmentos de la máquina Bombe sí que están hechos desde cero usando Inkscape.

Figura 6.11: Turing en el segundo nivel

Figura 6.12: Soldado Alemán: el enemigo del segundo nivel

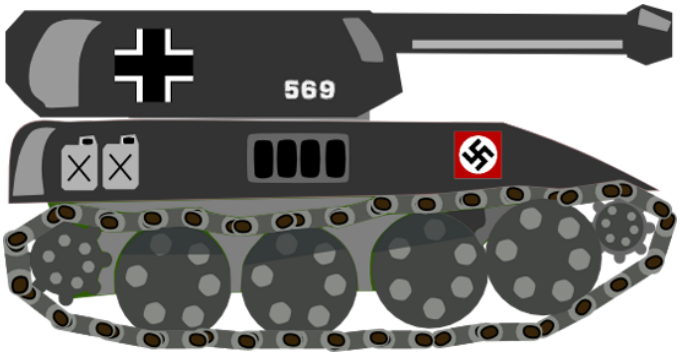


Figura 6.13: Tanque: Plataforma del segundo nivel

Figura 6.14: Viga: Plataforma flotante del segundo nivel

Figura 6.15: Bunker: Plataforma del segundo nivel

Figura 6.16: Paquete de munición disperso por el nivel 2

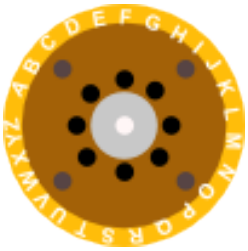


Figura 6.17: Fragmento de la Máquina Bombe

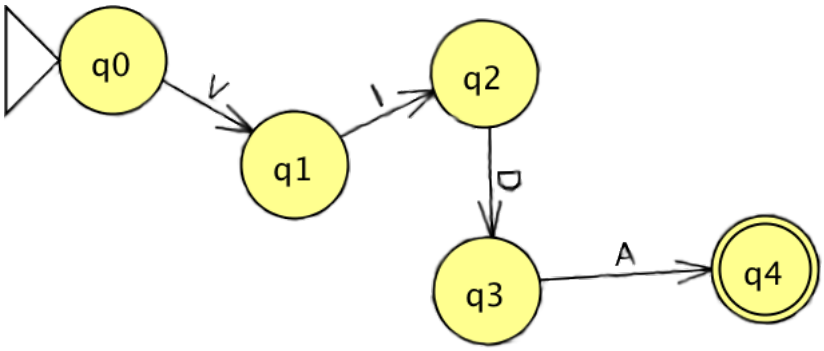


Figura 6.18: El visor de la vida de Turing

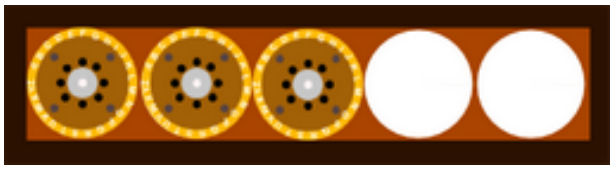


Figura 6.19: El visor de los fragmentos de la Máquina Bombe

Por último tendremos también una imagen para mostrar nuestra vida por pantalla. Se ha usado para ello una máquina de estados (Fig. 6.18) propia del software Jflap, como guiño a una de las asignaturas de la carrera donde más se mencionó el trabajo de Alan Turing. Ademas se ha creado dos contadores para visualizar los Fragmentos de la Máquina Bombe obtenidos (Fig. 6.19)(creado desde cero) y la munición restante (Obtenido de la web)(Fig. 6.20).

Figura 6.20: El visor de la munición de Turing

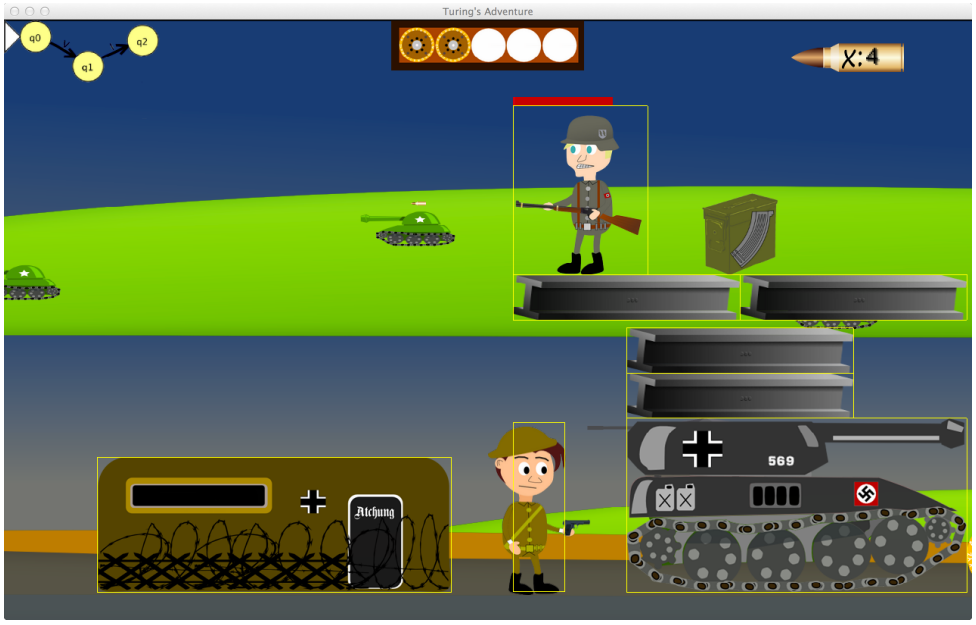


Figura 6.21: Pantalla del nivel 2

6.4.3. Fondo

Para este fondo se ha realizado un trabajo mas complejo que en el anterior. A partir de múltiples imágenes obtenidas de la web se han montado usando Gimp así como duplicando y retocando el fondo para hacerlo mas amplio (Fig. 6.21).

Figura 6.22: Granada: Elemento a destruir en el nivel 3

Figura 6.23: Pantalla del nivel 3

6.5. Nivel 3

Este nivel será el más sencillo en cuanto a gráficos y es el primero que explicaremos, por lo que hemos intentado incluir los menores objetos posibles, para centrarnos en explicar los conceptos básicos del juego.

En concreto, tendremos sólo dos elementos: Las granadas (Fig. 6.22), que hemos obtenido directamente en la web, y el fondo (Fig. 6.23), que hemos creado desde cero con Inkscape, excepto las zonas de hierba, que se han obtenido también de la web, y los soldados, que se han modificado a partir de imágenes de policías obtenidas del mismo sitio.

Figura 6.24: Turing en el cuarto nivel

6.6. Nivel 4

6.6.1. Personajes

Para este nivel se han creado desde cero, es decir, con Inkscape, todos los personajes, que son Turing (Fig. 6.24) y los tres posibles competidores en la carrera (Fig. 6.25, Fig. 6.26 y Fig. 6.27).
Por último hemos usado un visor para controlar en todo momento nuestra posición (Fig. 6.28).

6.6.2. Fondo

Para este fondo (Fig. 6.29) se han mezclado con Gimp y modificado con Inkscape múltiples imágenes obtenidas de la web, obteniendo así una calle ambientada en Londres, utilizando diversos objetos típicos como el autobús o la cabina telefónica.

Figura 6.25: Corredor 1: El primer contrincante de la carrera

Figura 6.26: Corredor 2: El segundo contrincante de la carrera

Figura 6.27: Corredor 3: El Tercer contrincante de la carrera

Figura 6.28: Visor del puesto para el nivel 3

Figura 6.29: Pantalla del nivel 4

Capítulo 7

Implementación

En un proyecto de desarrollo de software la implementación es la parte mas importante, ya que es necesario volcar todas las ideas y objetivos que tengamos a través del código fuente del programa.

Pero este proyecto tiene otra función, mas que la propia realización del videojuego, y es la creación de un manual (Página 71) que apoyado del resto del código fuente (`http://turing.forja.rediris.es/CodigoFuente.zip`) permita al programador novato conocer la estructura de un videojuego, en concreto a través de la librería LibGDX, y aprender a crearlo desde sus cimientos.

Aunque para este proyecto utilizaremos cuatro niveles utilizando distintos tipos de juego, la idea es dar a conocer los conceptos generales para que cuando el usuario termine no se limite a crear niveles del mismo tipo, sino que combinando todo lo aprendido tenga la base para crear cualquier otro tipo de juego.

7.1. Elementos a desarrollar

Los elementos que se incluirán en el Manual del Programador son:

- La inicialización del juego (A.4): Es la parte inicial. Desde donde se ejecuta la función principal, las funciones que utiliza y los parámetros de configuración.
- El manejo de pantallas del juego (A.5): LibGDX funciona a través de pantallas, veremos la clase principal de juego, que se encarga de lanzar y controlar todas las pantallas.
- Cómo mostrar imágenes por pantalla, cómo realizar la entrada de la pulsación del ratón o de la pantalla táctil y una breve introducción a las colisiones (A.6): Esto lo veremos en el nivel 'Descifrando el Enigma', donde los enemigos nos lanzarán granadas que debemos destruir haciendo click en ellas, y éstas ademas rebotarán con los bordes de la pantalla.

– Crear animaciones, entrada de teclado, botones en Android y lógica de adelantamiento en carrera (A.7) Lo describiremos en el nivel 'Maratón Olimpiadas Londres'. Realizaremos las animaciones tanto de Turing como del resto de corredores del nivel, ampliaremos el conocimiento de entrada de datos añadiendo el teclado y creando botones que sólo aparezcan si estamos en un dispositivo con Android y ademas aprenderemos a sincronizarlos para que la aceleración solo se realice si se pulsan consecutivamente a cierta velocidad. Añadiremos también 'inteligencia' a los corredores para que algunos de ellos no se dejen adelantar por Turing, siendo así fieles a la historia.

– Física del salto y ampliación de las colisiones (A.8): Aquí introduciremos la física, utilizando para ello el salto de Turing y mejoraremos los conocimientos obtenidos de colisiones, utilizando obstáculos para que al colisionar con ellos no podamos seguir avanzando, y enemigos que al chocar con ellos nos eliminen, pero si saltamos sobre ellos los derrotemos.

– Proyectiles, munición, vida y objetos coleccionables (A.9): En el nivel 'A por la Bombe' añadiremos nuevo contenido al anterior. Tendremos proyectiles, tanto propios (Para los cuales añadimos munición, que podemos recoger en el nivel) como enemigos (que son infinitos). Ademas añadiremos una nueva forma de finalizar el nivel: recogiendo un determinado número de objetos que tendremos repartidos por el nivel.

– Guardar datos, logros y usuarios (A.10): Esto lo veremos en el mismo Menú Principal, aprenderemos a guardar datos de forma externa al juego, en concreto en forma de logros, para al volver a iniciar el juego esa información no se haya perdido. Ademas individualizaremos estos logros haciéndolos propios de cada usuario que introduzcamos.

7.2. Control de versiones

Ante proyectos de un cierto tamaño es muy importante guardar siempre una copia de seguridad, pero cuando hablamos de desarrollo de software incluso teniendo copia de seguridad podemos encontrarnos con el problema de que lleguemos a un punto en el que nuestro programa no funcione y que nuestra copia de seguridad tenga incluido también el mismo error.

Esto se soluciona utilizando una herramienta de control de versiones, de manera que podamos volver a cualquier punto del desarrollo que hayamos guardado.

Para este proyecto hemos utilizado Subversion que es la herramienta de control de versiones que proporciona la forja en la que hemos registrado el proyecto (`https://forja.rediris.es/projects/turing/`). De manera que cada vez que lo guardamos, la versión actual del proyecto se guarda en el servidor de la forja, dándonos así mayor seguridad.

7.3. Documentación del código

Para generar la documentación del código se han seguido varios procedimientos. Por una parte se ha generado con Javadoc, que se integra dentro del propio proyecto de eclipse (directorio TuringAdventure/Javadoc, archivo index.html), que nos permitirá un acceso rápido a todas las clases del juego junto con una breve explicación de cada una de ellas, sus métodos, atributos y relaciones.

Por otra parte se han incluido comentarios dentro de cada sección del código que requiera una explicación mas precisa. Y por último para una guía completamente detallada, se ha desarrollado el documento ya descrito anteriormente como 'Manual del programador' adjunto en el Apéndice de esta memoria (71).

Capítulo 8

Pruebas del Sistema

En cualquier proyecto de desarrollo de software es imprescindible realizar una serie de pruebas para comprobar el estado del mismo.

Como todos sabemos es muy fácil cometer fallos al desarrollar cualquier programa o aplicación. Una simple línea o instrucción mal escrita o fuera de lugar genera errores en el código, que luego a la hora de ejecutar son fruto de errores de compilación o ejecución que a veces se hacen muy difíciles de resolver, aunque sean producto de un fallo muy simple, por lo que debemos ir haciendo pruebas constantemente para asegurarnos que nuestro programa esta limpio de errores.

8.1. Pruebas Unitarias

Se han ido realizando pruebas unitarias constantemente durante el desarrollo de todo el proyecto. Esto es, cada vez que se creaba un nuevo elemento se probaba por separado para garantizar el funcionamiento del mismo antes de integrarlo con el resto del juego.

Esto incluye tanto elementos concretos como nuevos personajes u objetos (cuya prueba se realizó generándolos en una pantalla vacía y comprobando su movimiento) como a secciones o lógica de juego, como por ejemplo las colisiones, que para probarlas se generaron una serie de obstáculos en una pantalla vacía junto con el personaje, y se probó una y otra vez hasta que las colisiones funcionaron correctamente.

Estas son las pruebas que se han realizado para el proyecto:

8.2. Pruebas de Integración

Para estas pruebas se comprobaron los módulos ya completos, es decir, los niveles. Una vez realizadas las pruebas unitarias y agregados todos los elementos a cada nivel se probaron individualmente, comprobando que cada uno funcionaba como es debido:

- Nivel 1: La principal comprobación en este nivel fue que la física funcionaba correctamente, desde diversos puntos aleatorios del nivel, ya fuera desde el suelo, plataformas, que no se pudiera saltar de nuevo en el aire, etc. Ademas de las colisiones, se probó con el máximo número de enemigos posibles, viendo que cuando colisionamos por encima los derrotamos y en cualquier otro caso éramos nosotros los abatidos. Además también se tuvieron en cuenta las colisiones con las plataformas, que no se pudiera avanzar al chocar con las mismas, que interrumpiesen el salto al chocar por arriba con ellas, y que permitiesen al personaje posarse y andar sobre ellas sin bloquearse.

- Nivel 2: Las pruebas para este nivel fueron muy parecidas al anterior, solo que añadimos las colisiones con balas, que tanto las balas de los enemigos nos reducían vida, y las nuestras la vida de los enemigos, pero que al chocar con ellos la derrota era automática.
  Se comprobó la recogida de objetos, tanto de munición como de fragmentos de la máquina Bombe y que al recoger una cantidad determinada de estos últimos se completaba el nivel.

- Nivel 3: Las pruebas para este nivel fueron mas sencillas, ya que por algo es el más básico. Simplemente hubo que comprobar las colisiones básicas con el borde de la pantalla, que las granadas rebotaban, la generación automática de las mismas, y que al dispararles, es decir, pinchar sobre ellas, eran destruidas.
  También se comprobó que al llegar una de estas al borde inferior de la pantalla concluía el juego siendo derrotados, o por el contrario al superar la cuenta atrás ganábamos la partida.

- Nivel 4: Este nivel fue algo mas complejo de probar, ya que había que tener en cuenta la posición de los corredores, comprobar que se adelantaban y actualizaban su puesto de forma correcta, y que cuando los designados como ganadores eran adelantados por Turing, estos aumentaban su velocidad. Por supuesto también hubo que comprobar la aceleración o deceleración según la velocidad con la que pulsáramos consecutivamente las teclas derecha e izquierda, y que al llegar al fin del nivel en el 5º puesto completábamos el nivel.

- Logros y usuarios: Ambos elementos van de la mano, hubo que comprobar que se guardaban bien los elementos de forma externa al juego, para que se guardaran al volver a entrar, y que se diferenciaran claramente los usuarios que íbamos introduciendo.

8.3. Pruebas Funcionales

Para estas pruebas debemos comprobar que el sistema se ajusta al modelo de casos de uso establecido, por lo que su realización es muy sencilla. Una vez implementados los menús, solo hay que comprobar que estos funcionan correctamente y que cada botón nos lleva a la pantalla correspondiente, a su vez que los botones de selección de nivel nos lleven a la correspondiente pantalla de juego, ya probada previamente. De esta forma comprobamos que la interacción del usuario es completa y cumple con los objetivos del modelo de casos de uso.

8.4. Pruebas No Funcionales

En esta prueba se ha intentado medir la eficiencia del juego. Para ello simplemente se ha probado en distintos sistemas operativos y dispositivos, tanto ordenadores con diversos sistemas operativos y versiones de los mismos (Windows XP y Windows 7, Ubuntu 10.4 y 12.04, Linux Mint y Mac OS Mountain Lion) y en distintos dispositivos Android (Nexus 7, Nexus One y Samsung Galaxy Mini, todos con Jelly Bean).

8.5. Pruebas de Aceptación

Por último se han realizado las pruebas de aceptación, las cuales han consistido en distribuir el juego entre varias personas para que todos lo prueben e intenten encontrar posibles fallos o carencias del juego. El juego ha sido testeado tanto por compañeros de Ingeniería Informática, por lo que han tenido una visión mas interna de los posibles fallos que podían encontrar, como por usuarios sin conocimientos de programación que simplemente probaron el juego de todas las formas que les fue posible.

Parte III

Epílogo

Capítulo 9

Manual de usuario

En este capítulo veremos el manual de juego, de cara al usuario, para ver las opciones que nos ofrece cada nivel, lo que podremos hacer en cada uno y como hacerlo.

9.1. Instalación

9.1.1. PC

Esta sección es válida para cualquier ordenador, independientemente del sistema operativo. Es una de las ventajas de que el videojuego este desarrollado en Java, que nos permite unificar la instalación haciéndola idéntica para todos los sistemas.

La instalación en PC es tan sencilla como descargar el ejecutable y hacer doble click sobre él. El único requisito es tener instalada en nuestro sistema la última versión de Java (o Java 6 en el caso de Mac), lo cual podemos hacer desde su página oficial . El ejecutable lo podemos bajar desde la web del proyecto: `http://turing.forja.rediris.es/TuringAdventure.jar`.

Se han dado casos en los que este sistema ha fallado en Mac OS X, si en este se ha llegado a instalar una versión posterior a Java 6, debido a que la versión estable para Mac es la 6 y por lo tanto la que se asocia al ejecutable, pero esto no debe ocurrir a no ser que se fuerce a la instalación de una versión posterior.

En tal caso, habría que volver a dejar la versión 6 de Java, o esperar a alguna actualización por parte del própio sistema operativo que funcione bien con la nueva versión.

