Proyecto Final de Carrera: Videojuegos educativos

Kamikawa, videojuego para practicar el vocabulario de la lengua inglesa.

Antonio Javier Garrido Rodríguez
2º ciclo de Ingeniería en Informática
antoniojaviergr@gmail.com

Heliodoro Tejedor Navarro

22/01/2014

Dedicado a mis padres,
Antonia y Sebastián.

AGRADECIMIENTOS:

A mi madre, por su paciencia y velar por mi "ambiente de estudio".

A mi primo Andrés, por haberme introducido en el mundo de la informática. Me regaló mi primer ordenador, y me dio clases veraniegas de manera totalmente voluntaria.

A mi cuñado, por tener la iniciativa y voluntad de prestarme su ordenador de sobremesa. Me ha ofrecido una buena herramienta para poder desarrollar sin problemas el videojuego.

A Helio por su asesoramiento, ayuda y ánimos.

GNU Free Documentation License (GNU FDL)

Copyright © 2014 Antonio Javier Garrido Rodríguez.

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".

FICHA DEL TRABAJO FINAL

Título del trabajo:	Kamikawa, videojuego para practicar el vocabulario de la lengua inglesa.
Nombre del autor:	Antonio Javier Garrido Rodríguez
Nombre del consultor:	Heliodoro Tejedor Navarro
Fecha de entrega (mm/aaaa):	01/2014
Área del Trabajo Final:	Videojuegos educativos
Titulación:	2º ciclo de Ing. Informática

Resumen del Trabajo (máximo 250 palabras):

El presente PFC (Proyecto Final de Carrera) trata sobre la implementación de un videojuego en Android, sin utilizar motores de desarrollo, únicamente la funcionalidad nativa de Android.

El juego consiste en formar palabras válidas en la lengua inglesa. Para ello, aparecen letras del vocabulario, las cuales se desplazan por la pantalla de forma aleatoria, y se deben seleccionar mediante pulsaciones para ir formando palabras válidas. Una palabra es válida cuando el Display que la muestra está de color verde. Entonces, pulsamos sobre él para validarla, recibir los puntos que procedan y eliminar las letras que la forman de la pantalla.

Completamos un nivel cuando permanecemos el tiempo de duración del mismo sin exceder el número máximo de letras permitidas en pantalla. A medida que superamos niveles va aumentando la frecuencia de aparición de las letras y su velocidad de movimiento. Es decir, la dificultad aumenta de forma proporcional en relación al número de niveles. El nivel máximo es el 10, una vez superado llegamos al final del juego.

Abstract (in English, 250 words or less):

The present Final Degree Project (FDP) discusses the implementation of a video-game in Android without using any video-game development engine, only the native Android functionality.

The game consists of forming valid English words. In order to do this, some letters of the English vocabulary will appear on the screen and these letters will be moving randomly. The letters will be selected by touching them one by one until a right word is formed. A word is considered valid when the display that is showing it is green. We then click on it in order to validate it , receive the points allotted and eliminate the letters that form this word from the screen.

You pass a level when you stay within the duration time without exceeding the maximum amount of letters allowed on screen. The more levels you pass, It increases the occurrence frequency of letters and their movement speed. The difficulty increases proportionally to the levels passed. The highest level is 10, once completed you will have reached the end of the game.

Palabras clave (entre 4 y 8):