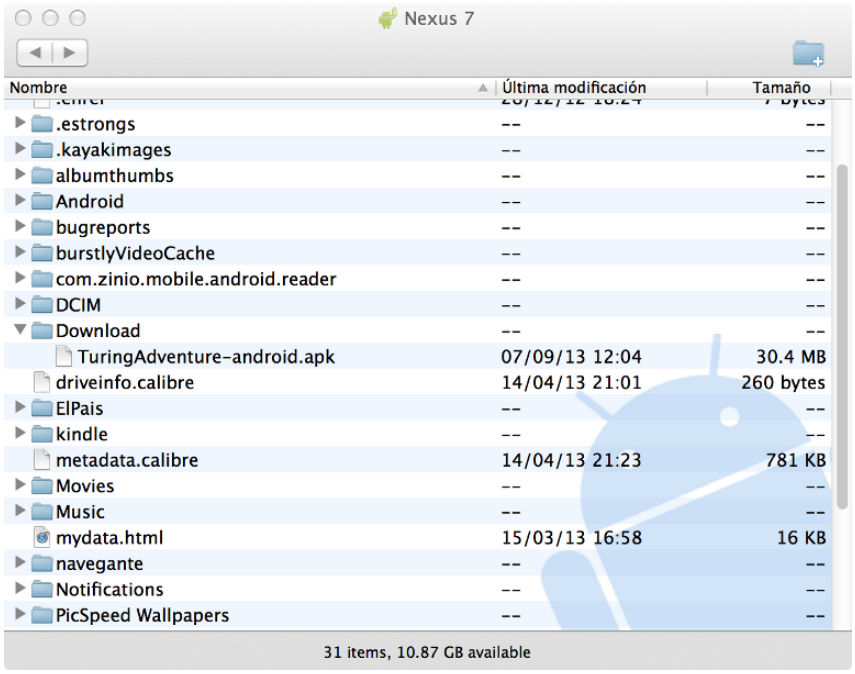


Figura 9.1: Copiamos el instalador en nuestro dispositivo desde el PC

9.1.2. Android

Para la instalación en Android simplemente debemos descargar el instalador de la web del proyecto, introducirlo en nuestro dispositivo e instalarlo.
Para ello debemos introducir el archivo de instalación 'TuringAdventure-android.apk' (que podemos descargar de la web `http://turing.forja.rediris.es/TuringAdventure-android.apk`) en nuestro dispositivo. Simplemente lo conectamos al ordenador por USB y trasladamos el archivo a la carpeta 'Downloads"del dispositivo (Fig. Fig. 9.1).

En el caso de Mac OS no es posible acceder directamente al directorio raíz del dispositivo, para hacerlo podemos descargar algún software como File Transfer. Al instalarlo podremos acceder al directorio de nuestro dispositivo Android nada mas conectarlo al Mac (Fig. Fig. 9.2).
Una vez introducido el archivo debemos instalarlo desde Android. Para entrar al explorador de archivos debemos descargar alguna aplicación de manejo de ficheros, como por ejemplo ES Explorador de Archivos. Una vez instalada solo la ejecutamos, y entramos en la carpeta en la que tengamos nuestro instalador, lo ejecutamos, y aceptamos la instalación (Fig. 9.3). Y con esto tendremos nuestro juego correctamente instalado en nuestro dispositivo.

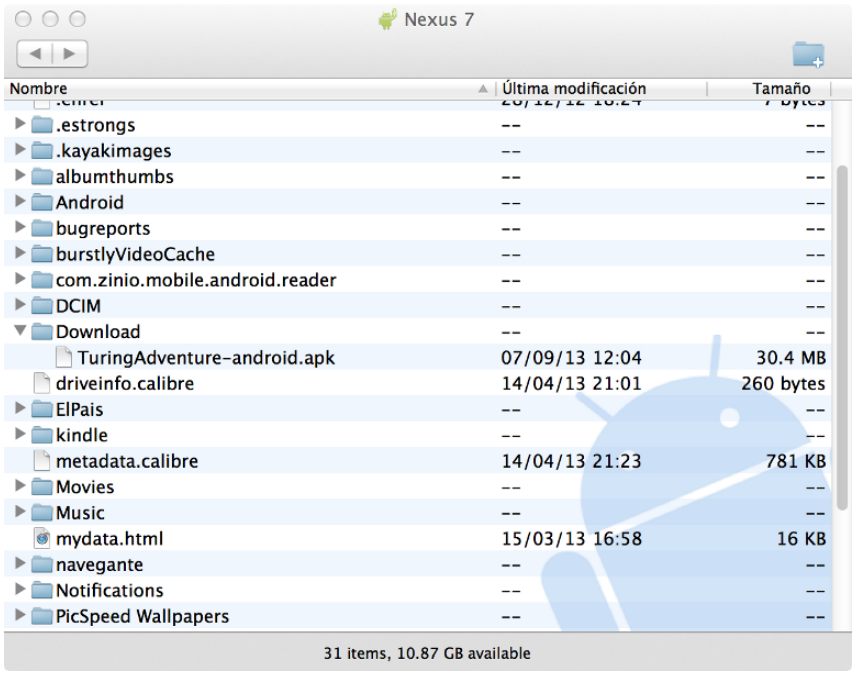


Figura 9.2: Copiamos el instalador en nuestro dispositivo desde el Mac

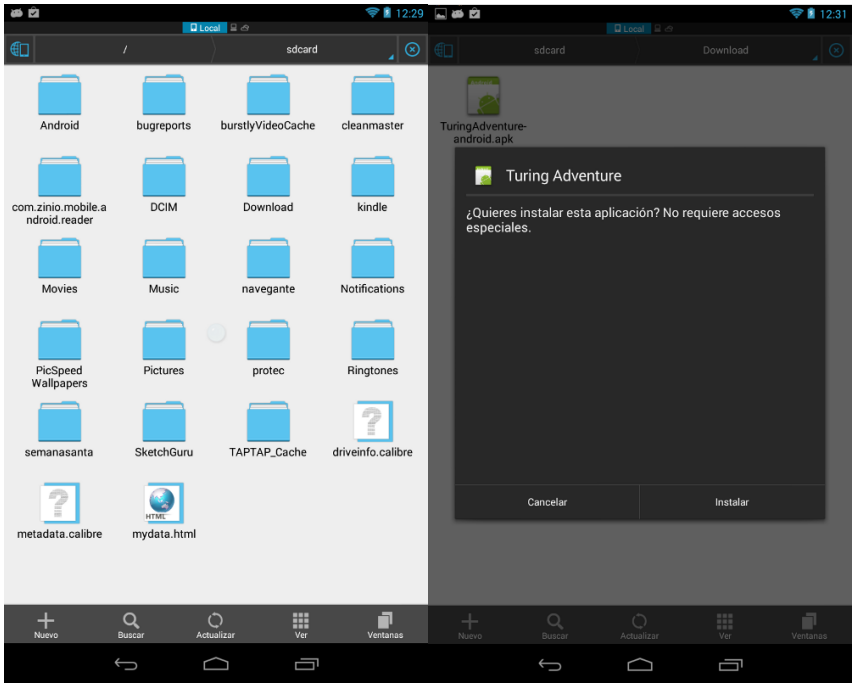


Figura 9.3: Instalación del juego en Android

9.2. Juego

9.2.1. Menus

En el menú principal (Pág. 33 Fig. 6.1) tenemos tres opciones, ademas de la de salir del juego.

La primera es la de cambiar o crear nuestro usuario. Haciendo click en el botón correspondiente obtendremos una ventana de entrada de texto, donde podremos introducir nuestro usuario. En el caso de que sea la primera, vez éste se creará.

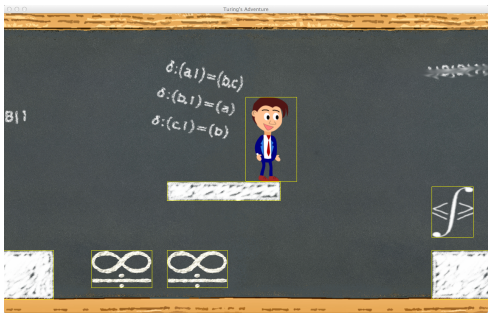
La segunda es el acceso al menú de selección de nivel (Pág. 34 Fig. 6.3), donde haciendo click en el botón correspondiente entraremos a cada nivel del juego, o volveremos a la pantalla del menú principal.

La última es el acceso a la pantalla de visualización de logros (Pág. 34 Fig. 6.2), en la que veremos los progresos en los logros del usuario introducido.

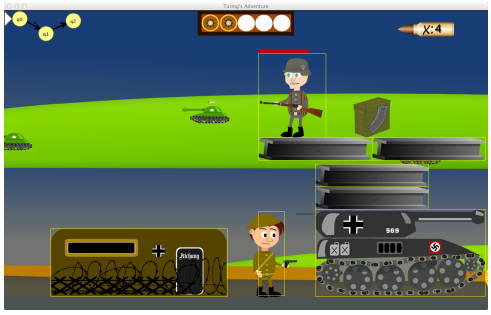
9.2.2. Controles y objetivos

Como ya hemos visto, los controles y objetivos para cada uno de los cuatro niveles son distintos, así que pasaremos a describirlos nuevamente:

– **Nivel 1 - Avanzando en Cambridge** (Pág. 39 Fig. 6.10): El objetivo de este nivel es simplemente llegar al final del nivel y obtener el examen aprobado, simulando así la finalización de los estudios de Alan Turing en Cambridge. En nuestro camino nos encontraremos enemigos que debemos esquivar o eliminar saltando encima de ellos. Los controles de este nivel se reducen a tres posibles botones. Movimiento a la izquierda y derecha con las teclas 'A' y 'D' respectivamente y Salto con la tecla 'W', o con los botones diseñados para ello en Android.



– **Nivel 2 - WW2: A por la Bombe** (Pág. 43 Fig. 6.21): Aunque el nivel es muy similar al anterior, los objetivos y controles varían. En este nivel no basta con llegar al final, sino que deberemos recoger los cinco fragmentos de la máquina Bombe repartidos por el escenario.
Da igual que lo recorramos entero o que hayamos derrotado a todos los enemigos, si no los tenemos todos no completaremos el nivel. Los controles son idénticos al nivel anterior solo que añadimos una nueva opción: Los disparos. Para ello podemos pulsar la tecla 'J' en el ordenador o el botón correspondiente en Android. En este nivel debemos alejarnos a toda costa de los enemigos ya que la única forma de derrotarlos es disparandoles, si chocamos con ellos no solo nos haran daño, sino que nos derrotaran sin importar las vidas que nos queden.



– **Nivel 3: WW2: Descifrando el Enigma** (Pág. 44 Fig. 6.23): El objetivo de este nivel es bien sencillo, resistir la cuenta atrás sin que las granadas lleguen a nosotros y estallen. Para ello simplemente tenemos que dispararles, por lo que los controles son tan simples como hacer click o pulsar sobre las granadas antes de que están lleguen al fondo de la pantalla.

– **Nivel 4: Maratón Olimpiadas Londres** (Pág. 48 Fig. 6.29): En este último nivel el objetivo es diferente al esperado, no debemos ganar la carrera, sino llegar en quinta posición. Para ello debemos acelerar pulsando consecutivamente las teclas derecha e izquierda del teclado (o los botones derecho e izquierdo en Android). Debemos hacerlo lo mas rápido posible, ya que si reducimos la velocidad de pulsación o nos equivocamos y no lo hacemos en orden la velocidad de Turing disminuirá.

Capítulo 10

Conclusiones

10.1. Experiencia personal

10.1.1. Lo mejor

Por supuesto lo que mas me ha gustado es el simple hecho de poder investigar sobre la realización básica de un videojuego. Es de las pocas cosas que me llamaban la atención para realizar el proyecto de fin de carrera, pero había descartado la idea ya que tenía entendido que era algo muy típico y había que tener una idea muy original para que te dieran el visto bueno, y por desgracia la imaginación no es mi punto fuerte.

Pero cual fue mi alegría al ofrecerme mi tutor la posibilidad de utilizar el taller que realicé sobre Introducción al Desarrollo de videojuegos con Java y LibGDX como base del proyecto, teniendo así una temática, enfocándolo al aprendizaje y realizando varios niveles en lugar de un único tipo de juego, dándole así ese pequeño toque de originalidad.

También ha sido una buena experiencia el trabajo con gráficos, nunca antes había tenido experiencia con el diseño de gráficos, menos aún de animaciones, ya que es algo que no se me da especialmente bien.

La idea inicial era utilizar sólo material libre para los gráficos del juego o pedir a algún compañero que los realizara por mi. Pero finalmente me atreví a dar el paso e intentar al menos realizar diseños básicos. El resultado creo que ha sido satisfactorio, aunque no me he convertido en un artista ni mucho menos, pero he conseguido realizar unos modelos muy simples, aunque medianamente decentes, que distan mucho de la idea que tenia de mis capacidades al inicio del proyecto, ya que pensaba que no sería capaz ni de tan siquiera algo así.

10.1.2. Lo peor

Lo peor del proyecto, no haberle podido dedicar todo el tiempo que quisiera, ha sido provocado por una, por otra parte, afortunada circunstancia, ya que como se comentó en la planificación, desde el mes de mayo estoy trabajando en Alcatel-Lucent Technologies en Sevilla, lo que provocó un importante parón en la planificación del proyecto, debido tanto a la incorporación al mundo laboral como al hecho del traslado de mi domicilio a Sevilla y mi emancipación, por lo que el tiempo dedicado al proyecto se ha visto reducido a los ratos libres de las tardes entre semana y aprovechar los fines de semana completos.
Dicha circunstancia de falta de tiempo se podría haber solucionado simplemente alargando aun mas la entrega del proyecto, pero sin embargo, el hecho de no contar aún con el título de Ingeniero provocó la presión de la empresa para dejarlo finalizado lo antes posible.

10.2. Posibles ampliaciones

Aunque estoy contento con el resultado del proyecto, la presión por su finalización ha hecho que me queden algunas cosas que me gustaría ampliar o investigar, como por ejemplo:

- Mejora de la física: En los dos últimos niveles explicados hemos visto una introducción a la física, reflejada en el salto de Turing, que controlamos mediante constantes y variables de estado. Una mejora directa de esto es la inclusión de gravedad, añadiendo aceleración al personaje para que la velocidad del salto no sea constante. O incluso un añadido completo sería la investigación del motor Tween Engine, proporcionado por la propia librería LibGDX.

- Guardado de datos en red: Gracias a las Preferencias hemos visto como guardar información de manera externa al juego, guardando asi usuarios y logros personales, pero al cambiar de ordenador o dispositivo esta información se pierde.
  Sería una interesante investigación del almacenamiento de estos datos en red mediante un servidor PHP o alguna herramienta similar, de manera que mantengamos nuestro usuario y logros sin importar en qué dispositivo juguemos, siempre que tengamos conexión a Internet.

- Nuevos niveles: Experimentar con algún nuevo nivel, por ejemplo de tipo aventuras de vista aérea, del estilo de los primeros Zelda (Fig. 10.1), o tipo shooter del mismo estilo como Minigore. (Fig. 10.2)

- Especialización en móviles: Realizar una investigación mas profunda de las capacidades de Android y añadir funciones táctiles de deslizamiento, multitáctiles o incluso incluyendo la brújula o el giroscopio, como por ejemplo en los niveles de desplazamiento lateral mover al personaje inclinando el móvil, saltar deslizando un dedo hacia arriba y disparando si pellizcamos la pantalla con varios dedos.

Figura 10.1: The Legend of Zelda

Figura 10.2: Minigore

Parte IV

Apéndice: Manual del Programador

Apéndice A

Manual del Programador

A.1. Introducción

Durante todo este documento se hará referencia al código del juego (http://turing.forja.rediris.es/CodigoFuente.zip), indicando la clase o incluso las lineas en las que se encuentra dicho fragmento de código, aunque para cosas concretas se mostrara dicho fragmento en el texto para explicarlo directamente.

Al ser el propósito de éste proyecto la realización de un manual básico para que programadores novatos puedan iniciarse en el desarrollo de videojuegos hacemos mas hincapié en una explicación mas detallada.

También es necesario saber que el orden en el que se explicará el juego no es el mismo que el que podemos observar en el menú. En el menú están situados por orden cronológico, acorde a la historia de Alan Turing, pero aquí los iremos viendo por nivel de complejidad, añadiendo cada vez mas elementos o formas distintas de programar.

Cuando veamos que hay algo en el código que no está explicado en su correspondiente sección, es porque se explicará más adelante, como sucede con los logros, que van guardándose y comprobándose durante todo el juego, pero se explicaran al final de la memoria junto a las pantallas de menú, selección de nivel, usuario y logros.

A.2. Jerarquía de clases

Antes de nada veremos cuales son las clases que usaremos durante todo el desarrollo. No hace falta que las entendamos todas ahora mismo, pero nos referiremos a este apartado cada vez que nos centremos en alguna especifica.

Algo a destacar es la modularización que vamos a seguir para este juego. Como cualquier programa informático, un videojuego debe estar bien modularizado, estructurado en clases que se

centren cada una en un apartado del juego, ya que si nos dedicamos a tratar todo el juego en la función principal render() podemos convertirlo en un completo caos.

Como mínimo, deberemos tener una clase que se encargue de manejar la Entrada/Salida y otra que se encargue de la lógica de juego (Con física y colisiones si es preciso), ademas de los elementos individuales como objetos, personajes, etc.

En cualquier caso normal usaríamos una única clase para manejar la E/S y otra para la lógica de juego, bien estructuradas y que abarquen todos los aspectos del juego. Pero en este caso nos encontramos ante una situación especial: cada nivel va a ser independiente, centrándose en un aspecto de juego distinto o ampliando lo ya visto en un nivel anterior, por lo que se ha optado por crear un manejador de E/S y un controlador específico para cada nivel.

Al igual que con nuestro personaje principal, Turing, tendremos uno específico para cada nivel, cada uno con las características y atributos necesarios en ese momento. De esta forma podemos centrarnos en explicar solo lo necesario para cada nivel y no abrumarnos viendo cosas que no sean necesarias en el nivel actual.