Índice

1. Introducción……………………………………………………………..……..1
   1.1 Contexto y justificación del Trabajo……………………………............1
   1.2 Objetivos del Trabajo…………………………………………..……......1
   1.3 Enfoque y método seguido……………………………..…………....3
   1.4 Planificación del proyecto…………………………..………………..…3
      1.4.1 Recursos necesarios……………………………………..………4
      1.4.2 Actividades y planificación……………………………………….5
   1.5 Breve sumario de productos obtenidos…………...………………........7
   1.6 Breve descripción de los otros capítulos de la memoria…………..…8
2. Análisis………………………………………………………………..…..9
   2.1 Requisitos funcionales……………………………………………......9
   2.2 Requisitos no funcionales...………………………………………..…9
3. Diseño…………………………………………………………………11
   3.1 Especificaciones del videojuego………………………………………11
   3.2 Diseño de la interfaz gráfica……………………………………......14
   3.3 Diseño de datos………………………………………………........19
   3.4 Diseño de la arquitectura……………………………………………..…19
4. Desarrollo……………………………………………………………..21
   4.1 Entorno de desarrollo utilizado……………………………………....21
   4.2 Inicio del desarrollo……………………………………………………...21
   4.3 Representación visual de funcionalidades y elementos del código...22
   4.4 Lógica de funcionamiento…………………………………………23
   4.5 Aspectos más relevantes de la codificación………………………….24
      4.5.1 Menú del juego……………………………………………......24
      4.5.2 Bucle del juego…………………………………………..…25
      4.5.3 Los agentes letras……………………………………………….29
      4.5.4 Palabras válidas de la gramática inglesa y su uso……….…34
      4.5.5 Persistir datos del juego…………………………………………35
   4.6 Codificación……………………………………………………………36
      4.6.1 Clases implementadas y su funcionalidad……………………36

4.6.2 Estructuración de los elementos……………………………..45
5. Pruebas…………………………………………………………………..…48
6. Valoración económica……………………………………………………50
7. Viabilidad del producto……………………………………………………51
8. Conclusiones………………………………………………………………52
9. Glosario…………………………………………………………………….54
10. Bibliografía…………………………………………………………………55

Lista de figuras

Figura 1: Diagrama de Gantt…………………………………………………..7
Figura 2: Pantalla Inicial de carga……………………………………………15
Figura 3: Pantalla Menú principal……………………………………………..15
Figura 4: Pantalla Iniciar juego………………………………………………..…16
Figura 5: Pantalla Juego……………………..…………………………………...16
Figura 6: Pantalla Mejores jugadas…………………………………………...17
Figura 7: Pantalla Configuración………………………………………………17
Figura 8: Ayuda antes de jugar……………………………………………..…18
Figura 9: Pantalla Sobre el juego…………………………………………… .18
Figura 10: Diagrama de transición entre pantallas…………………………..…19
Figura 11: Diagrama arquitectura básica funcional….…………………………20
Figura 12: Elementos de codificación…………………………………………22
Figura 13: Pantallas y elementos de la codificación……………………………23
Figura 14: Agente letra azul……………………………………………………30
Figura 15: Agente letra rojo.……………………………………………………30
Figura 16: Clases del código………..…………………………………………36
Figura 17: Otros elementos del código….……………………………………45

1. Introducción

El objetivo del presente capítulo es hacernos una idea general sobre el proyecto. Para ello, expongo brevemente los motivos de la realización de este trabajo, sus objetivos, enfoque y metodología, planificación de las actividades que componen el proyecto, cuales son los productos obtenidos y una escueta descripción del resto de capítulos que forman la memoria.

1.1 Contexto y justificación del Trabajo

En la elección de mi PFC apliqué como principal criterio que fuese útil y provechoso. Para ello, analicé un poco las tendencias actuales, y digo un poco porque es demasiado evidente, al menos desde mi punto de vista, cuales son las tendencias actuales. Los teléfonos inteligentes están proliferando a un ritmo exponencial; en el último trimestre se han vendido aproximadamente 250 millones de unidades, y más del 80% de éstos funcionan con el SO de Android.

Ante tales circunstancias, me parece muy interesante el desarrollo de aplicaciones para dispositivos con Android. Porque estaremos desarrollando aplicaciones con la posibilidad de estar presente en millones de dispositivos.

Mis conocimientos sobre Android antes de realizar el trabajo eran sólo a nivel de usuario, tenía un teléfono inteligente con dicho SO y simplemente lo utilizaba. Bajo las circunstancias expuestas, me pareció una gran oportunidad y buena idea, aprender a desarrollar aplicaciones para dispositivos móviles con Android; y aprovechar la realización de éste trabajo para iniciarme en ello. Además nunca había implementado un videojuego de cierta envergadura, sólo juegos muy sencillos, y con fines puramente didácticos.