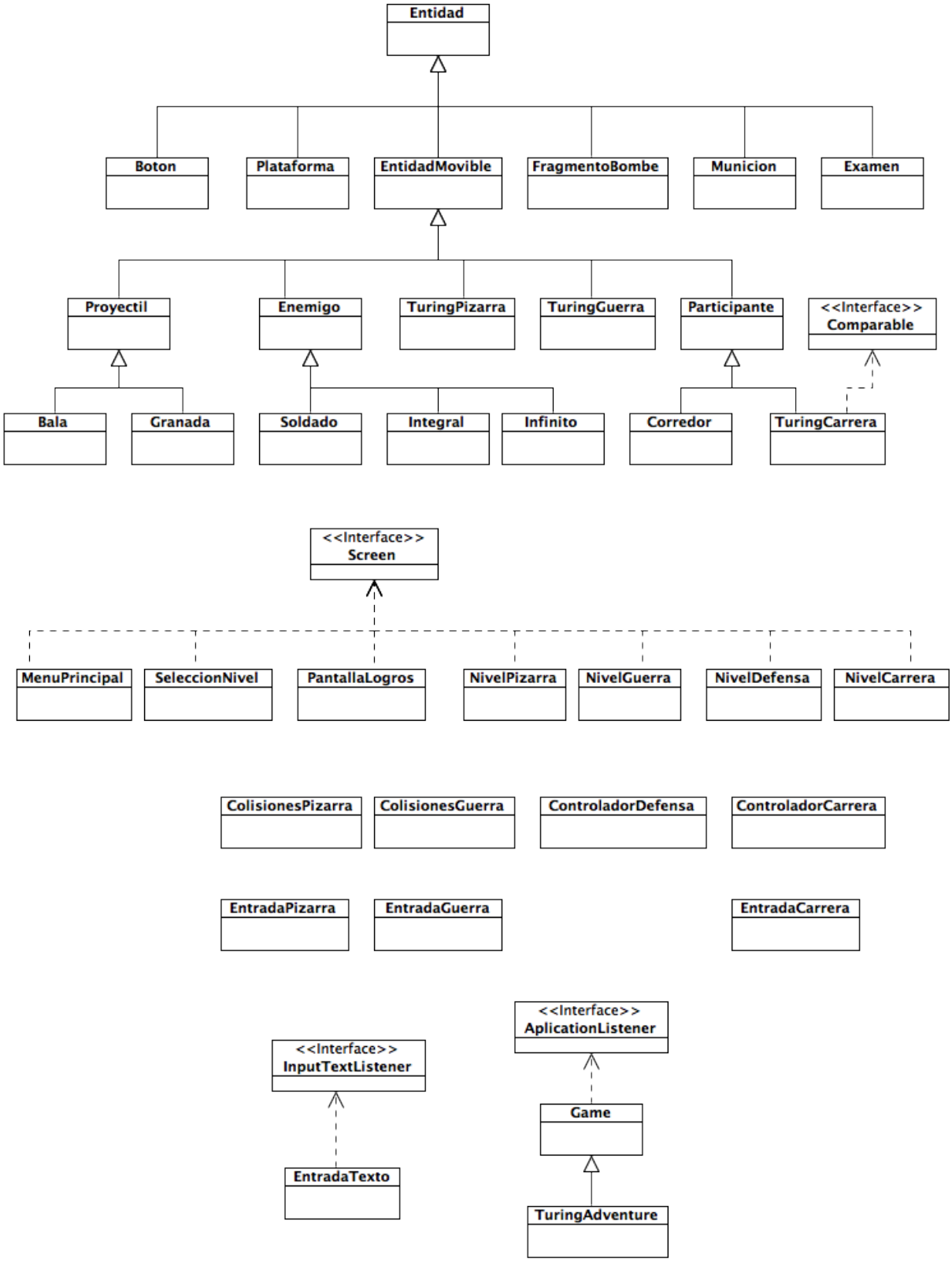


Figura A.1: Jerarquía de Clases

A.3. Clases y objetos comunes

A.3.1. OrthographicCamera

Es un objeto obligatorio para nuestro juego. Determina la sección de la pantalla que vamos a ver, es decir, si nuestro tamaño de pantalla definido en el main (o por defecto el tamaño de pantalla en Android) es de tamaño 1000x1000 y nuestra cámara es de tamaño 500x500, hará el efecto de zoom: en la pantalla de tamaño 1000x1000 estaremos viendo la sección de 500x500 expandida.
Por lo general lo mas cómodo es que el tamaño de la cámara sea el mismo que el de la pantalla, a no ser que nuestro objetivo sea precisamente el de usar algún aumento o reducción de la misma. Además, también nos servirá para movernos por el fondo en el caso de que este sea mas grande. Es decir, servirá para el scroll lateral.

A.3.2. Texture

Es el objeto básico de textura, con él cargaremos una imagen y lo utilizaremos para dibujarlo por pantalla.

A.3.3. SpriteBatch

Es la clase que se encarga de dibujar en pantalla utilizando las texturas. Literalmente significa 'grupo de imágenes'. Lo que hace es eso: agrupar todas las texturas que tenemos y crear una única imagen para dibujarla en el instante actual.

Se utiliza dentro del método render(), con la función draw() para ir dibujando imagen por imagen. Todas las imágenes dibujadas deben ir entre los métodos begin() y end(). Aqui vemos un ejemplo de su uso:

```
Texture fondo = new Texture("Fondo.png");

public void render(){
        batch.begin();
        batch.draw(fondo, x, y, altura, altura); // Es el metodo basico,
            hay muchos segun la necesidad
        batch.end();
```

A.3.4. Proporciones

Esto no es una clase, sino que es la forma de ajustar las proporciones que utilizaremos en todo el juego de manera que lo hagamos independiente a la resolución de pantalla o el dispositivo en el que lo ejecutemos.

Por supuesto, lo mas óptimo es adaptar todas las imágenes que tengamos a diversas resoluciones y que al empezar el juego carguemos unas u otras según corresponda. Pero eso tiene el inconveniente de que se necesita mucho espacio al instalar el juego y ademas para los que no se

nos de tan bien el manejo de gráficos puede volverse una tarea muy engorrosa. Además requiere mucho mas tiempo y buscamos una forma de aprender a programar lo mas rápida posible, sin utilizar demasiado tiempo en la realización de los gráficos.

Por ello hemos decidido realizar los gráficos una sola vez, ajustándolos a un tamaño determinado (1280x800 en nuestro caso) y que el juego detecte el tamaño de la pantalla y dibuje todo respetando la proporción original.

Es decir, si por ejemplo queremos dibujar un objeto a la mitad de la pantalla, esto sería a altura 400 píxeles en el caso del tamaño original, pero si nuestra pantalla actual es de 1000x500 la altura a dibujarlo seria a 250 píxeles.

Esto es tan simple como realizar una regla de tres: nueva altura = Altura de pantalla actual * Altura original / Altura de pantalla Original.

Esto se puede simplificar si obtenemos una constante (Altura actual / Altura original), que al multiplicarla por cualquier dimensión o posición en la que queramos dibujar obtendremos la proporción correspondiente. Lo hemos hecho de la siguiente forma:

```
anchuraJuego = Gdx.graphics.getWidth();
alturaJuego = Gdx.graphics.getHeight();

proporcionAncho = anchuraJuego/1280;
proporcionAlto = alturaJuego/800;
```

El único inconveniente de este método es que si usamos una resolución mayor a 1280x800 las imágenes se verán expandidas, aunque la solución es simplemente usar una base mayor (cosa que en este caso no se considera necesaria pues es un tamaño mas que de sobra para móviles y un tamaño decente para escritorio), o en el caso de que la resolución de aspecto sea distinta a 16:10 esta podría verse algo distorsionada, pero me temo que la única solución a este problema sí que es hacer imágenes adaptadas a cada resolución.

A.3.5. Entidad

Entidad es nuestro modelo base del que partirán el resto de objetos que estén delimitados por bordes. La clase esta compuesta por:

- Vector2 posicion: Es un vector de 2 elementos en el que almacenaremos la posición en la que se encuentra la Entidad
- Float anchura: La anchura de la Entidad
- Float altura: la altura de la Entidad
- Rectangle bordes: Es un rectángulo, así de simple. Una figura delimitada por cuatro Float (posicion x, posicion y, anchura y altura), cuya principal ventaja consiste en que podemos utilizar directamente este objeto para obtener los bordes de la Entidad y utilizarlo para

calcular las colisiones, ya que la clase Rectangle nos proporciona un método para saber si se superpone con otro Rectangle o si una posición concreta se haya dentro de el, aunque esto lo veremos mas adelante.

Esta es la clase base. Todos nuestros modelos contaran con estos atributos ademas de los propios, como por ejemplo la clase EntidadMovible en la que tendremos velocidad y dirección. Podemos ver todas las clases que heredan de Entidad en la sección de Jerarquía (Fig. A.2).

Todos los atributos son accesibles mediante métodos observadores y modificadores, a los que por simplicidad simplemente se les ha añadido **get** o **set** delante del nombre del atributo. Por ejemplo, para obtener la posición sería **entidad.getPosicion()**, que nos devuelve un Vector2 con la posición, y para modificarla **entidad.setPosicion(x, y)**. Y así con todos los atributos.

Mención especial a la clase EntidadMovible, y a todas las que hereden de ésta, ya que son objetos que al estar en movimiento deberán ser actualizados, por lo que se le añade un método **update()** que realizará como mínimo la actualización de la posición y de los bordes (igualándolos a la posición actual del objeto), ademas de las acciones propias del objeto.

```
public void update() {
        posicion.x = posicion.x + (direccion*(Gdx.graphics.getDeltaTime()*
            SPEED));

        bordes.x = posicion.x;
        bordes.y = posicion.y;
}
```

Este sería el código necesario para realizar el movimiento de cualquier objeto: simplemente le sumamos a la posición X la dirección (1, 0 o -1) por la velocidad. Añadimos también la variable de LibGDX interna **DeltaTime**, que es el tiempo que pasa entre un frame y otro, para que el movimiento sea igual independientemente de la tasa de refresco del dispositivo en el que se ejecuta, es decir, si la tasa de refresco es mayor, pasa menos tiempo entre un frame y otro, por lo que multiplicaremos por un numero menor, y viceversa. Así hacemos que la velocidad sea igual sin importar la velocidad del procesador.

A.3.6. Array e Iterator

Los utilizaremos conjuntamente cuando necesitemos grupos de elementos, por ejemplo proyectiles o simplemente para almacenar las plataformas o enemigos de un nivel.

Array sera el vector que almacene los objetos, mientras que Iterator sera el iterador especifico de ese Array, que lo recorrerá continuamente para dibujar cada elemento o procesarlo según corresponda. Veremos un ejemplo de uso concreto mas adelante.

A.4. Inicialización

Empezaremos explicando como comenzar nuestro videojuego: cual es la clase principal que se ejecuta cuando lanzamos nuestro ejecutable (TuringAdventure.jar).

Como es habitual, la clase que se llama al iniciar el programa es la clase main, esta va a ser una clase muy sencilla, que simplemente se encargará de establecer la configuración del sistema (si usamos OpenGL, el titulo del juego, la resolución de pantalla...) y de crear nuestro juego en si, que en nuestro caso sera la clase TuringAdventure:

```
public class Main {
        public static void main(String[] args) {
                LwjglApplicationConfiguration cfg = new
                    LwjglApplicationConfiguration();//Es el archivo de
                    configuracion
                cfg.title = "Turing's Adventure";
                cfg.useGL20 = true; //si es falso las dimensiones de las
                    imagenes tienen que ser potencia de 2
                cfg.width = 1280; // Resolucion de la pantalla
                cfg.height = 800;

                new LwjglApplication(new TuringAdventure(), cfg); // Crea
                    una nueva aplicacion con un objeto TuringAdventure (de
                    tipo Game) y la configuracion que le hemos dado
        }
}
```

Esta clase es específica del dispositivo en el que se ejecute. Debe existir una para desktop y otra para Android (cada una en su respectiva carpeta), ya que no se llamarán de la misma forma. Es una de las principales diferencias a la hora de importar el juego a distintas plataformas, por lo demás bastara con que creemos nuestras clases en la carpeta general del juego.

A.5. La clase principal

Para entender esta parte deberemos explicar tres conceptos, **Screen, AplicationListener y Game**:

- Screen serán las pantallas del juego, tienen las estructura ya vista de libGDX (Capítulo 2.4)

- AplicationListener: es una interfaz que proporciona métodos que se llaman cada vez que es necesario crear, pausar, continuar, renderizar o destruir una aplicación, nos permite ademas manejar los gráficos

- Game: Es la clase principal: implementa de AplicationListener y nos permite delegar en una Screen, es decir, que permite a la aplicación tener y manejar fácilmente varias ventanas

La clase principal de un juego en LibGDX debe heredar de la clase Game y es la que se llamará al inicio de la aplicación desde el main, en nuestro caso sera la clase TuringAdventure.

Como podemos imaginar, esta clase implementa la estructura básica de LibGDX. Sus funciones principales son **render(), resize(), show(), hide(), pause(), resume() y dispose()**. En esta clase no es necesario el uso de todas las funciones, ya que lo que nos interesa ahora mismo es crear Screens para empezar a mostrar nuestro juego, lo cual haremos en la función **create()**, pero podemos modificar el resto de funciones como queramos, decidir que hará el juego cuando pausamos la aplicación, cuando redimensionamos la pantalla o liberar los recursos que hayamos reservado.

Como observamos estamos utilizando la función **render()** para calcular 'logros' pero eso es algo que explicaremos mas adelante, de momento basta con saber que para lanzar una nueva pantalla de juego, debemos ejecutar lo siguiente:

```
setScreen(new MenuPrincipal(this));
```

De esta forma, lanzamos la clase MenuPrincipal, que como podremos adivinar hereda de la clase Screen, y recibe como parámetro la propia clase TuringAdventure, que deberá estar presente en todo el juego, ya que servirá para unificarlo todo.

Como su nombre indica, ésta llamada lanza el menú principal del juego, pero eso es conveniente explicarlo cuando tengamos un mayor conocimiento, por lo que pasaremos directamente al primer nivel, donde explicaremos como mostrar imágenes por pantalla y como realizar la entrada desde ratón y pantalla táctil.

Pasar a una pantalla de juego es tan simple como lanzar su correspondiente clase, que lógicamente también hereda de Screen. LibGDX alterna entre ventanas, de manera que cuando llamamos a otra ventana, ejecuta el método **dispose()** de la ventana actual, liberando los recursos y pasando a la ventana siguiente.

En nuestro caso llamaríamos a la ventana del primer nivel, que se corresponde con el tercer nivel de juego 'WW2 Descifrando el enigma', que sería:

```
setScreen(new NivelDisparos(this));
```

A.6. Nivel '1943: WW2. Descifrando el Enigma': Mostrar imágenes, entrada de ratón y colisiones básicas

A.6.1. Ambientación y objetivo

Este nivel se ubica a finales de la 2º Guerra Mundial. Alan Turing tuvo un papel importantísimo en la victoria británica ya que gracias a su desarrollo de la Máquina Bombe (Fig. A.9) los Ingleses consiguieron descifrar los códigos encriptados de la Maquina Enigma usada por los Alemanes. Pues bien, aquí nos encontramos en este último tramo de la guerra, descifrando dichos códigos, mientras los enemigos intentan frenar nuestro avance. Tendremos que resistir el tiempo que tarden los códigos en descifrarse destruyendo las granadas enemigas antes de que estas lleguen hasta nosotros y nos impidan completar nuestra tarea.

Será un nivel muy sencillo, en el que aprenderemos a mostrar imágenes por pantalla (aún sin animaciones). Daremos los primeros pasos en la entrada de datos: simplemente haciendo click con el ratón o de manera táctil (pulsando en las granadas para dispararles) y en las colisiones (detección de pulsación en las granadas y choque en los bordes de la pantalla), lo básico para empezar a ver los elementos principales en la creación de un videojuego.

A.6.2. Elementos del Nivel

Nos encontramos como ya hemos dicho en la clase NivelDefensa, una clase que hereda de Screen, por lo que tenemos los métodos necesarios para la creacion y renderizado de la pantalla.

Lo primero es crear los elementos necesarios, para este nivel:

- TuringAdventure juego: Es el elemento principal del juego, que se recibe en el constructor, necesario para conectar todas las pantallas.
- ControladorDefensa controlador: Es la clase que se encargará de la lógica de juego, al margen de la impresión por pantalla que se realizara en el **render()**, la describiremos mas adelante.
- SpriteBatch batch: nuestro agrupador de imágenes, el cual ya hemos descrito en Fig. A.3.3.
- Array<Granada>granadas: Un Array con las granadas que se irán generando en el juego.
- Iterator<Granada>IterGranadas: El iterador gracias al cual recorreremos el Array de granadas.
- BitmapFont visor: Es simplemente una fuente de texto que podremos mostrar por pantalla con la cadena de texto deseada, la usaremos para el visor del tiempo restante.
- OrthographicCamera camara: la cámara que usaremos en el nivel, explicada en Fig. A.3.1.

El único atributo que necesita una explicación algo mayor es el objeto granada. Estas son simplemente objetos EntidadMovible, a los que le hemos añadido cuatro atributos adicionales, uno para añadir la dirección en el eje Y a la EntidadMovible, otro para saber si ya hemos disparado a la granada y está explotando, y otros dos de tiempo para controlar el tiempo de la explosión.

```
int direccionY;//Anadimos la direccion en en eje Y, para que se pueda mover
    hacia abajo
boolean explotando; //Sabremos si esta explotando
long tiempoExplosion; //El tiempo que lleva explotando
long momentoExplosion; // El momento en el que explota
```

Con estos atributos en el método **update()** simplemente comprobaremos que si la granada no esta explotando se siga moviendo y en el caso de que lo esté, actualizamos el tiempo que lleva explotando. Esto lo hacemos restandole el instante en el que empezó a explotar al instante actual.

```
public void update() {
        if(!explotando){
                posicion.x = posicion.x + (direccion * (Gdx.graphics.
                    getDeltaTime() * SPEED));
                posicion.y = posicion.y + (direccionY * (Gdx.graphics.
                    getDeltaTime() * SPEED));
        }
        else{
                tiempoExplosion = TimeUtils.nanoTime() - momentoExplosion;
        }

        bordes.x = posicion.x;
        bordes.y = posicion.y;
}
```

Por último, tenemos el resto de elementos generales del nivel: Texturas, constantes (anchuras y alturas, proporciones, el tiempo que tenemos que aguantar), música y sonidos... y ya solo nos queda inicializar todos los atributos, lo cual haremos en el constructor.

A.6.3. Método render(): Mostrar Imágenes

Entramos de lleno en la función principal de una Screen, el método **render()**, aquí es donde se deberán llevar a cabo todas las acciones y lógica del juego, así como la impresión por pantalla, pues este método se ejecutara tantas veces por segundo como sea posible y habrá que ir actualizando todos los elementos del juego.

Vamos a describir los pasos obligatorios cuando ejecutamos el método **render()**:

Lo primero es limpiar la pantalla, para ello usaremos los siguientes métodos, extraídos de OpenGL:

```
Gdx.gl.glClearColor(0, 0, 0, 1);// Gdx es una clase con la que podemos
    acceder a variables que hacen referencia a todos los subsitemas, como
    son graficos, audio, ficheros, entrada y aplicaciones gl es una variable
     de tipo GL, nos permite acceder a metodos de GL10, GL11 y GL20. En este
     caso glClearColor es un bucle (game loop) que establecera el fondo de
    la pantalla negro (0,0,0) con transparencia 1
Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT); // Despues de la funcion anterior
     es necesario ejecutar esta, para que se lleve a cabo
```

Lo siguiente es actualizar la cámara:

```
camara.update();
```

Ahora toca utilizar nuestro SpriteBacht para dibujar los elementos: primero se le indica el sistema de coordenadas que utilizará (normalmente sera la proyección de la cámara) y posteriormente iniciamos el proceso de dibujado, que debe estar comprendido entre los métodos **begin() y end()** del SpriteBatch.

Entre estos dos métodos utilizaremos la función **draw()**, en cualquiera de sus sobrecargas para dibujar cada elemento. En este nivel en concreto, ademas comprobaremos si la granada ha explotado o no, para dibujarla normal o dibujar la explosión:

```
batch.setProjectionMatrix(camara.combined); // indicamos que el spritebatch
     usara el sistema de coordenadas establecido por la camara

batch.begin();

// Dibujamos las granadas
IterGranada = granadas.iterator();
while (IterGranada.hasNext()) {
        granada = IterGranada.next();
        granada.update();
        if(granada.getExplotando()){
                batch.draw(texturaExplosion, granada.getPosicion().x,
                    granada.getPosicion().y, anchuraExplosion,
                    alturaExplosion);
        }
        else{
                batch.draw(texturaGranada, granada.getPosicion().x, granada
                    .getPosicion().y, granada.getAnchura(), granada.
                    getAltura());
        }
}

//Dibujamos el tiempo restante
visor.draw(batch, "Aguanta " + tiempoRestante + " segundos", camara.
    position.x + 400, camara.position.y + 400);

batch.end();
```

Como vemos, aquí utilizamos el Iterador para recorrer el Array de objetos, actualizarlos para que realicen su movimiento y dibujarlos en su posición actual.

Aprovechamos también y dibujamos el visor, que simplemente recibe el SpriteBatch, la posición en la que se quiere dibujar y el texto a escribir.

Por último, después de haber dibujado, debemos actualizar todos los elementos del juego, para que todo se compruebe en cada frame y el juego avance.

El movimiento de las granadas ya lo hemos actualizado dentro de la propia impresión por pantalla, así que queda actualizar todo lo demás, comprobar si las granadas han llegado abajo de la pantalla para saber si hemos perdido o a los bordes de la pantalla para que reboten, actualizar el contador de tiempo, comprobar la entrada de ratón/pantalla táctil y ver si colisiona con alguna granada... Pero como queremos procurar modularizar todo lo posible tendremos que aislar cada elemento de juego. Esto lo haremos dentro de la clase ControladorDefensa, en concreto dentro de su método **update()**, por lo que el único paso restante en el **render()** sería:

```
controlador.update();
```

A.6.4. Controlador

Nos ubicamos en la clase ControladorDefensa, esta clase será una prolongación del **render()** pues contará con un método **update()** en el que realizaremos todos los cálculos referentes a la lógica de juego.

El controlador simplemente recibirá un objeto tipo Screen (en este caso nuestra Screen NivelDisparos) ya que mediante los métodos observadores podremos obtener todos los parámetros del nivel. Esto lo haremos en el constructor, obtendremos las granadas del nivel, los atributos de anchura y altura, y las proporciones. Ademas añadimos el concepto de 'tiempo', ya que en este nivel llevaremos una cuenta atrás, y tendremos que saber cada cuanto tiempo lanzar una nueva granada.

Esto lo haremos mediante el objeto de LibGDX TimeUtils, en concreto con su método **nanoTime()**, que nos devuelve en nanosegundos el valor actual del tiempo del sistema.

Obteniéndolo dos veces, una en la creación del controlador, y la otra en cualquier momento del juego, y restandolas, sabremos el tiempo que ha pasado, y lo mismo para el lanzamiento de granadas cada cierto tiempo, solo reiniciamos el contador al lanzar, y volvemos a empezar a contar.

```
tiempoInicial = TimeUtils.nanoTime();//obtenemos el momento en el que
    empezamos el juego
tiempoUltimaGranada = TimeUtils.nanoTime();
```

Y por fín comenzamos el método **update()**.

Primero vamos a comprobar si ha pasado el tiempo necesario para lanzar una nueva granada. Para ello le restamos al tiempo actual el tiempo en el que se lanzó la última granada. Si este tiempo es igual o mayor al tiempo establecido, lanzamos una nueva granada.

¿Como hacemos esto? Pues vamos a introducir el concepto de pseudoaleatoriedad, que es tan sencillo como utilizar la función de Java **Math.random()**, que nos devuelve un número entre 0 y 1(este no incluido). En este caso nosotros necesitamos simplemente un número entero (0, 1 o 2) para diferenciar entre las 3 posibles posiciones de lanzamiento (nuestros 3 enemigos). Para ello debemos multiplicar el resultado por 3, lo que nos dará un Double entre 0 y 3, y forzarlo a que sea entero, para quedarnos con los tres valores. Y lo mismo para elegir la dirección en la que lanzaremos la granada: puede ser hacia abajo, hacia la derecha o hacia la izquierda, por lo que al resultado del **Math.random()** ademas le restaremos 1, con lo que los valores que obtendremos serán -1, 0 o 1, obteniendo de esta manera la dirección.

Y por último, reiniciamos el tiempo de la última granada;

```
if(TimeUtils.nanoTime() - tiempoUltimaGranada > tiempoEntreGranadas){//Si
    ha pasado el tiempo, lanzamos granada
        posicionAleatoria = (int)(Math.random()*3); //Nos da un numero
            entre 0 y 3, para que lanze uno de los 3 enemigos
        direccionAleatoria = (int)(Math.random()*3); //Las 3 posibles
            direcciones en las que lanzar la granda, de frente, derecha o
            izquierda

        direccionAleatoria--; //direccionAleatoria nos devuelve 0, 1 o 2,
            si le restamos uno nos devuelve -1, 0 o 1, es decir, las
            posibles direeciones

        if(posicionAleatoria == 0){
                granadas.add(new Granada(new Vector2(proporcionAncho*100,
                    proporcionAlto*700), anchuraGranada, alturaGranada, 50,
                    direccionAleatoria, -1));
        }

        else if(posicionAleatoria == 1){
                granadas.add(new Granada(new Vector2(proporcionAncho*500,
                    proporcionAlto*700), anchuraGranada, alturaGranada, 50,
                    direccionAleatoria, -1));
        }

        else if(posicionAleatoria == 2){
                granadas.add(new Granada(new Vector2(proporcionAncho*800,
                    proporcionAlto*700), anchuraGranada, alturaGranada, 50,
                    direccionAleatoria, -1));
        }
        tiempoUltimaGranada = TimeUtils.nanoTime();
}
```

.

Lo siguiente que haremos será recorrer el Array de las granadas del juego, comprobar si hemos clickado en ellas para hacer que exploten, o por el contrario, si han alcanzado la parte inferior de la pantalla para dar fin al juego.

Esto lo hacemos con el método ya visto anteriormente: utilizamos un Iterator para recorrer el Array de granadas, y una vez tenemos cada granada comparamos:

- Para ver si estamos realizando una pulsación utilizamos los atributos internos de LibGDX, con **Gdx.input** accedemos al módulo de entrada, con **Gdx.input.justTouched()**, sabremos si la pantalla es tocada, y con **Gdx.input.getX() y getY()** obtenemos dichas coordenadas dentro de la pantalla. Ahora solo queda comprobar que estas coordenadas se encuentran dentro de los bordes de la granada. podríamos hacerlo manualmente, pero para ahorrarnos esto creamos el atributo 'bordes' propio de cada objeto Entidad (Fig. A.3.5). Gracias a este atributo de tipo Rectangle, podemos acceder al método **contains(x,y)**, que pasandole la posición (también sobrecargado para pasarle otro Rectangle) nos dirá si el punto esta contenido dentro de dichos bordes.

- Comprobamos por el contrario si la granada ya ha explotado, y en ese caso, si ha pasado su tiempo de explosión, simplemente deberemos eliminarla del Array.
  Ahora comprobamos si la granada ha llegado a nuestra posición, dando fin al nivel, lo cual es tan simple como ver si la posición en el eje Y es menor al borde inferior de la pantalla.

- Y hacemos lo propio con los bordes derecho e izquierdo de la pantalla, pero debemos diferenciar ambos casos: para comprobar la colisión por la izquierda es muy simple, las coordenadas se miden desde la esquina inferior izquierda, tanto de la pantalla como de los objetos, por lo que comprobamos que la posición X de la granada no sea menor que 0, y que ésta no se este moviendo hacia la izquierda, si es así simplemente le cambiamos la dirección. Pero para la colisión derecha hay que comprobar algo mas: el borde derecho de la granada es la posición en X más la anchura de la granada, y hay que comprobar que ésta posición no es mayor que la anchura de la pantalla. Con esta simple comprobación tendremos listas las colisiones.

Y por último actualizamos el tiempo que llevamos de juego, que es el momento actual obtenido con TimeUtils menos el tiempo inicial, y el tiempo que nos queda de la cuenta atrás (que es el tiempo total de juego menos el tiempo actual).

```
Iterator<Granada> IterGranadas = granadas.iterator();
        while(IterGranadas.hasNext()){//Recorremos las granadas
                Granada g = IterGranadas.next();

                if(!g.getExplotando() && Gdx.input.justTouched() && g.
                    getBordes().contains(Gdx.input.getX(), Gdx.graphics.
                    getHeight() - Gdx.input.getY())){//Si no estan ya
                    explotando y los bordes de la granada contienen el punto
                     donde tocamos
                        g.setExplotando(true);//explotan
                }
                if(g.getExplotando() &&  g.tiempoExplosion() >=
                    tiempoExplosion){
                        IterGranadas.remove();//Pasada la explosion,
                            desaparaecen
                }
                if(g.getBordes().y <= 0){//Y si las granadas llegan al
                    suelo, perdemos
                        juego.getJuego().setScreen(new MenuPrincipal(juego.
                            getJuego()));
                }

                //Controlamos que no se salga de la pantalla, que rebote
                if(g.getBordes().x <= 0 && g.getDireccion() < 0 ){//Si se
                    sale por la izquierda y se esta moviendo hacia la
                    izquierda
                        g.cambiarDireccion();//Cambiamos la direccion
                }
                else if(g.getBordes().x + g.getAnchura() >= Gdx.graphics.
                    getWidth() && g.getDireccion() > 0 ){//Si se sale por la
                     derecha y se esta moviendo hacia la derecha
                        g.cambiarDireccion();//Cambiamos la direccion
                }
        }
tiempoActual = TimeUtils.nanoTime() - tiempoInicial;
juego.setTiempoRestante(tiempoMaximo - tiempoActual);
```

Finalmente deberemos comprobar si hemos superado la cuenta atrás. Si el tiempo actual es mayor que el tiempo de la cuenta atras, habremos superado el nivel.

```
if(tiempoActual > tiempoMaximo){// Si el tiempo de juego (el actual menos
    el momento en el que empezamos) es mayor al tiempo maximo que tenemos
    que aguantar
        juego.getJuego().setScreen(new MenuPrincipal(juego.getJuego()));
}
```

Con esto concluimos el primer nivel, en el que hemos repasado todos los aspectos básicos de un videojuego. Ya sabemos mostrar imágenes por pantalla, realizar colisiones muy básicas, interactuar con el ratón/pantalla táctil y dar movimiento a los objetos.

A.7. Nivel '1948: Maratón Olimpiadas Londres' : Animaciones, teclado, botones y adelantamiento en carrera

A.7.1. Ambientación y objetivo

Este nivel sucede en el año 1948, mas concretamente en las clasificatorias de la maratón para las Olimpiadas de Londres, en las que Alan Turing participó, y aunque no tuvo la suerte de clasificarse para los juegos, quedó en un mas que decente quinto puesto (quedando tan solo a 10 minutos del posterior ganador de la medalla de oro en la competición).

Esto es algo que dará juego a la hora de crear el nivel, pues no se basa en una simple carrera, en la que los contrincantes corren independientes y si estamos los primeros ganamos y si no perdemos: en este caso deberemos de buscar la manera de que Turing no pueda sobrepasar la quinta posición, por lo que tendrá que interactuar con el resto de corredores.

Ademas de esto, en este nivel veremos como crear las animaciones de los corredores, añadiremos la entrada de teclado, ya que la forma de avanzar será la pulsación consecutiva de las flechas derecha e izquierda (y por lo tanto veremos también como sincronizarlas para que la velocidad aumente o disminuya según la velocidad a la que pulsemos) y la aparición de botones en Android (diferenciando por lo tanto según el dispositivo) con la misma funcionalidad que las flechas, añadiendo ademas un módulo especifico para ello, ya que en este nivel la entrada de datos será mas compleja y nos resultara más cómodo realizarla en un módulo independiente.

A.7.2. Elementos del nivel

Vamos a ver los elementos de la clase NivelCarrera. Aquí tendremos los mismos elementos básicos y necesarios que ya vimos en el nivel anterior, como la clase principal del juego TuringAdventure, el controlador del nivel, el SpriteBatch, OrthographicCamera, Textures, constantes de tamaño, proporciones y velocidad.

En primer lugar debemos comentar una cosa con respecto a la cámara: se diferencia en un punto importante con respecto al nivel anterior, y es que esta cámara se mueve con la pantalla, por lo que justo antes de actualizarla en el método **render()** debemos realizar dicho movimiento para que siga a nuestro personaje.

Esto es tan simple como que la posición X de la cámara (que recordamos: hemos establecido el origen de ésta en su centro) coincida con la posición X de Turing. Pero además, le vamos a añadir una particularidad. Solo haremos que siga este movimiento si estamos dentro del fondo de juego, ya que si simplemente hacemos que lo siga, al principio y al final del nivel habrá un trozo negro a la izquierda y a la derecha de la pantalla (Fig. A.2).
Para que esto no ocurra debemos hacer que la cámara solo siga a Turing si la posición X de Turing no ha llegado aun al centro (la mitad) de la cámara. Y lo mismo al final del nivel, que no sobrepase el ancho del juego menos la mitad del ancho de la cámara.

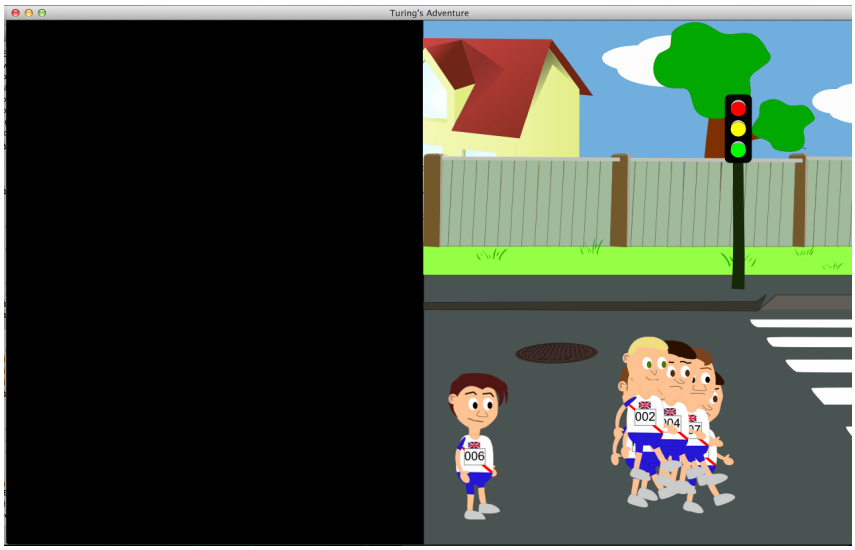


Figura A.2: Cámara sin control

```
if(turing.getPosicion().x >= camara.viewportWidth/2 && turing.getPosicion()
    .x <= finNivel - camara.viewportWidth/2){
        camara.position.x = turing.getPosicion().x;
}
camara.update();// Se actualiza la camara, obligatorio
```

Ahora nos centraremos en los elementos nuevos, principalmente para el manejo de animaciones, pero también encontraremos algunos más:

- Participante: Serán los participantes de la carrera, heredan de EntidadMovible, pero como añadido implementan la interfaz Comparable de Java, que nos permitirá comparar su atributo característico: la posición en la carrera.
- TuringCarrera turing: Es nuestro personaje principal, simplemente hereda de la clase Participante. Lo único que tenemos que calcular en su método **update()** es la actualización del movimiento y de la animación, ademas de la reducción de movimiento si no pulsamos correctamente los botones, lo cual veremos en el siguiente apartado.
- Corredor: serán nuestros contrincantes. También heredan de la clase Participante, aunque éstos si que tienen mas trabajo que hacer: necesitan al igual que Turing actualización de movimiento y animación, pero ademas deben comprobar la posición en la que van para no ser adelantados por Turing. Y como añadido le modificaremos la velocidad aleatoriamente, para que cada uno sea independiente de los demás.
- Tendremos nuestro Array de Corredores con su correspondiente Iterator. De momento estarán diferenciados los corredores de Turing para poder dibujarlos independientemente, pero en el Controlador los unificaremos en un único Array de Participantes.
- Boton: es una simple Entidad, que utilizaremos para crear los botones en Android.

– TextureRegion frameActual: habrá uno para cada animación, un TextureRegion es, como su nombre indica, una región dentro de una textura, ya que las animaciones vendrán en un sprite de imágenes y de ahí seleccionaremos el correspondiente a dibujar en el frame actual.

– TextureRegion [ ][ ] turingFrames: también necesario para cualquier otra animación, aqui almacenaremos todos los frames (TextureRegion) de la animación, y con frameActual seleccionaremos el correspondiente.

– Animation animacionTuring: el objeto animación, veremos como usarlo mas adelante.

– float StateTime: es un atributo propio de EntidadMovible que no hemos mencionado hasta ahora ya que el nivel anterior no lo requería al no haber animación. StateTime son los segundos que pasan desde el comienzo de una animación. Se usa para determinar el estado de la animación.

Ademas hay un elemento que no es necesario inicializar ya que se usa directamente, y es la clase encargada de procesar la entrada de datos al juego, que en este nivel ya será algo mas compleja y podemos abstraerla en un módulo aparte. Pero de esta clase hablaremos un par de secciones mas adelante.

A.7.3. Animación

La animación se dividirá en varias partes. Lo primero es la inicialización de las variables que hemos descrito anteriormente, que realizaremos en el propio constructor del nivel. Para ello necesitamos conocer el archivo de imagen con los sprites de la animación que vamos a utilizar. Será un archivo de imagen en forma de matriz (dividido en filas y columnas) en el que cada celda será un frame de la animación.

Pues bien, necesitamos meter cada una de esas 'subimagenes' en el el vector de TextureRegion[ ] turingFrames (o el de la animación correspondiente). Para ello primero obtenemos el tamaño de cada imagen de la animación (simplemente dividimos el ancho de la imagen completa entre el número de columnas, y el alto entre el número de filas).

Ahora crearemos una matriz temporal de TextureRegion mediante el método **split(Texture textura, int ancho, int alto)** pasandole la imagen original y el ancho y alto de las imágenes de la animación. Ahora simplemente tenemos que trasladar el contenido de esta matriz temporal al vector de TextureRegion [] turingFrames, lo cual hacemos con dos bucles for anidados.