En conclusión, creo que la elaboración de dicho trabajo va a ser una experiencia muy productiva y enriquecedora a nivel de conocimientos, a la vez que divertida y amena, por el propio carácter del videojuego. En resumen, demasiados motivos de peso para hacerlo, y ninguno para no hacerlo.

1.2 Objetivos del Trabajo

En cuanto a los objetivos personales, el principal es aprender sobre el diseño y desarrollo de aplicaciones para Android. Éste lleva implícito a su vez los siguientes objetivos:

- Conocer cómo funciona el SO Android.
- Aprender a utilizar el Framework de Android para el desarrollo de aplicaciones.
- Conocer y utilizar parte de las funcionalidades ofrecida en sus APIs.
- Tener alguna experiencia en el desarrollo para Android.

Continuando con los personales, y un poco más en segundo plano; aprender algo sobre el diseño y desarrollo de videojuegos. Hasta ahora era un campo casi desconocido para mí.

En cuanto a los objetivos del videojuego, debemos ser consciente y tener muy en cuenta el contexto al que se estará expuesto cuando se juega. Es decir, la mayoría de jugadores de videojuegos para dispositivos móviles juegan en sus “tiempos muertos”. Éstos son por ejemplo, cuando van en transporte público, cuando están esperando a alguien o simplemente disponen de un pequeño tiempo libre de ocio. En consecuencia, los objetivos a tener en cuenta deben ser:

- Que las partidas sean cortas.
- Fácil de jugar.
- Con una mecánica sencilla.
- Pausable. Ya que estará expuesto a continuas interrupciones.

Otro objetivo propuesto es desarrollar el videojuego completamente desde cero, es decir, no utilizar ningún motor de desarrollo para videojuegos. Toda la funcionalidad será implementada por mí.

Por último, creo que es muy interesante la compatibilidad del videojuego con los distintos dispositivos móviles. En principio se va a desarrollar para que sea compatible con el mayor número posible de dispositivos diferentes. Ya que en el mercado existe un gran variedad en cuanto a resolución y densidad de pantalla.

1.3 Enfoque y método seguido

Para llevar a cabo el trabajo había varias posibilidades: desarrollar alguna funcionalidad para la plataforma K-PAX, tener una idea propia y hacer una propuesta de trabajo o que el consultor propusiese un trabajo. Había bastante flexibilidad a la hora de proponer y elegir trabajo. En mi caso particular, para cumplir con mis objetivos y pretensiones, creo que la mejor opción es realizar un videojuego completamente desde cero. Y así aprender los conocimientos teóricos y prácticos necesarios para su elaboración.

Era consciente que el desarrollo del trabajo era toda una aventura, sin falta de riesgo, ya que no tenía ni idea sobre el desarrollo de videojuegos para dispositivos móviles y tampoco de la plataforma de desarrollo que iba a utilizar, así que el aprendizaje estaba garantizado pero no tanto el éxito.

Debía enfocar bien las acciones de trabajo y realizar un buen plan, si quería obtener un resultado digno y aceptable. Por lo tanto, el enfoque inicial fue el siguiente:

1- Tener claro o lo suficientemente claro **qué** debo hacer. Para ello, debía leer e informarme sobre videojuegos para dispositivos móviles.

2- Aprender **cómo** debo hacerlo. Preparar el entorno de desarrollo; aprender los conocimientos básicos sobre la plataforma de desarrollo de Android y realizar algunas aplicaciones básicas y sencillas; aprender lo básico sobre el desarrollo de videojuegos.

3- Una vez sabido qué debo hacer y cómo, estaba en condiciones de poder hacer estimaciones. Otro factor importante y determinante es el tiempo, así que teniendo en cuenta los factores de: **qué**, **cómo** y el **tiempo** disponible; estaba en disposición de poder hacer estimaciones sobre la magnitud del resultado que podría obtener.

Para la consecución del punto 1, analicé y jugué a bastantes videojuegos para dispositivos móviles, y así poder establecerme una buena referencia. Hubo un videojuego que llamó especialmente mi atención, me gustó; y ese es “Letris”.

Dado que no tenía ninguna experiencia en el diseño de videojuegos, creo que la mejor opción es tomar uno como referencia y a partir de ahí, diseñar y desarrollar el mío. Así hice, tomé como guía el videojuego de “Letris”, y basándome en él, diseñé y desarrollé el mío. He tomado como referencia la temática y modo de juego, basándome en unos videos que he visto en “Youtube” sobre unas grabaciones de partidas. Donde muestran la interfaz del juego, con el menú, acceso a las diferentes opciones del juego y el inicio de una de dos partidas; una en la que supera el nivel y otra en la que pierde. Teniendo esta información como referencia, diseñé el mío.

Una vez enfocado mi trabajo, y con la realización del GDD y la PEC1, tenía un boceto bastante ilustrativo sobre mi videojuego. Ahora debía elegir el método de trabajo a seguir. Éste lo tuve claro desde un principio, aplicaría la metodología o principio de “**desarrollo ágil de software**”. Donde tuviese rápidamente una demo (versión inicial ejecutable y con algún tipo de resultado), y mediante iteraciones, ir añadiendo funcionalidades hasta obtener el resultado final.

El resultado de mi primera iteración, y base tomada para el resto de iteraciones, fue algo muy simple. Dos pantallas; en la primera y punto de inicio de ejecución, tenía un sólo botón que daba paso a la siguiente pantalla; y en ésta, una imagen que se movía aleatoriamente por la pantalla, delimitando su movimiento por los límites de la pantalla.

1.4 Planificación del proyecto

En este apartado comentaré los recursos de software y hardware necesarios para realizar el trabajo, las actividades realizadas y una planificación temporal de éstas.

1.4.1 Recursos necesarios

Nombraré los recursos utilizados en mi caso personal, todos son sustituibles por otros que cumplan sus mismas funciones, excepto los propios de la plataforma de desarrollo de Android.

Recursos hardware:

- *Ordenador de sobremesa*: AMD Intel Core Duo a 3 GHz y 4 GB de memoria RAM. Esta herramienta es esencial en la fluidez para avanzar el proyecto. El ordenador debe esperar a que nosotros pensemos, no al contrario… En un principio no tenía esta maravilla como herramienta de desarrollo, disponía de un portátil con 7 años y 1 GB de memoria RAM. Cuando instalé todas las herramientas necesarias para el desarrollo e intenté utilizarlas, nos echamos a llorar los dos… Para solucionarlo mi cuñado me prestó el ordenador de sobremesa, acto por el cual le estoy enormemente agradecido.

- *Dispositivos móviles auxiliares*: En un principio para ir probando los avances en el desarrollo del videojuego utilizaba un dispositivo virtual, pero éste no funcionaba con fluidez, así que lo sustituí por mi Smartphone. Un Sony Xperia S, este dispositivo ofrece un velocidad de procesamiento más que aceptable y una pantalla de calidad, con una alta resolución y densidad. Además, para las pruebas finales de compatibilidad en diferentes dispositivos móviles utilicé:

    - Smartphone LG Optimus L7.
    - Smartphone Samsung Galaxy S3.
    - Tablet Galaxy Tab 3 10.1

Recursos software:

Software de desarrollo: software utilizado para programar utilizando el Framework de Android.

- Eclipse [1]: IDE utilizado para la programación del videojuego.
- JDK [2]: Recordemos que el sistema operativo de Android se basa en el lenguaje de programación Java. Por tanto, necesitamos el intérprete de Java.
- SDK Android [3]: Para programar aplicaciones para el SO de Android, necesitamos dicho kit de desarrollo.
- Plugin ADT [4] para Eclipse: Una vez instalado el SDK, necesitamos incluir y configurar el ADT en Eclipse, éste permite la integración entre el SDK y el IDE Eclipse.

Creo innecesario explicar cómo instalar y configurar dichas herramientas, en internet hay multitud de artículos que lo hacen, además creo que no entra dentro del ámbito de la memoria.

Otras herramientas:

- Microsoft Word para la redacción de documentos.
- Microsoft Project para elaborar el diagrama de Gantt.
- Navegador Chrome para la búsqueda de información.
- Xara 3D Maker para la creación de los Sprites letras.
- Paint para algunos retoques gráficos.

1.4.2 Actividades y planificación

A continuación comento las tareas que he realizado para elaborar el TFC y una planificación temporal de éstas.