Figura A.3: Animación Turing

```
//Preparamos la animacion de turing
filas = 5;
columnas = 5;
anchura = texturaTuring.getWidth() / columnas;
altura = texturaTuring.getHeight() / filas;

tmp = TextureRegion.split(texturaTuring, (int)anchura, (int)altura);
// creamos un array bidimensional temporal que contiene las imagenes de la
   animacion, para separarlos recibe
// la anchura del mismo dividido entre el numero de columnas y la altura
   entre el numero de filas

turingFrames = new TextureRegion[filas * columnas];
indice = 0;
for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {// aqui rellenamos el vector
           con cada imagen
                turingFrames[indice++] = tmp[i][j];
        }
}
```

Ya tenemos un vector con todas las imágenes de la animación, ahora debemos crear la animación propiamente dicha simplemente mediante el constructor de Animation. Le pasamos el vector de frames y el tiempo que queremos que dure cada imagen. Para hacerlo siempre uniforme y siguiendo el mismo patrón haremos que las animaciones duren un segundo, por lo que el tiempo de cada imagen sera 1 dividido entre el número de imágenes.

```
animacionTuring = new Animation(0.04f, turingFrames); // y creamos la
   animacion con el array de sprites. El primer parametro es el tiempo que
   dura cada imagen, es decir, un segundo dividido entre el numero de
   imagenes
```

Con esto tenemos lista la inicialización de todas las variables, ahora toca utilizarlas, lo cual haremos en el método **render()**.

Lo primero que debemos hacer es obtener el StateTime de nuestra Entidad a animar. Como ya hemos dicho el StateTime son los segundos que pasan desde el comienzo de una animación. De esta manera sabremos qué elemento de la animación toca dibujar.

Para calcular el StateTime debemos hacerlo dentro del método **update()** de nuestra EntidadMovible, y es incluso mas sencillo que calcular su movimiento.

Si la velocidad es 0 estaremos parados, así que el StateTime es 0, y si nos estamos moviendo, iremos incrementándolo sumándole DeltaTime, que como hemos dicho es el tiempo que pasa entre un frame y otro, es decir, con StateTime sabemos el tiempo que ha pasado desde el comienzo de la animación. Pero es mas, no tenemos porque preocuparnos de reiniciarlo una vez termine la animación, ya que esto se realiza de manera automática. Una vez que ha transcurrido el tiempo que hemos estimado de animación, comienza de nuevo, por lo que con una sola línea podemos calcular el estado de la animación.

```
//Movemos a Turing, aumentamos su posicion en el eje x en funcion de su
    velocidad
posicion.x = posicion.x + (velocidad * (Gdx.graphics.getDeltaTime() * SPEED
    ));

if(velocidad == 0){
        stateTime = 0;
}
else{
        stateTime = stateTime + Gdx.graphics.getDeltaTime();
        // como nos estamos moviendo modificamos el stateTime, que servira
            para acceder a un nuevo frame en la animacion
}
```

Continuando en el método **render()**, una vez obtenido el StateTime del personaje, debemos obtener la imagen a dibujar, para ello utilizamos el método de la clase Animation **getKeyFrame(StateTime, true)**, que nos devolverá la imagen correspondiente a este StateTime, y lo asignamos a la TextureRegion frameActual, que irá almacenando el frame a dibujar cada segundo.

```
// Preparamos a Turing para ser dibujado (el momento de la animacion)
stateTime = turing.getStateTime();

frameActual = animacionTuring.getKeyFrame(stateTime, true); // el
    frameActual es el frame numero
// "stateTime" del vector de imagenes
```

Ya tenemos la imagen que queremos dibujar en este instante, ya solo queda el proceso habitual, dentro de **batch.begin() - batch.end()**: dibujamos nuestro personaje como lo haríamos normalmente, pero usando la imagen que hemos obtenido:

```
// Dibujamos el frame actual en la posicion en la que se encuentra turing
batch.draw(frameActual, turing.getPosicion().x, turing.getPosicion().y,
    turing.getAnchura(), turing.getAltura());
```

Con esto tenemos lista la animación. La única diferencia al hacer la de los corredores (o de algun elemento múltiple), es que como recorreremos el Array que los contiene dentro del **begin-end** del SpriteBatch, es ahí donde debemos obtener su StateTime. Ademas, en este caso los corredores pueden ser de tres tipos, para que no tengan todos la misma animación, por lo que según su tipo dibujaremos una animación u otra.

```
//Dibujamos los corredores
IterCorredor = corredores.iterator();
while(IterCorredor.hasNext()){
        corredor = IterCorredor.next();
        corredor.update();
        stateTimeCorredor = corredor.getStateTime();
        if(corredor.getTipo() == 1){
                frameActualCorredor = animacionCorredor1.getKeyFrame(
                    stateTimeCorredor, true);
        }
        else if(corredor.getTipo() == 2){
                frameActualCorredor = animacionCorredor2.getKeyFrame(
                    stateTimeCorredor, true);
        }
        else if(corredor.getTipo() == 3){
                frameActualCorredor = animacionCorredor3.getKeyFrame(
                    stateTimeCorredor, true);
        }
        batch.draw(frameActualCorredor, corredor.getPosicion().x, corredor.
            getPosicion().y, corredor.getAnchura(), corredor.getAltura());
}
```

A.7.4. Participantes: Control de velocidad y adelantamiento

Aunque no tiene que ver con la velocidad, vamos a comentar una característica de la clase general Participante, y es que al implementar la interfaz Comparable de Java, ésta posee un método que nos permite definir como se van a comparar entre si dos elementos de la misma clase.

En este método simplemente haremos que se compare la posición en X, devuelva 1 si es mayor, 0 si son iguales y -1 si es menor:

```
public int compareTo(Participante o) {
        if(this.posicion.x < o.posicion.x){
                return -1;
                }
        else if(this.posicion.x == o.posicion.x){
                return 0;
        }
        else{
                return 1;
        }
}
```

Así, a la hora de ordenar el Array que los contenga se ordenarán por su posición en X.

Ademas, la clase Participante cuenta con una variable long tiempo, que servirá para dos funciones distintas según sea Turing u otro Corredor el que la utilice.

Veamos ahora la característica especial de Turing en este nivel. Para él, la variable tiempo sera el tiempo desde la ultima pulsación correcta de las flechas de dirección (es decir, que sean consecutivas y dentro del tiempo establecido). Si este tiempo es menor al correspondiente, su velocidad irá disminuyendo de manera que nos veremos obligados a pulsar correctamente los botones para que Turing avance. Esta sincronización la veremos en la descripción del controlador.

En cambio, para los Corredores, la variable tiempo sera simplemente un contador temporal, que cada cierto tiempo (por ejemplo un segundo) irá incrementando su velocidad, para que la aceleración sea gradual. Ademas, una vez llegado a la velocidad máxima ira variando su velocidad, para que no tengan todos una velocidad estática. Así, simplemente cada vez que pase un segundo y no hayamos llegado a la velocidad mínima, incrementaremos la velocidad aleatoriamente.

Si por el contrario hemos sobrepasado la velocidad mínima, hay un 50% de probabilidad de que la velocidad aumente o disminuya (siempre que no se pase la velocidad máxima). De esta forma cada Corredor se comportará de forma independiente.

```
if(TimeUtils.nanoTime() - tiempo > 1000000000){//Si ha pasado mas de un
    segundo
        if(velocidad < 15){
                velocidad += Math.random()*1 + 1;//Nos devuelve un numero
                    entre 1 y 2(este no incluido)
        }
        else{
                if(Math.random()*2 <= 1 && velocidad < 20){//aumenta o
                    disminuye la velocidad con una probabilidd del 50%, si
                    no hemos llegado a la velocidad maxima
                        velocidad+=1;
                }
                else{
                        velocidad-=1;
                }
        }

        tiempo = TimeUtils.nanoTime();
}
```

Pero con esto, la velocidad de los corredores es aleatoria, y nos interesa que 4 de ellos no dejen que Turing les adelante para intentar representar lo que ocurrió en aquel momento.

Esto lo hacemos teniendo a unos corredores especiales, que simplemente tendrán una variable de tipo boolean ganador=true, y otra variable del mismo tipo, sprint=false, que activaremos desde el controlador cuando Turing los adelante, y de esta forma aumentar su velocidad sin importar si hemos llegado al máximo.

```
if(sprint ){
        velocidad++;
        sprint = false;
}
```

Con esto tenemos listo el control de la velocidad interna de los corredores, ahora vamos a ver como gestionar las posiciones y los adelantamientos.

A.7.5. Controlador

Pasamos al controlador del nivel: 'ControladorCarrera', en el cual realizaremos la lógica del juego. En este caso tenemos que gestionar las posiciones y según estas, hacer que los contrincantes indicados adelanten a Turing si este les sobrepasa.

El controlador se inicia como ya hemos visto en el constructor del nivel, posee un método **update()** que es donde irá todo el código y ejecutaremos por cada pasada del **render()**. Para empezar debemos inicializar todas las variables, en este caso serán el propio nivel, para tener acceso a sus elementos, Turing y los corredores.

Hasta ahora hemos diferenciado a estos dos últimos, los corredores los almacenábamos en un Array específico mientras que a Turing lo instanciabamos independientemente. Pues ahora necesitamos manejarlos a todos conjuntamente, ya que serán participantes de la misma carrera, por lo que creamos un Array de Participantes y los metemos a todos en él:

```
public ControladorCarrera(NivelCarrera juego){
        this.juego = juego;
        this.turing = juego.getTuring();
        this.corredores = juego.getCorredores();

        participantes = new Array<Participante>();

        participantes.add(turing);
        participantes.addAll(corredores);
}
```

Ahora comienza el método **update()**, aquí vamos a recalcular las posiciones de cada Participante. Como al introducirlos a todos lo hemos hecho de forma ordenada según su posición, y procuraremos que siempre lo estén, le reasignaremos posiciones según su propia posición en el Array, de mayor a menor.

```
Iterator<Participante> IterParticipante = participantes.iterator();
int i = participantes.size;
while(IterParticipante.hasNext()){
        Participante P = IterParticipante.next();
        P.setPuesto(i--);
}
```

Ahora haremos el cálculo del sprint. Esto es tan sencillo como recorrer de nuevo el Array, y si el corredor es uno de los ganadores y la posición de Turing es mejor que la de este, hacemos que esprinte.

```
if(turing.getPuesto() < 5){
        Iterator<Corredor> IterCorredor = corredores.iterator();
        while(IterCorredor.hasNext()){
                Corredor C = IterCorredor.next();
                if(C.getGanador() && turing.getPuesto() < C.getPuesto()){
                        C.setSprint(true);
                }
        }
}
```

Por último, nos interesa que todos estén siempre bien ordenados dentro del Array, para que cuando se asignen las posiciones en el primer paso estas sean correctas.
Hacerlo es la parte mas sencilla, ya que como hemos ya establecido la forma de comparación de los Participantes (por la posición en X), simplemente con ordenar automáticamente el Array estos se colocaran según su posición en el eje X, y por lo tanto en la carrera. Así cuando le reasignemos las posiciones sabemos que estas serán correctas.

A.7.6. Entrada de Datos: Teclado y Botones

Vamos a ver ahora como realizar la entrada por teclado, como crear botones para Android y como hacer que se sincronicen para que haya que pulsarlos consecutivamente.

Pero primero vamos a explicar un nuevo concepto: InputProcessor. InputProcessor es la interfaz básica de LibGDX que se encarga de manejar los eventos de entrada. Se compone de los métodos necesarios para cada tipo de entrada, es decir, que si nuestra clase EntradaCarrera implementa esta interfaz, dispondrá de todos estos métodos y podremos utilizarlos cómodamente en lugar de especificar expresamente qué tecla pulsamos cada vez o si estamos pinchando en la pantalla. Lo único que recibirá la clase será el propio nivel, para que tenga acceso a sus parámetros.

Lo mejor de todo es que no hace falta crear e inicializar la clase en el nivel, sino que hacemos que el procesador básico de entrada se sustituya por nuestra clase. Para ello dentro del constructor del nivel ejecutamos la siguiente instrucción:

```
//Establecemos el procesador de E/S
Gdx.input.setInputProcessor(new EntradaCarrera(this));// hacemos que el
    procesador de entrada de GDX sea nuestra clase EntradaCarrera
```

Necesitaremos obtener tres elementos para poder manejar la entrada: A Turing por supuesto, para saber su estado y modificarlo según corresponda, y los dos botones que usaremos en Android, para ello utilizamos las funciones observadoras del nivel:

```
public EntradaCarrera(NivelCarrera juego) {
        this.juego = juego;
        this.turing = juego.getTuring();
        this.botonDcha = juego.getBotonDcha();
        this.botonIzq = juego.getBotonIzq();
}
```

Aqui solo necesitaremos realizar acciones en el momento de PULSAR teclas. En concreto las de dirección derecha e izquierda, por lo que nuestro código ira dentro de la función **keyDown(int keycode)**.

Ahora la manera de controlar las teclas es tan simple como utilizar un switch y acceder a las teclas a través del módulo de entrada de LibGDX. En cuanto a la lógica, debemos sincronizar las dos teclas. Para ello dentro de la clase TuringCarrera tenemos una variable tipo boolean control, que será true si hemos pulsado izquierda y false si hemos pulsado derecha. Con esta tecla de control sincronizar es tan fácil como decir que si pulsamos izquierda, la variable de control es true. Sí el tiempo desde la ultima pulsación (sea cual sea) es menor que un segundo (recordemos que la variable de tiempo de Turing indicaba el tiempo entre pulsaciones) y la velocidad de Turing no ha llegado al máximo, entonces cambiamos el control a false y aumentamos la velocidad.
Ademas, independientemente de si la pulsación es correcta o no, reiniciamos el contador de tiempo de Turing, para que empiece a contar desde esta pulsación. Ahora hacemos lo mismo pero con la tecla derecha y listo.

```
public boolean keyDown(int keycode) {
        switch(keycode){
        case Keys.LEFT:
                if(turing.getControl() == true && TimeUtils.nanoTime() -
                    turing.getTiempo() < 1000000000 && turing.getVelocidad()
                     < juego.getVelocidadMaxTuring()){
                        turing.setControl(false);
                        turing.AumentarVelocidad();
                }
                turing.setTiempo(TimeUtils.nanoTime());
                break;
        case Keys.RIGHT:
                if(turing.getControl() == false && TimeUtils.nanoTime() -
                    turing.getTiempo() < 1000000000 && turing.getVelocidad()
                     < juego.getVelocidadMaxTuring()){
                        turing.setTiempo(TimeUtils.nanoTime());
                        turing.setControl(true);
                        turing.AumentarVelocidad();
                }
                turing.setTiempo(TimeUtils.nanoTime());
                break;
        }
        return true;
}
```

Toca el turno de los Botones. Antes de ver como procesar la entrada y detectar cuando pulsamos en ellos, veremos como dibujarlos, ya que tenemos que tener en cuenta que solo queremos que se dibujen si estamos en Android, y que los botones se mueven con la pantalla, dando el efecto de que son estáticos para el jugador.

Para ello, dentro del método **render()** comprobaremos si estamos en Android mediante el módulo de aplicación de LibGDX, y si es así modificaremos la posición de los botones para dibujarlos en el punto actual que queremos de la pantalla.

Para ello dibujaremos en función de la cámara. Debemos recordar que el punto inicial de la cámara esta en el centro en lugar de en la esquina inferior izquierda, por lo que para obtener nuestro punto de inicio habitual le restamos a la posición en X de la cámara, la mitad del ancho de la misma: con esto tenemos el borde izquierdo de la pantalla. Después le sumamos la posición en la que queremos dibujar el botón y ya lo tenemos en la posición en X deseada.

Hacemos lo mismo para la posición Y y tendremos la posición en la que debemos dibujar los botones. Lo siguiente es eso, dibujarlos, con el método habitual:

```
//Si estamos en android, dibujamos los botones
if(Gdx.app.getType() == ApplicationType.Android){
        //camara.position.x nos da la posicion de la camara en el juego (la
            camara se inicia en su centro)
        //carama.viewportWidth nos da la anchura de la camara

        boton_izq.setPosicion(new Vector2(camara.position.x - camara.
            viewportWidth/2 + boton_izq.getInicialX(), camara.position.y -
            camara.viewportHeight/2 + boton_izq.getInicialY()));

        boton_dcha.setPosicion(new Vector2(camara.position.x - camara.
            viewportWidth/2 + boton_dcha.getInicialY(), camara.position.y -
             camara.viewportHeight/2 + boton_dcha.getInicialY()));

        batch.draw(texturaBotonIzq, boton_izq.getPosicion().x, boton_izq.
            getPosicion().y, boton_izq.getAnchura(), boton_izq.getAltura());

        batch.draw(texturaBotonDcha, boton_dcha.getPosicion().x, boton_dcha
            .getPosicion().y, boton_dcha.getAnchura(), boton_dcha.getAltura
            ());
}
```

Volviendo al controlador de Entrada, nos vamos al método encargado de controlar la entrada por pantalla (sirve tanto para pantalla táctil como de teclado).

Como vemos, este método recibe cuatro enteros, la coordenada X e Y en las que pulsamos (eso si, desde la esquina superior izquierda en lugar de desde la inferior), un puntero para el evento (que no se suele usar) y el botón que se pulse, si es un botón característico de Android, tal y como el botón Home.

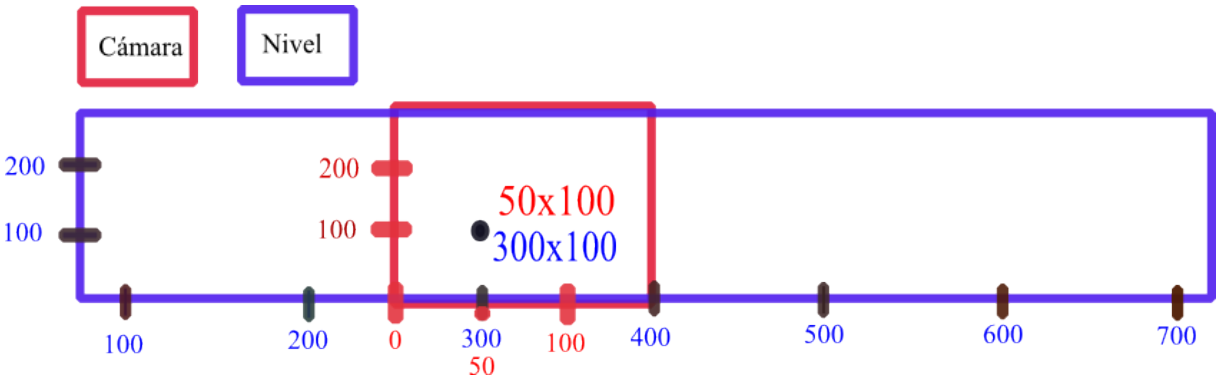


Figura A.4: Ejemplo de desproyección de la cámara

Pero tenemos un problema, y es que la posición de las coordenadas es con respecto a la propia cámara: es decir, si la cámara tiene una dimensión de 1280x800, nos devolverá un posición interna a esas dimensiones, y nosotros lo que queremos es un punto en el juego. Para obtener este punto podemos desproyectar la cámara en un puntero (Vector de 3 dimensiones) auxiliar. Esto es algo más complejo así que mejor verlo:

```
public boolean touchDown(int screenX, int screenY, int pointer, int button)
    {
        if(Gdx.app.getType() != ApplicationType.Android){
                return false; // Si no estamos en android, no hay botones,
                    asi que devuelve falso
        }

        juego.getCamara().unproject(puntero.set(screenX, screenY, 0));
        //screenX y screenY devuelven las coordenadas en la camara, es
            decir, 0 < screenX < anchura de la camara, y lo mismo para la Y,
             pero haciendo unproject obtenemos el valor de dichas
            coordenadas en el juego
        .
        .
        .
```

Puede parecer algo lioso, pero si nos fijamos bien, lo que hacemos simplemente es desproyectar la posición (ScreenX y ScreenY) de la cámara, convirtiéndolos en la posición correspondiente al juego y pasándosela al Vector3 puntero, de manera que ahora solo tenemos que usar el método ya visto de la clase Rectangle para ver si ese punto esta contenido en el botón:

```
        .
        .
        .
        if(botonIzq.getBordes().contains(puntero.x, puntero.y)){
                if(turing.getControl() == true && TimeUtils.nanoTime() -
                    turing.getTiempo() < 1000000000 && turing.getVelocidad()
                     < juego.getVelocidadMaxTuring()){
                        turing.setControl(false);
                        turing.AumentarVelocidad();
                }
                turing.setTiempo(TimeUtils.nanoTime());
        }

        else if(botonDcha.getBordes().contains(puntero.x, puntero.y)){
                if(turing.getControl() == false && TimeUtils.nanoTime() -
                    turing.getTiempo() < 1000000000 && turing.getVelocidad()
                     < juego.getVelocidadMaxTuring()){
                        turing.setTiempo(TimeUtils.nanoTime());
                        turing.setControl(true);
                        turing.AumentarVelocidad();
                }
                turing.setTiempo(TimeUtils.nanoTime());
        }
        return true;
}
```

Por lo demás, la lógica de la sincronización es la misma que la que usamos con las teclas.
Y con esto finalmente hemos completado el nivel.

A.8. Nivel '1934: Avanzando en Cambridge': Física de salto y colisión con obstáculos y enemigos

A.8.1. Ambientación y objetivo

Éste será el primer nivel cronológicamente, ya que esta ambientado en el paso de Turing por la Universidad de Cambridge en 1934, donde deberá enfrentarse a complicados problemas y fórmulas matemáticas hasta llegar a aprobar los exámenes y completar sus estudios universitarios.

El objetivo aquí será explicar una introducción a las físicas básicas usando para ello el salto de Turing y sobre todo profundizaremos en las colisiones, que aunque hemos visto una leve introducción en los choques con los bordes de la pantalla, ahora se verán complicados introduciremos obstáculos y plataformas, que nos impedirán el paso y nos permitirán subirnos en ellas, ademas de las colisiones con los enemigos, que depende de por donde choquemos los eliminaremos o por el contrario seremos derrotados.

A.8.2. Elementos del nivel

Lo primero, como siempre, vamos a analizar todos los elementos de la clase NivelPizarra. Por suerte en este nivel no vamos a ver objetos nuevos, sino que la parte importante será la lógica, por lo que todos los elementos ya nos resultarán conocidos.

La clase implementa a la clase Screen y recibe como parámetro nuestro objeto de juego principal TuringAdventure. Ademas tendremos:

- Un objeto ColisionesPizarra, que será nuestro controlador del nivel.
- El SpriteBatch para dibujar todas las texturas.
- Nuestro personaje, que sera de tipo TuringPizarra.
- Tendremos un Array de Plataformas, que no serán mas que objetos heredados de Entidad, pero con un atributo int 'tipo', para poder dibujar distintos tipos de plataforma (igual que hicimos con los Corredores en el nivel anterior).
- Un Array de Enemigos, los cuales simplemente son objetos heredados de EntidadMovible, pero con un atributo int 'tipo' también.
- Una OrthographicCamera, para la cámara.
- Los Botones para Android.
- Un Examen, que sera el objeto de fin de nivel, deberemos llegar hasta él para completar el nivel.
- Las texturas, TextureRegion [] y Animaciones.

- Las variables de estado de dimensión y stateTime.
- Y un Vector3 para controlar la pulsación en Android.

Por ultimo sí que vamos a añadir un objeto adicional, pero que realmente no será parte del juego, sino que usaremos para dibujar el área de cada objeto y comprobar que las colisiones se hacen correctamente.

Este objeto es un ShapeRender, y es precisamente eso, una forma de dibujar figuras geométricas, y utilizaremos para pintar los bordes de los objetos, aunque esto es solo de cara al programador, no hay que utilizarlo en el juego final.

A.8.3. Novedad en las animaciones

El contenido de la clase NivelPizarra no tiene ya ningún misterio, solo hay que seguir el mismo proceso de inicialización que hemos seguido hasta ahora: obtenemos las dimensiones, proporciones, establecemos la cámara, creamos el SpriteBatch y cargamos las texturas.

La única novedad que nos podemos encontrar es que para cada personaje crearemos dos animaciones, una hacia la derecha y otra hacia la izquierda, para que podamos dibujar a los enemigos en cualquier dirección y podamos girar a Turing. A la hora de crear las animaciones simplemente realizaremos el mismo proceso dos veces.

```
/*
* Preparamos el vector con las imagenes del movimiento de Turing
*/

anchura = texturaturingdcha.getWidth() / 5;
altura = texturaturingdcha.getHeight() / 5;

tmp = TextureRegion.split(texturaturingdcha, (int)anchura, (int)altura);
turingdchaFrames = new TextureRegion[5 * 5];
indice = 0;
for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {// aqui rellenamos el vector con cada
            imagen
                turingdchaFrames[indice++] = tmp[i][j];
        }
}

tmp = TextureRegion.split(texturaturingizq, (int)anchura, (int)altura);
turingizqFrames = new TextureRegion[5 * 5];
indice = 0;
for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {// aqui rellenamos el vector con cada
            imagen
                turingizqFrames[indice++] = tmp[i][j];
        }
}
```

```
anchura = proporcionAncho*anchura;
altura = proporcionAlto*altura;


// Creamos a Turing
turing = new TuringPizarra(juego, new Vector2(500, 300), anchura , altura ,
    200, 1);

turingdcha = new Animation(0.04f, turingdchaFrames);
turingizq = new Animation(0.04f, turingizqFrames);
```

Continuamos el conocido proceso de inicialización, creamos los botones, el examen, añadimos las plataformas y los enemigos, establecemos el procesador de entrada (aquí hemos creado la clase específica EntradaPizarra), creamos nuestro controlador y creamos el objeto ShapeRender.

En el método **render()** pasa lo mismo, seguimos el mismo esquema visto hasta ahora: Limpiamos la pantalla, hacemos que la cámara siga a Turing, indicamos el sistema de coordenadas de la cámara, calculamos el StateTime de Turing. Y aquí es donde viene el pequeño cambio: igual que en el constructor, según Turing este mirando hacia la derecha o hacia la izquierda, cargamos el frame de una animación o de la otra, ¡así de simple!

```
// Preparamos a Turing para ser dibujado (si esta mirando hacia la derecha
   o la izquierda)
stateTime = turing.getStateTime();

if (turing.getDireccion() == 1) {
        frameActual = turingdcha.getKeyFrame(stateTime, true); // el
            frameActual es el frame numero "stateTime" del vector de
            imagenes
} else {
        frameActual = turingizq.getKeyFrame(stateTime, true);
}
```

Por último dibujamos los botones de la manera que ya vimos en el nivel anterior, desproyectando la cámara sobre un Vector3 y como toque final, vamos a dibujar los bordes de los objetos para poder comprobar las colisiones.

Para ello iniciamos un nuevo proceso de dibujado, pero con el objeto ShapeRender **begin()-end()**. Establecemos el objeto a dibujar como un rectángulo, y simplemente dibujamos un rectángulo para cada objeto que queramos, utilizando su posición, anchura y altura:

```
sr.setProjectionMatrix(camara.combined);
sr.begin(ShapeType.Rectangle);
sr.rect(turing.getBordes().x, turing.getBordes().y, turing.getAnchura(),
    turing.getAltura());
IterPlataforma = plataformas.iterator();
while (IterPlataforma.hasNext()) {
        plataforma = IterPlataforma.next();
        sr.rect(plataforma.getBordes().x, plataforma.getBordes().y,
            plataforma.getAnchura(), plataforma.getAltura());
}

IterEnemigo = enemigos.iterator();
while (IterEnemigo.hasNext()) {
        enemigo = IterEnemigo.next();
        sr.rect(enemigo.getBordes().x, enemigo.getBordes().y, enemigo.
            getAnchura(), enemigo.getAltura());
}
sr.end();
```

Y para terminar actualizamos a Turing, el controlador, y los botones.

A.8.4. Acciones enemigas

Ahora vamos a ver como creamos a los enemigos. Para que haya algo de dificultad, haremos que se muevan en un bucle, por lo que deberán heredar de la clase Enemigo (que a su vez hereda de EntidadMovible), a la que ademas de sus atributos básicos añadiremos un int 'movimiento' para controlar el recorrido que hará el enemigo.

Controlar dicho movimiento ahora es tan fácil como aumentar la posicion en X (o en Y, según el enemigo) y a la vez aumentar un contador auxiliar que parta desde cero. Cuando ese contador llegue al máximo de la distancia que le hemos introducido con el atributo 'movimiento' solo tenemos que cambiar la dirección del enemigo y reiniciar el contador.

Por ejemplo para el enemigo 'Infinito', que se moverá de derecha a izquierda:

```
public void update() {
        if (aux == movimiento) {//Si llegamos al fin del movimiento,
                aux = 0;
                direccion = -direccion;// damos la vuelta
        }
        posicion.x = posicion.x + (direccion * (Gdx.graphics.getDeltaTime()
            * SPEED));
        aux++;

        bordes.x = posicion.x;
        bordes.y = posicion.y;
}
```

A.8.5. Física y colisiones

Vamos a explicar por fín como realizar la física del salto de Turing y cómo hacer que colisione con obstáculos y enemigos.

Los trataremos conjuntamente, ya que al ser ambos imprescindibles para determinar el movimiento que realizará Turing lo implementaremos todo dentro de la clase ColisionesPizarra, que al igual que el controlador del nivel anterior, tendrá un método **update()** en el que se harán las comprobaciones necesarias en cada frame del juego, ya que se ejecutara dentro del método render en la clase NivelPizarra.

Antes de nada debemos crear e inicializar los elementos que vamos a usar. La clase recibe el objeto NivelPizarra, por lo que de ahí podemos obtener a Turing, el Array de Plataformas, el Array de Enemigos, el de Flechas y el Examen de fin de nivel:

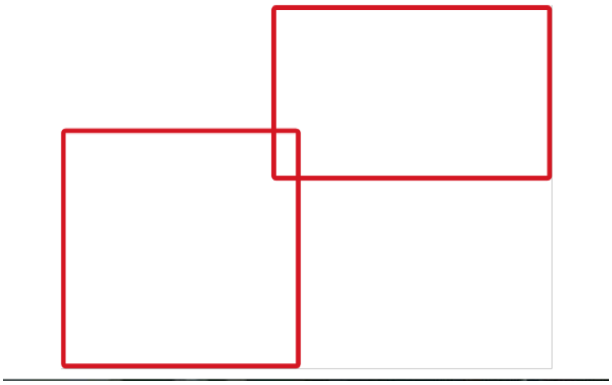
```
NivelPizarra juego;
TuringPizarra turing;
Array<Plataforma> plataformas;
Array<Enemigo> enemigos;
Examen examen;
float SPEED;

public ColisionesPizarra(NivelPizarra juego) {
        this.juego = juego;
        this.turing = juego.getTuring();
        this.plataformas = juego.getPlataformas();
        this.enemigos = juego.getEnemigos();
        this.examen = juego.getExamen();
        this.SPEED = turing.getSPEED();
}
```

Ahora tocaría explicar el método **update()** donde realizaríamos todo el movimiento y las colisiones, pero antes vamos a explicar las funciones auxiliares que usaremos para calcular las colisiones, y así cuando entremos en el método **update()** podremos centrarnos en el movimiento.

Para las colisiones al igual que antes usamos la función de la clase Rectangle **contains(int x, int y)**, ahora necesitaríamos algo parecido pero que en lugar de un punto reciba otro rectángulo. La función mas idónea para ello es **overlaps(Rectangle r)**, que al recibir otro rectángulo nos dice si se superponen o no.

¿Ya está? ¿Tan fácil? Si simplemente queremos saber si dos objetos han colisionado sí, pero pensemos en que necesitamos: nosotros queremos saber si por ejemplo Turing ha chocado con una plataforma a la izquierda, para que no pueda seguir avanzando a la izquierda. Con esta función tenemos un problema:



¿Por donde hemos chocado, por la derecha o por abajo? Esta función nos devuelve verdadero una vez que los elementos YA han colisionado, con lo cual si queremos hacer que Turing no pueda seguir avanzando a la izquierda si ha chocado con algo por ese lado no podemos hacerlo con esta función.

¿Cual es la solución?: Prevenir el movimiento, es decir, en lugar de comprobar en el momento actual si estamos colisionando con los objetos, comprobarlo en el instante siguiente. Si nos estamos moviendo a la izquierda, ver donde estará Turing en el instante siguiente, y si choca con algo no dejar que se mueva. Para ello vamos a utilizar una función auxiliar para cada posibilidad, dicha función recibirá un Rectangle (el borde de Turing) y el Array de Plataformas, para que compruebe si choca con alguna. Primero definimos una función general que sirva para la colisión en sí:

```
boolean Colisiona(Rectangle a, Rectangle b) {
        if (a.overlaps(b)) {
                return true;
        }
        return false;
}
```

Y ahora debemos comprobar la colisión para cada lado. Mostraremos el ejemplo de la izquierda. Recorremos todas las plataformas, tenemos dos Rectangles, 'a' será el objeto a comprobar que colisiona (Turing) y 'b' la plataforma. Ahora creamos un Rectangle auxiliar idéntico al 'a' y lo movemos un instante a la izquierda. Por ultimo comprobamos si aux y 'b' colisionan. Ya hemos comprobado si en el instante siguiente van a colisionar Turing y la plataforma en cuestión.

Hacemos lo mismo para cada lado.

```
boolean ColisionIzquierda(Rectangle a, Array<Plataforma> plataformas) {//
   Hay colision a la izquierda del personaje
        terator<Plataforma> iter = plataformas.iterator();
        while (iter.hasNext()) {
                Entidad e = iter.next();
                Rectangle b = e.getBordes();
                Rectangle aux = new Rectangle(a.x, a.y, a.width, a.height);
                aux.x = aux.x + (-1 * (Gdx.graphics.getDeltaTime() * SPEED)
                    );
                if (Colisiona(aux, b)) {
                        return true;
                }
        }
        return false;
}
```

Hemos terminado con las colisiones de plataformas, pero aprovechando que hemos visto como prevenir el movimiento por cada lado, vamos a aprovecharlo para crear una nueva función auxiliar que se encargue de controlar la colisión con los enemigos. Esta función, llamada **MataEnemigos** recibirá al igual que las anteriores, un Rectangle y un Array, pero esta vez de enemigos. En ella recorreremos todos los enemigos y comprobaremos si han colisionado con Turing. Si es así el resultado es simple, Turing será derrotado.

¿Pero como hacemos para vencer a nuestros enemigos? Como en cualquier juego típico de plataformas, deberemos saltar encima de ellos para eliminarlos, por lo que vamos a crear un Rectangulo auxiliar igual a los bordes de Turing y le restamos en un frame la posición en Y, es decir, el siguiente instante si estuviera cayendo. Si ahora colisiona con el enemigo, entonces será este el que eliminaremos del Array. De esta forma diferenciamos la forma de colisionar con los enemigos. Solo tenemos que ejecutar esta función dentro de **update()** y siempre comprobaremos si hemos sido derrotados o por el contrario hemos eliminado algún enemigo.

La ultima colisión que comprobaremos será la del examen de fin de nivel. Como es muy simple la haremos directamente en el método **update()** al final. Simplemente usamos la función básica **Colisiona(Rectangle a, Rectangle b)** y si Turing y el Examen han colisionan entonces habremos finalizado el nivel.

Ahora ya podemos volver al método **update()** y empezar con el movimiento:

Lo primero que vamos a hacer es comprobar hacia qué lado nos movemos y actuar en consecuencia, gracias a las funciones que acabamos de definir, ademas de actualizar el StateTime, necesario para dibujar la animación.

Si la velocidad de Turing es 0, no haremos nada, solo reiniciar el StateTime.

```
if (turing.getVelocidad() == 0) {//si esta parado, el estado de la
    animacion se reinicia
        turing.setStateTime(0f);
}
```

Si nos movemos hacia la izquierda, NO hay colisión a la izquierda y no hemos llegado al límite del nivel, pues nos movemos a la izquierda:

```
else if (turing.getVelocidad() == -1 && !ColisionIzquierda(turing.getBordes
    (), plataformas) && turing.getPosicion().x >= juego.getMarcoPizarra()) {
        turing.getPosicion().x = turing.getPosicion().x + (turing.
            getVelocidad() * (Gdx.graphics.getDeltaTime() * SPEED));
        // a la posicion en x le sumo la velocidad que hemos determinado (
            SPEED) por DeltaTime, que es el tiempo que pasa entre un frame y
             otro, para que el movimiento sea igual independientemente de la
             tasa de refresco, y lo multiplicamos por -1, ya que nos movemos
             a la izquierda

        turing.setStateTime(turing.getStateTime() + Gdx.graphics.
            getDeltaTime()); //como nos estamos moviendo modificamos el
            stateTime, que servira para acceder a un nuevo frame en la
            animacion
}
```

Para la derecha será exactamente igual pero con la función ColisionDerecha y moviéndonos a la derecha.

Y ahora entramos en la parte de física, el salto. Para realizar el salto lo único que hay que hacer realmente es tener una serie de variables de estado que nos permitan saber en que momento nos encontramos: si hemos saltado, si hemos pulsado el botón de salto ya, si hemos llegado al suelo, etc. Luego solo queda sincronizar y controlar bien estas variables para manejar el salto a nuestro antojo. Las variables que usaremos serán las que hemos incluido en TuringPizarra:

- boolean saltando: para saber si estamos actualmente saltando.
- boolean enelsuelo: para saber si estamos en el suelo o sobre alguna plataforma.
- float alturaSalto: Para saber la altura máxima de nuestro salto.
- float salto: Es la distancia recorrida al saltar.

Ahora solo queda controlarlas:

- Si aterrizamos (chocamos por abajo) en una plataforma o llegamos al suelo, enelsuelo = true, si no, es que estamos en el aire aun, por lo que enelsuelo = false.
- Si llegamos a la altura máxima, chocamos por arriba con alguna plataforma, o llegamos al techo, dejamos de saltar para empezar a caer, así que saltando = false.

Esas son las comprobaciones, ahora el movimiento en sí:

- Si estamos saltando, debemos incrementar nuestra posición en Y, ademas de aumentar la distancia de la variable salto, para saber cuando llegamos al máximo.
- Por el contrario, si no estamos saltando, no chocamos con nada por debajo ni llegamos al suelo, caemos, es decir, reducimos nuestra posición en Y.

Estas son todas las comprobaciones durante el juego. Si están bien hechas no habrá ningún error, es decir, no saltaremos indefinidamente ni sobrepasaremos el techo, el suelo o alguna plataforma.

Pero aun no está listo el salto, faltan ciertas comprobaciones, y son las realizadas al pulsar los botones, las que activarán el salto. Estas se encuentran en la clase EntradaPizarra que al igual que en el nivel anterior, implementa a InputProcessor.

Aprovechamos que hemos llegado a esta parte para explicar como activar el movimiento a los lados. Pero es tan simple como que al pulsar la tecla deseada ('A' para la izquierda y 'D' para la derecha en este caso) establezcamos la velocidad y la dirección de Turing a -1 o 1 respectivamente.

Para el salto, pues igual, si pulsamos la tecla 'W' lo activamos. Esto solo lo podemos hacer si no estamos ya en el aire, por lo que si estamos en el suelo: enelsuelo = false, saltando = true y reiniciamos el medidor del salto.

```
public boolean keyDown(int keycode) {
        switch (keycode) {
        case Keys.A:
                juego.getTuring().setVelocidad(-1);
                juego.getTuring().setDireccion(-1);
                break;
        case Keys.D:
                juego.getTuring().setVelocidad(1);
                juego.getTuring().setDireccion(1);
                break;
        case Keys.W:
                if (juego.getTuring().isEnelsuelo()) {
                        juego.getTuring().setEnelsuelo(false);
                        juego.getTuring().setSaltando(true);
                        juego.getTuring().setSalto(0);
                }
                break;
        }
        return true;
}
```

Y con esto sí que estarían listas la física y colisiones del nivel. Ya sabemos realizar de forma manual un salto básico compuesto por variables de estado y prevenir colisiones según la dirección.

A.9. Nivel '1939-43: WW2. A por la Bombe': Proyectiles, munición, vida y objetos coleccionables

A.9.1. Ambientación y objetivo

Este nivel se sitúa cronológicamente antes del nivel 'Descifrando el Enigma', y ocurre también durante la 2º Guerra Mundial. Turing por fin ha conseguido completar su máquina Bombe, que le permitirá descifrar los códigos encriptados de la máquina Enigma de los Alemanes, pero un ataque enemigo ha divido su máquina en pedazos, por lo que debe salir al campo de batalla y recuperar todos los trozos!

Ya tenemos los conocimientos básicos referentes a la creación de un videojuego. Ahora solo queda combinarlos para crear algo mas grande. Así que en este nivel solo vamos a añadir detalles a lo usado en el nivel anterior. Será también un nivel de plataformas de desplazamiento lateral, con física y colisiones pero añadiremos elementos extra,que serán los proyectiles, de los enemigos, que tendremos que definir para que sigan un patrón y serán infinitos, y los propios, que podremos disparar a nuestro antojo y serán limitados, pero que podremos rellenar usando paquetes de munición.
Ademas añadiremos objetos coleccionables (que serán los trozos de la máquina Bombe) que deberemos recoger para completar el nivel en lugar de tener que esperar un tiempo o tener que llegar al final del nivel como hasta ahora.

A.9.2. Elementos del nivel

En este caso ya conocemos todos los elementos del nivel, pero aun así daremos un rápido repaso para que no se olvide nada:

- Clase principal TuringAdventure, para que todo quede interconectado.
- Nuestro controlador de movimiento y colisiones, ColisionesGuerra.
- El SpriteBatch para dibujar todos los elementos por pantalla.
- Nuestro protagonista: TuringGuerra.
- Arrays de Plataformas, Enemigos, Municion, Balas, y FragmentoBombe. Serán los Arrays en los que introduciremos los elementos del juego. Los primeros ya los conocemos, la Munición y los FragmentoBombe son simplemente objetos heredados de Entidad, ya que solo nos hace falta instanciarlos y dibujarlos para poder recogerlos. Las balas sí serán un objeto que hereda de Proyectil, que a su vez hereda de EntidadMovible, para que al crearlas con su dirección y velocidad se muevan.
- Usaremos de nuevo un BitmapFont para usarlo como visor.
- Una OrthographicCamera que será como siempre nuestra cámara.
- Los Botones para el manejo del juego en Android.

– Las Texturas del juego, TextureRegion[] para las animaciones y objetos Animation.

– Un Vector3 para la pulsación en Android.

– Las constantes de dimensiones, proporciones, suelo, fin de nivel y StateTime.

Y por supuesto nuestro controlador de Entrada, que ya sabemos que no hace falta crearlo, sino que lo instanciaremos dentro del constructor como sustituto del InputProcessor por defecto:

```
Gdx.input.setInputProcessor(new EntradaGuerra(this));/
```

A.9.3. Cambios en personajes

En la clase principal no tendremos ningún cambio notable, todo continuará tal y como hemos visto en el nivel anterior. Primero inicializamos todas las variables, dimensiones, proporciones, cámara, SpriteBatch, Texturas y animaciones. Añadimos los elementos del juego a sus respectivos Arrays y creamos el controlador.

Donde veremos algún pequeño cambio es en las entidades de Turing y los enemigos, pues ambos tienen que manejar a su manera el control de los disparos.

TuringGuerra recibirá el nivel como parámetro para poder acceder a las balas, y añadirlas cuando dispare, al igual que los enemigos de este nivel, los Soldados. Turing ademas tendrá un contador de vida, un tamaño máximo para éstas, y otro contador para saber cuantos FragmentosBombe ha recogido, así como de munición.

Ademas, tendremos una función nueva, que llamaremos al disparar para crear nuevas balas. Esta función simplemente comprueba si aún tenemos munición, y si es así añade una bala en la dirección que estemos mirando y resta uno a la munición. Es importante hacer que al crear la bala ésta se cree fuera de los bordes de Turing, ya que como haremos las colisiones generales para todas las balas, si chocamos con nuestra propia bala nos dañará.

```
public void addBala() { // DISPARO DE BALAS
        if(MunicionTuring > 0){
                if(direccion == 1){
                        balas.add(new Bala(new Vector2((bordes.x + bordes.
                            width), CentroY(bordes) - proporcionAlto*20),
                            proporcionAncho*20, proporcionAlto*5, 300,
                            direccion));
                }
                else{
                        balas.add(new Bala(new Vector2((bordes.x -
                            proporcionAncho*20), CentroY(bordes) -
                            proporcionAlto*20), proporcionAncho*20,
                            proporcionAlto*5, 300, direccion));
                }
```

```
                MunicionTuring--;
        }
}
```

Los Soldados incluirán también un contador de vida, y un objeto de tipo NivelGuerra para acceder a las Balas y a Turing, ya que tendrán una pequeña interacción con el.

Haremos que los enemigos miren siempre a Turing, es decir, que si éste se encuentra a su izquierda se giren hacia la izquierda, y viceversa. Aquí debemos controlar las balas de un modo distinto, y ademas debemos sincronizarlas con la animación de disparo, por lo que incluimos también un contador de tiempo.

Dispararemos cada dos segundos, y para que la animación no sea continua, sino que se tome un segundo entre disparo y disparo (recordemos que la animación dura un segundo), por lo que si queremos que dispare cada dos, solo tenemos que esperar un segundo entre animación y animación:

```
public void update() {
        if(juego.getTuring().getPosicion().x < posicion.x){
                direccion = -1;
        }
        else{
                direccion = 1;
        }

        if (TimeUtils.nanoTime() - ultimo_disparo < 1000000000) { // Si ha
            pasado mas de 1 segundo
                        stateTime += Gdx.graphics.getDeltaTime();
        }

        if (TimeUtils.nanoTime() - ultimo_disparo > 2000000000) {

                if(direccion == 1){
                        balas.add(new Bala(new Vector2((bordes.x + bordes.
                            width), CentroY(bordes)), proporcionAncho*20,
                            proporcionAlto*5, 300, direccion));
                }
                else{
                        balas.add(new Bala(new Vector2((bordes.x -
                            proporcionAlto*20), CentroY(bordes)),
                            proporcionAncho*20, proporcionAlto*5, 300,
                            direccion));
                }

                ultimo_disparo = TimeUtils.nanoTime();
        }

        bordes.x = posicion.x;
        bordes.y = posicion.y;
}
```

De esta forma tendremos sincronizados animación y disparo.

A.9.4. Controlador: Disparos, recoger munición y fragmentos

La primera parte del controlador será la misma que hemos visto hasta ahora, calcularemos el movimiento y la física del salto de la misma forma. Utilizaremos también las mismas funciones para calcular las colisiones en cada lado.

El único cambio es que aquí no podremos eliminar enemigos al saltarles, solo podremos hacerlo con disparos, por lo que sustituiremos la función **MataEnemigos** por la función **ChocarEnemigos**, que hará exactamente lo mismo que la antigua, menos la comprobación de la caída, es decir, si chocamos con algún enemigo, Turing perderá todas sus vidas:

```
boolean ChocarEnemigos(Rectangle a, Array<Enemigo> enemigos) { // Hay
    colision abajo del personaje
        Iterator<Enemigo> iter = enemigos.iterator();
        while (iter.hasNext()) {
                Entidad e = iter.next();
                Rectangle b = e.getBordes();
                Rectangle aux = new Rectangle(a.x, a.y, a.width, a.height);
                if (Colisiona(aux, b)) {
                        // muere turing
                        turing.RecibeDanio(turing.getVida());
                        return false;
                }
        }
        return true;
}
```

Como añadidos al controlador, debemos calcular las colisiones con los paquetes de munición, con los Fragmentos de la Máquina Bombe y con las Balas.
Los dos primeros son muy simples, solo recorremos el Array de Municion y el Array de Fragmentos, y si colisionamos con ellos eliminamos la Municion o el Fragmento del Array y se lo añadimos a Turing:

```
Iterator<Municion> IterMunicion = municiones.iterator();
while(IterMunicion.hasNext()){
        Municion m = IterMunicion.next();
        if(Colisiona(m.getBordes(), turing.getBordes())){
                IterMunicion.remove();
                turing.RecogeMunicion();
        }
}

Iterator<FragmentoBombe> IterFragmento = maquinaBombe.iterator();
while(IterFragmento.hasNext()){
        FragmentoBombe frag = IterFragmento.next();
        if(Colisiona(frag.getBordes(), turing.getBordes())){
                IterFragmento.remove();
                turing.RecogerFragmentoBombe();
        }
}
```

La colisión con las balas no es mucho mas complicada, primero debemos recorrer todas las balas que hay en juego y debemos tener en cuenta tres cosas, las plataformas, los enemigos y Turing.

Para ello dentro del while que recorre las balas, debemos recorrer a su vez todas las plataformas, y simplemente si ambas chocan eliminamos la bala. Recorremos también los enemigos, y si chocan eliminamos la bala y le quitamos una vida al enemigo, y ya puestos aprovechamos para comprobar si la vida del enemigo es 0 para eliminarlo del juego. Por ultimo comprobamos si la bala y Turing chocan, y si es así perdemos una vida.

```
// COLISION DE BALAS

Iterator<Bala> IterBala = balas.iterator();
while (IterBala.hasNext()) {
        Bala t = IterBala.next();
        Iterator<Plataforma> iterPlataforma = plataformas.iterator();
        while (iterPlataforma.hasNext()) {
                Plataforma p = iterPlataforma.next();
                if (Colisiona(t.getBordes(), p.getBordes())) {//Si alguna
                    bala choca con una plataforma...
                        IterBala.remove();//...se destruye la bala
                }
        }
        Iterator<Enemigo> IterEnemigo = enemigos.iterator();
        while (IterEnemigo.hasNext()) {
                Enemigo e = IterEnemigo.next();
                if (Colisiona(t.getBordes(), e.getBordes())) { //Si choca
                    con un enemigo le hace un punto de danio
                        e.Daniar(1);
                        if(e.getVida() <= 0){//Y si el enemigo tiene 0 de
                            vida...
                                IterEnemigo.remove();//...muere
                        }
                        IterBala.remove();// Y por supuesto se destruye la
                            bala, muera o no el enemigo
                }
        }

        if (Colisiona(t.getBordes(), turing.getBordes())) {
                turing.RecibeDanio(1);
                IterBala.remove();
        }
}
```

Como último detalle añadir que para realizar el disparo solo tenemos que añadir un estado mas en la detección de teclas del controlador de entrada EntradaGuerra. Al pulsar la tecla 'J', hacemos que se ejecute la función **addBala()** que hemos definido en TuringGuerra:

```
case Keys.J:
        juego.getTuring().addBala();
        break;
}
```

Y lo mismo al pulsar el botón correspondiente en Android.

Con esto hemos terminado el control de las colisiones de este nivel.

A.9.5. Tipos de Vida

Ya hemos visto como reducir la vida tanto de Turing como de los enemigos. Ahora vamos a ver como mostrarla por pantalla.
Para dar variedad vamos a ver dos métodos distintos:
Para Turing vamos a dibujar la vida en la esquina superior izquierda de la pantalla. Será muy sencillo, tendremos 5 imágenes, una para cada estado de la vida, y según esté dibujaremos una u otra. Solo tendremos que tener en cuenta que queremos ver la vida durante todo el nivel, por lo que para dibujarla en función de la cámara seguiremos el mismo método que para los botones: desproyectamos la cámara en un Vector3 y lo utilizamos para dibujar cada imagen:

```
camara.unproject(posicion.set(0, texturaVida5.getHeight(), 1));

int opc = turing.getVida();
switch(opc){
case 1:
        batch.draw(texturaVida1, posicion.x, posicion.y, texturaVida1.
            getWidth(), texturaVida1.getHeight());
        break;
case 2:
        batch.draw(texturaVida2, posicion.x, posicion.y, texturaVida2.
            getWidth(), texturaVida2.getHeight());
        break;
case 3:
        batch.draw(texturaVida3, posicion.x, posicion.y, texturaVida3.
            getWidth(), texturaVida3.getHeight());
        break;
case 4:
        batch.draw(texturaVida4, posicion.x, posicion.y, texturaVida4.
            getWidth(), texturaVida4.getHeight());
        break;
case 5:
        batch.draw(texturaVida5, posicion.x, posicion.y, texturaVida5.
            getWidth(), texturaVida5.getHeight());
                break;
default: break;
}
```

Para la vida enemiga en cambio vamos a utilizar una barra de vida proporcional, es decir, tenemos la imagen básica de la barra de vida, y según la vida restante del enemigo, multiplicamos su longitud. Para ello debemos dibujarla en el recorrido de los enemigos, para pintarla siempre encima de su personaje correspondiente:

```
// Dibujamos los enemigos
IterEnemigo = enemigos.iterator();
while (IterEnemigo.hasNext()) {
        enemigo = IterEnemigo.next();
        enemigo.update();
        if (enemigo.getTipo() == 1) {// Si es un soldado
                stateTimeSoldado1 = enemigo.getStateTime();
                if (enemigo.getDireccion() == -1) {
                        frameActualSoldado1 = soldado1Izq.getKeyFrame(
                            stateTimeSoldado1, true);
                } else if (enemigo.getDireccion() == 1) {
                        frameActualSoldado1 = soldado1Dcha.getKeyFrame(
                            stateTimeSoldado1, true);
                }
                batch.draw(frameActualSoldado1, enemigo.getPosicion().x,
                    enemigo.getPosicion().y, enemigo.getAnchura(), enemigo.
                    getAltura());
        }

        //Y dibujamos su vida
        batch.draw(texturaVidaEnemiga, enemigo.getPosicion().x, enemigo.
            getPosicion().y + enemigo.getAltura(), proporcionAncho*
            texturaVidaEnemiga.getWidth() * enemigo.getVida(),
            proporcionAlto*texturaVidaEnemiga.getHeight());

}
```

Así podemos dibujar la vida de dos formas distintas.

Por último vamos a añadir las condiciones de fin de nivel, que sería el último punto a comprobar en el **render()**, debemos ver si el número de vidas de Turing es 0, para en ese caso acabar el nivel y por el contrario, si hemos recogido los Fragmentos de la Máquina Bombe necesarios, para completarlo.

```
//MUERTE DE TURING
if(turing.getVida() <= 0){
        juego.setScreen(new MenuPrincipal(juego));
}

//FIN DE NIVEL
if(turing.getFragmentosBombe() == 1){
        juego.setScreen(new MenuPrincipal(juego));
}
```

Listo. Con esto hemos completado todos los niveles del juego, ahora aplicando las técnicas aprendidas y llevándolas a un nivel superior, podemos añadir tantos elementos como queramos al juego así como expandirlo tanto como queramos.

A.10. Menús, Pantallas Intermedias y Guardar Datos

Ya hemos visto toda la base de la parte jugable de un videojuego. Con estos conocimientos básicos podemos crear casi cualquier cosa, aunque por supuesto hay muchas herramientas que nos facilitarán el trabajo, como motores de física, gestores de partículas y luces, etc, pero al menos sabemos lo principal.

Ahora hablaremos de algunos extras y detalles que podemos añadirle al juego, aunque no será nada complicado comparado con lo visto hasta ahora.

Primero veremos como usar las pantallas y botones para hacer menús y cambiar entre pantallas. Ya hemos visto como establecer pantallas, pero para que quede claro veremos como combinarlo con los botones y crear distintas pantallas de menús.

Veremos también como añadir música y sonidos al juego, que es muy sencillo ya que LibGDX tiene clases que prácticamente lo hacen por nosotros.

Ademas, nos habremos dado cuenta que en nuestro juego falta algo: alguna forma de guardar datos, para que cuando volvamos a entrar en el juego quede algún registro de nuestra anterior partida. Esto puede ser alguna partida guardada (guardando la vida, objetos conseguidos, etc), configuración del sistema, por si tenemos opciones a cambiar la resolución, gráficos, música o cualquier ora cosa, o en nuestro caso, guardar estadísticas y logros de la partida.

Y por último, añadiremos la forma de crear usuarios, para que estos logros no sean una característica propia del juego, sino que se independicen según el usuario que esté jugando.

A.10.1. Menús y Pantallas Intermedias

Como ya vimos al principio, la clase principal del juego es TuringsAdventure, que es la clase que lanza el main (o MainActivity para Android), en el constructor de esta clase lo que haremos será llamar a nuestro MenuPrincipal que para recordar, es una clase que implementa a Screen, por lo que podremos dibujar en ella y tratarla como a una pantalla:

```
setScreen(new MenuPrincipal(this)); // Coloca la pantalla actual, se llama
    desde cualquier pantalla anterior y se llama a Screen.show desde la
    nueva pantalla
```

Ya en MenuPrincipal (o en cualquier pantalla de menú) actuaremos como de costumbre: definimos los elementos que vamos a usar, que principalmente serán las texturas del fondo y de los botones, las dimensiones y las proporciones, así como la cámara y el SpriteBatch.

En el constructor nos encargaremos de inicializar estos elementos y en el método **render()** dibujamos el fondo y los botones que vayamos a utilizar, y es justo al terminar esta acción, después del **batch.end**, cuando tendremos que recoger las acciones a realizar al pulsar los botones.
Esto es tan simple como usar el módulo de entrada de LibGDX para detectar si pulsamos en la pantalla (aquí no hace falta controlador, es muy simple así que lo hacemos directamente) y si los bordes del botón contienen ese punto, lanzamos el submenu correspondiente:

```
if(Gdx.input.justTouched() && botonSeleccionNivel.getBordes().contains(Gdx.
    input.getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){

        juego.setScreen(new SeleccionNivel(juego));// Y si pulsamos en los
            demas botones, llamamos a sus respectivas pantallas
}

if(Gdx.input.justTouched() && botonPantallaLogros.getBordes().contains(Gdx.
    input.getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){

        juego.setScreen(new PantallaLogros(juego));
}

if(Gdx.input.justTouched() && botonSalir.getBordes().contains(Gdx.input.
    getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){
        Gdx.app.exit();
}
```

Una pequeña explicación sobre **justTouched()**, hay otro método muy parecido llamado **isTouched()**, pero ese nos devuelve verdadero durante TODO el tiempo que la pantalla este pulsada, por lo que si al cambiar de pantalla hay un botón en el mismo sitio donde estamos pulsando el render procesa tan rápido que también será pulsado.

En cambio **justTouched** nos devuelve solo si en una pulsación aislada, por lo que es la más adecuada para usar con botones.

En principio para pasar a las pantallas de niveles es exactamente lo mismo. Pero queda muy soso simplemente empezar o finalizar el nivel sin más como hemos hecho hasta ahora. Así que crearemos unas pantallas intermedias para cada vez que entremos en un nivel o cuando terminemos y lo completemos o seamos derrotados.

Esta clase se llamará PantallaIntermedia, y recibirá la clase TuringAdventure (como siempre, para tener todo el juego unificado), un indicador de la pantalla a la que queremos cambiar (-1 = GameOver(SeleccionNivel), 0 = Menu, 1 = NivelPizarra, 2 = NivelGuerra, 3 = NivelDefensa, 4 = NivelCarrera), y la imágen de fondo de la pantalla intermedia.

La única acción que realizaremos en el **render() de** esta clase será dibujar el fondo que le pasemos, y mediante un switch, comprobar la pantalla a la que queremos que se cambie y actuar en consecuencia:

```
public void render(float delta) {
        Gdx.gl.glClearColor(0, 0, 0, 1);
        Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

        batch.begin();
        batch.draw(texturaFondo, 0, 0, anchura, altura); //Dibujamos el
            SpriteBatch
        batch.end();

        if(Gdx.input.justTouched()){//Si tocamos la pantalla
        switch(pantalla){
        case -1: juego.setScreen(new SeleccionNivel(juego));
                 break;
        case 0: juego.setScreen(new MenuPrincipal(juego));
                 break;
        case 1: juego.setScreen(new NivelPizarra(juego));
                 break;
        case 2: juego.setScreen(new NivelGuerra(juego));
                 break;
        case 3: juego.setScreen(new NivelDefensa(juego));
                 break;
        case 4: juego.setScreen(new NivelCarrera(juego));
                 break;
        }

}
```

Ahora solo queda utilizar ésta clase a la hora de pulsar los botones de inicio de nivel o al terminar un nivel (pasandole un fondo de victoria o de derrota según el caso).

```
if(Gdx.input.justTouched() && botonPizarra.getBordes().contains(Gdx.input.
    getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){

        juego.setScreen(new PantallaIntermedia(juego, 1,
            texturaInicioPizarra));
}

if(Gdx.input.justTouched() && botonGuerra.getBordes().contains(Gdx.input.
    getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){

        juego.setScreen(new PantallaIntermedia(juego, 2,
            texturaInicioGuerra));
}

if(Gdx.input.justTouched() && botonDefensa.getBordes().contains(Gdx.input.
    getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){

        juego.setScreen(new PantallaIntermedia(juego, 3,
            texturaInicioDefensa));
}

if(Gdx.input.justTouched() && botonCarrera.getBordes().contains(Gdx.input.
    getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){

        juego.setScreen(new PantallaIntermedia(juego, 4,
            texturaInicioCarrera));
}

if(Gdx.input.justTouched() && botonVolver.getBordes().contains(Gdx.input.
    getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){

        juego.setScreen(new MenuPrincipal(juego));
}
```

.
Al completar un nivel:

```
juego.setScreen(new PantallaIntermedia(juego, 0, texturaNivelCompletado));
    // Volvemos a la pantalla principal
```

.
Y al ser derrotados:

```
juego.setScreen(new PantallaIntermedia(juego, -1, texturaGameOver ));//
    Volvemos a la pantalla de seleccion de nivel
```

Con esto tendríamos listas todas las pantallas de menús.

A.10.2. Sonidos y Música

Establecer la música y los sonidos es una de las tareas mas sencillas, ya que gracias al módulo de audio de LibGDX y a las clases que nos proporciona podemos instanciarlos y controlarlos de una manera muy simple.

Para los sonidos usamos la clase Sound y para la música la clase Music. Para inicianizarlos accedemos al modulo de audio y creamos el nuevo sonido o música, pasandole el archivo de audio donde lo tengamos almacenado:

```
public Sound salto = Gdx.audio.newSound(Gdx.files.internal("data/Sonidos/
    Salto.wav"));
```

```
public Music musica = Gdx.audio.newMusic(Gdx.files.internal("data/Sonidos/
    MusicaNivelPizarra.mp3"));
```

Ahora para reproducirlos solo debemos utilizar la función **play()**. Pero con una pequeña diferencia: los sonidos debemos reproducirlos en el momento en que sea necesario, en este caso 'salto' deberá reproducirse en el momento que comenzamos a saltar, es decir, en el controlador de entrada, cuando pulsamos la tecla 'W' o el boton de salto en Android:

```
public boolean keyDown(int keycode) {
        switch (keycode) {
        case Keys.A:
                juego.getTuring().setVelocidad(-1);
                juego.getTuring().setDireccion(-1);
                break;
        case Keys.D:
                juego.getTuring().setVelocidad(1);
                juego.getTuring().setDireccion(1);
                break;
        case Keys.W:
                if (juego.getTuring().isEnelsuelo()) {
                        juego.getTuring().setEnelsuelo(false);
                        juego.getTuring().setSaltando(true);
                        juego.getTuring().setSalto(0);
                        juego.salto.play();
                }
                break;
        }
        return true;
}
```

En cambio la música se reproducirá durante todo el nivel, por lo que debemos comenzar a reproducirla en el constructor del nivel y decirle que se repita cuando acabe. Esto lo hacemos mediante el método **setLooping()**:

```
musica.setLooping(true);
musica.play();
```

A.10.3. Logros

Para guardar los logros y estadísticas usaremos una herramienta de LibGDX que nos permitirá guardar localmente preferencias en forma de variables, es la manera mas simple y mas básica, pero servira mas que de sobra para nuestro propósito.

Esta herramienta sera la clase Preferences, que básicamente creará un archivo en nuestro sistema en el que crearemos pares de "Nombre-Valor"que se almacenarán permanentemente, podremos sobreescribir dichos valores dado un nombre, añadir nuevos pares, y leer el valor dado un nombre.

De esta forma, en nuestra clase principal TuringAdventure, añadimos una variable Preferences logros, que van a estar presentes durante todo el juego, ya que esta clase se pasa como parámetro a todas las pantallas y actúa como conector de todo el juego, por lo que podemos modificar en cualquier lugar nuestros logros.

Esta variable la inicializamos en el método **create()**. Esto lo hacemos accediendo al módulo de aplicación de LibGDX y cargando el archivo de preferencias que queramos. No es necesario incluir la ruta, ya que según el dispositivo, LibGDX crea una carpeta oculta fija en el sistema para almacenar estos archivos. Tampoco es necesario comprobar si existe, pues si es así se cargará, y sino se creara uno nuevo vacío con ese nombre:

```
logros = Gdx.app.getPreferences("logros-Invitado");//Cargamos el archivo de
    preferencias (Si no existe lo crea)
```

Ahora tenemos que añadir logros. Vamos a ver un ejemplo, en el caso de que completemos el Nivel 'Avanzando en Cambridge'.

Como ya hemos visto, las variables internas de las preferencias no tienen porque existir, al utilizarlas por primera vez LibGDX las crea si no existen, por lo que debemos tener muy claro la lista de logros y preferencias que vamos a necesitar, darle un nombre único a cada una, y asignarla o leerla cuando sea necesario.

En este caso, para saber si hemos completado el nivel, necesitamos una variable boolean que se ponga a true cuando lleguemos al examen de fin de nivel. Para ello entramos en el controlador del nivel (ColisionesPizarra), que es donde realizamos la colisión con el examen. En esta clase tenemos una variable NivelPizarra 'juego' para acceder a los elementos del nivel, y en en nivel (como en cada Screen del juego) tenemos una variable de tipo TuringAdventure, donde tendremos la variable global de preferencias (logros). Por lo que simplemente accediendo a ella y modificándola haremos que el logro esté conseguido:

```
//COLISION CON EL EXAMEN (Fin de nivel)
if(Colisiona(turing.getBordes(), examen.getBordes())){
        juego.getJuego().getLogros().putBoolean("Completa-Nivel-Pizarra",
            true);//Ponemos el logro a verdadero
        juego.getJuego().getLogros().flush();// Necesario para asegurarnos
            que la escritura de preferencias es correcta
        juego.getJuego().setScreen(new MenuPrincipal(juego.getJuego()));//
            Y volvemos a la pantalla principal
}
```

Debemos recordar realizar un **flush()** a la variable logros siempre que modifiquemos algún dato para asegurarnos de que se modifique correctamente.

Como vemos, el método de introducción de datos es específico para cada tipo, **putBoolean()** para datos booleanos, **putInteger()** para enteros, y asi sucesivamente. Es el único inconveniente de este método, que solo permite guardar variables básicas, pero como ya hemos dicho, para nuestro propósito es mas que suficiente.

¿Como lo hacemos ahora para mostrar los logros? Pues el método inverso: esta vez en el menú PantallaLogros. Simplemente tenemos una imagen para cada logro, y una imagen que sea la que indique si lo hemos desbloqueado.

Lo que hacemos es comprobar primero si tenemos desbloqueado el logro, y si es así dibujamos la marca, seguida de la imagen del logro en si, que dibujaremos aunque no lo tengamos desbloqueado, para saber qué logros podemos conseguir:

```
Preferences logros = juego.getLogros();
.
.
.
if(logros.getBoolean("Completa-Nivel-Pizarra")){ //Si tenemos el logro
    desbloqueado (verdadero) dibujamos la medalla
        batch.draw(texturaLogro, marcoPizarra, proporcionAlto*450);
}
batch.draw(texturaLogroNivel1, marcoPizarra + texturaLogro.getWidth();,
    proporcionAlto*450);
```

Listo, solo tenemos que modificar cada logro donde corresponda y dibujarlo en la pantalla de logros en el caso de que esté desbloqueado.

Como ya hemos comentado es muy importante tener un listado de los logros que queremos, ya que no vamos a usar un Array ni ninguna forma rápida de recorrer los logros, sino que los dibujaremos y modificaremos manualmente.

A.10.4. Usuarios

Ahora toca el momento de crear usuarios y que los logros se modifiquen sólo para el usuario en cuestión. Pero entonces habrá que cambiar la forma de acceder y leer los logros, separandolos por usuarios.... es lo lógico pensar.

Pues no, ya que para los que se hayan dado cuenta, la forma de modificar logros según el usuario es mucho más sencilla de lo que parece.

Recordemos que al cargar el archivo de preferencias para los logros hacíamos esto:

```
logros = Gdx.app.getPreferences("logros-Invitado");//Cargamos el archivo de
    preferencias (Si no existe lo crea)
```

Y tal y como hemos dicho, sirve tanto para cargar uno existente como para crearlo. Y lo mismo para cada atributo interno, se crea o carga si existe o no. Entonces... si mas adelante volvemos a cargar en la misma variable logros otro archivo de preferencias (por ejemplo 'Usuario1'), cada vez que muestre o modifique un logro desde dicha variable lo estará haciendo para ese usuario, por lo que todo lo explicado sigue valiendo perfectamente, lo único que hay que hacer es cargar o crear ese nuevo archivo de preferencias.

Ademas, añadiremos un atributo 'Nombre' a cada archivo con el nombre del usuario, para poder mostrarlo por pantalla y saber que usuario esta activo cada vez.

La primera parte sigue siendo igual. Al iniciar el juego crearemos un usuario general, que valga para aquel que no quiera tener su propio usuario, será el usuario invitado.

Y tendremos un String con el nombre de usuario así como una variable booleana que nos indique cuando cambiemos de usuario, y simplemente controlaremos en el render de la clase principal TuringAdventure (recordamos que es de tipo Game, que como cualquier clase base de LibGDX tiene sus métodos básicos) si esta variable es true, cargamos de nuevo el archivo de preferencias con ese usuario y añadimos su nombre:

```
public void create() {
        logros = Gdx.app.getPreferences("logros-Invitado");//Cargamos el
            archivo de preferencias (Si no existe lo crea)
        logros.putString("Nombre-Usuario", "Invitado"); // Anadimos el par
            "Nombre"-"Valor" al archivo de preferencias
        logros.flush();//Necesario para asegurarnos que se escribe
            correctamente en el archivo de preferencias

        usuario = logros.getString("Nombre-Usuario"); // Leemos el nombre
            de usuario
        cambiaUsuario = false;
        setScreen(new MenuPrincipal(this)); // Coloca la pantalla actual,
            se llama desde cualquier pantalla anterior y se llama a Screen.
            show desde la nueva pantalla
```

```
}

public void render() { //Metodo principal, se ejecuta constantemente (Game
    Loop)
        if(cambiaUsuario){
                logros = Gdx.app.getPreferences("logros-" + usuario);//
                    Cargamos el archivo de preferencias (Si no existe lo
                    crea)
                logros.putString("Nombre-Usuario", usuario);
                logros.flush();//Necesario para asegurarnos que se escribe
                    correctamente en el archivo de preferencias
                cambiaUsuario = false;
        }
        super.render();
}
```

Este es el único cambio que hay que hacer para individualizar los logros, mucho mas simple de lo que se podía pensar en un principio.

Pero queda introducir dicho usuario, para lo cual necesitaremos procesar la entrada de teclado. También muy sencillo: LibGDX nos proporciona un elemento para realizar la entrada de teclado de una manera muy cómoda.

Este elemento es un listener o 'escuchador', que consiste en una interfaz compuesta de dos funciones, la función **input(String text)** que se encargará de procesar el texto que introduzcamos por teclado, y la función **canceled()**, que se ejecutará al cancelar el cuadro de entrada de texto.

De esta forma, crearemos una nueva clase llamada EntradaTexto, que implemente esta interfaz y reciba como parámetro nuestra clase principal TuringAdventure, para que al introducir por teclado el usuario podamos añadirlo a nuestro juego.

Esto lo haremos mediante la función **CambiaUsuarios(String usuario)** que hemos definido en la clase TuringAdventure, que simplemente añadirá la cadena que le pasamos a la variable usuario de TuringAdventure, ademas de igualar a true la variable cambiaUsuario, para que en el render cargue de nuevo el archivo de preferencias:

```
public class EntradaTexto implements TextInputListener { //Con esta clase
    podremos recoger la entrada del teclado

        TuringAdventure juego;

        public EntradaTexto(TuringAdventure juego){
                this.juego = juego;
        }

        public void input (String text) { //Se ejecuta cuando pulsamos en
            aceptar
                juego.CambiaUsuarios(text); // Cambiamos el usuario actual
        }

        public void canceled () { //Cancela la entrada de teclado, se
            ejecuta cuando pulsamos en cancelar
        }
}

//En la clase TuringAdventure:
public void CambiaUsuarios(String usuario){
        this.usuario = usuario;
        cambiaUsuario = true;
}
```

Por último nos queda darle uso a esta clase que hemos creado. Lo haríamos en el menú principal, justo al pulsar el botón para cambiar usuarios. Para ello entramos al módulo de entrada de LibGDX y ejecutamos la función **getTextInput(TextInputListener listener, String title, String text)**, que creará un cuadro de texto con el título y el texto que le hayamos indicado, y procesará la entrada según el listener que le pasemos, que en nuestro caso sera un objeto de la clase EntradaTexto:

```
if(Gdx.input.justTouched() && botonCambiarUsuario.getBordes().contains(Gdx.
    input.getX(), Gdx.graphics.getHeight() - Gdx.input.getY())){
        Gdx.input.getTextInput(listener, "Usuario", "Introduce el usuario: "
            );//Llamamos a la funcion getTextInputs con nuestra clase
            EntradaTexto
}
```

De esta forma introducimos el usuario. Si no existe LibGDX creara un nuevo archivo de preferencias 'logros-usuarioX', y si ya existe entonces cargará el archivo ya existente.

Con esto concluimos por fín, habiendo añadido a la estructura básica de un videojuego algunos elementos extra que nos permitirán añadirle mas elementos y características adicionales.

Bibliografía

[1] Wiki oficial de LibGDX: https://code.google.com/p/libgdx/wiki/TableOfContents?tm=6

[2] Documentación de LibGDX: http://libgdx.badlogicgames.com/documentation.html

[3] Beginning Android Games. Mario Zechner. ISBN-13: 978-1430230427

[4] Blog dedicado a LibGDX: http://libgdxua.wordpress.com/

[5] Introducción a la programación en Android con LibGDX: https://docs.google.com/document/d/1YfuJ-gsc7VfIlX2G8Ix5K0laaqEsvFDuTz3Cnecy2_M/edit?pli=1

[6] Lista de Youtube: Juegos con LibGDX: http://www.youtube.com/playlist?list=PLTd5ehIj0goMPH76uEG4bXSNcwhGeda9N

[7] Canal de desarrollo de un videojuego en LibGDX: http://www.youtube.com/user/doctoriley?feature=watch

[8] Book of Inkscape: The Definitive Guide to the Free Graphics Editor. Dmitry Kirsanov. ISBN-13: 978-1593271817

[9] Inkscape: Guide to a Vector Drawing Program. Tavmjong Bah. ISBN-13: 978-0132764148

[10] Diseño de Gráficos con Inkscape: http://2dgameartforprogrammers.blogspot.co.uk/

GNU Free Documentation License

Version 1.3, 3 November 2008

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.
You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.
The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.
"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.
An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.