Calendario de trabajo:

El calendario o ritmo de trabajo son 3,5 horas, seis días a la semana, de lunes a sábados, descansando los domingos. Este ha sido el ritmo de trabajo por norma general, excepto en algunas ocasiones citadas a continuación:

- Durante la fase de implementación, no pude dedicar la mayoría de viernes y sábados.
- Los días 11 y 12 de enero dedico 9 horas cada día.
- Los días comprendidos entre el 13 y 22 de enero, ambos inclusive, trabajo todos, y dedico 7 horas al día.
- Los días 25 de diciembre y 1 de enero me los tomo libre.

El número de horas totales dedicadas a la realización del trabajo son **361.**

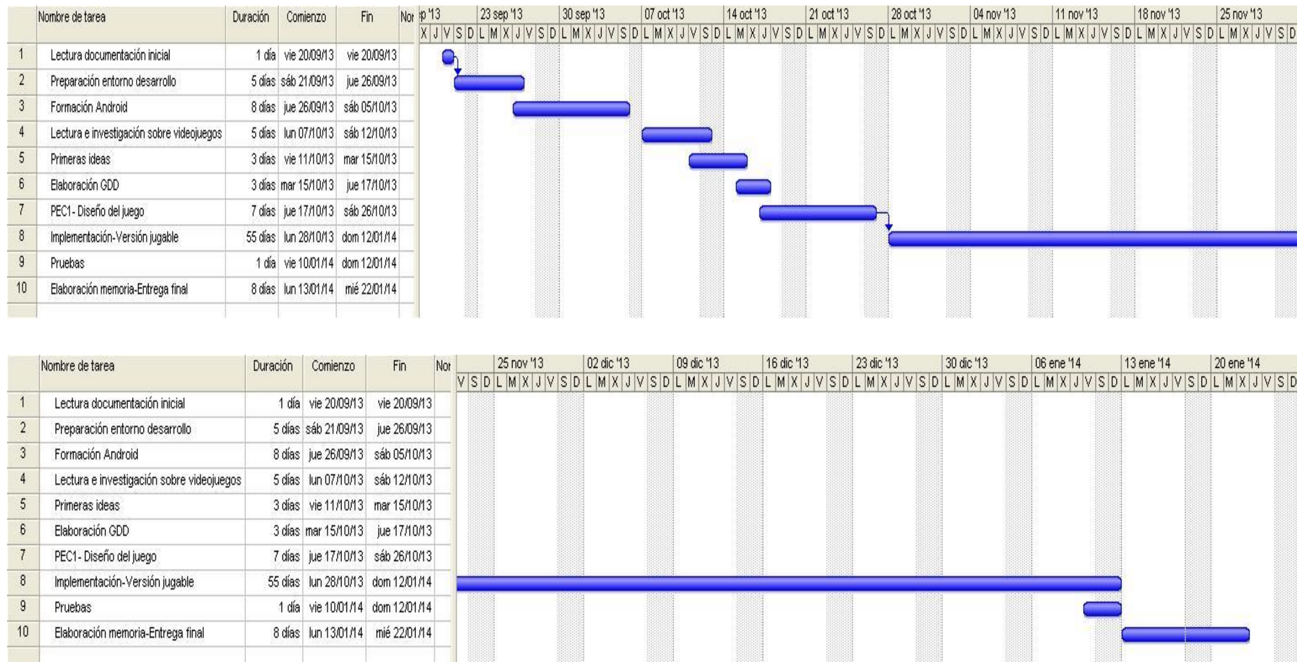
Tareas:

Las tareas ejecutadas para realizar el proyecto son las siguientes:

ID Tarea	Nombre	Descripción	Duración (horas)	Fecha Inicio	Fecha Fin
1	Lectura documentación inicial	Lectura de la documentación inicial recomendada	3,5	Semana 1 20 septiembre	Semana 1 20 septiembre
2	Preparación entorno de desarrollo	Lectura sobre el entorno de desarrollo necesario, obtención de las herramientas necesarias, instalación y configuración	15	Semana 1 21 septiembre	Semana 2 26 septiembre
3	Formación Android	Lectura y prácticas sobre programación básica en Android y sobre programación de videojuegos, para adquirir los conocimientos básicos	28	Semana 2 26 septiembre	Semana 3 5 octubre
4	Lectura e investigación sobre videojuegos	Leer información, probar y analizar algunos videojuegos existentes	17,5	Semana 3 7 octubre	Semana 4 12 octubre
5	Primeras ideas	Pienso y anoto las primeras ideas sobre el videojuego	7	Semana 4 11 octubre	Semana 5 15 octubre
6	Elaboración GDD	Defino a grandes rasgos algunos aspectos del videojuego	7	Semana 5 15 octubre	Semana 5 17 octubre
7	PEC1-Diseño del juego	Elaboración de la PEC1	28	Semana 5 17 octubre	Semana 6 26 octubre
8	Implementación-Versión jugable	Implementación del videojuego	167	Semana 6 28 octubre	Semana 17 12 enero
9	Pruebas	Pruebas del videojuego en diferentes dispositivos y correcciones para su adecuada adaptación a las diferentes pantallas (teniendo en cuenta resolución y densidad)	18	Semana 17 10 enero	Semana 17 12 enero
10	Elaboración memoria-Entrega final	Elaboración de la memoria del proyecto	70	Semana 18 13 enero	Semana 19 22 enero

Diagrama de Gantt:

Expongo la planificación de las tareas en el tiempo mediante un diagrama de Gantt, para visualizar claramente la programación temporal.



	Nombre de tarea	Duración	Comienzo	Fin
1	Lectura documentación inicial	1 día	vie 20/09/13	vie 20/09/13
2	Preparación entorno desarrollo	5 días	sáb 21/09/13	jue 26/09/13
3	Formación Android	8 días	jue 26/09/13	sáb 05/10/13
4	Lectura e investigación sobre videojuegos	5 días	lun 07/10/13	sáb 12/10/13
5	Primeras ideas	3 días	vie 11/10/13	mar 15/10/13
6	Elaboración GDD	3 días	mar 15/10/13	jue 17/10/13
7	PEC1- Diseño del juego	7 días	jue 17/10/13	sáb 26/10/13
8	Implementación-Versión jugable	55 días	lun 28/10/13	dom 12/01/14
9	Pruebas	1 día	vie 10/01/14	dom 12/01/14
10	Elaboración memoria-Entrega final	8 días	lun 13/01/14	mié 22/01/14

	Nombre de tarea	Duración	Comienzo	Fin
1	Lectura documentación inicial	1 día	vie 20/09/13	vie 20/09/13
2	Preparación entorno desarrollo	5 días	sáb 21/09/13	jue 26/09/13
3	Formación Android	8 días	jue 26/09/13	sáb 05/10/13
4	Lectura e investigación sobre videojuegos	5 días	lun 07/10/13	sáb 12/10/13
5	Primeras ideas	3 días	vie 11/10/13	mar 15/10/13
6	Elaboración GDD	3 días	mar 15/10/13	jue 17/10/13
7	PEC1- Diseño del juego	7 días	jue 17/10/13	sáb 26/10/13
8	Implementación-Versión jugable	55 días	lun 28/10/13	dom 12/01/14
9	Pruebas	1 día	vie 10/01/14	dom 12/01/14
10	Elaboración memoria-Entrega final	8 días	lun 13/01/14	mié 22/01/14

Figura 1: Diagrama de Gantt

1.5 Breve sumario de productos obtenidos

Durante la realización del PFC he obtenido los siguientes productos o resultados:

- **GDD**: documento conceptual, el cual define a grandes rasgos algunos aspectos del videojuego.
- **PEC1-Diseño del videojuego**: documento el que defino la idea inicial del juego, indicando su mecánica y detalles del diseño.
- **Versión jugable**: código fuente del videojuego, completamente funcional y en su versión final.
- **Memoria**: presente documento, en el que explico el proceso de desarrollo del videojuego.
- **Presentación multimedia**: video explicativo en el que presento y expongo el PFC.

1.6 Breve descripción de los otros capítulos de la memoria

Capítulo 2. Análisis: indico los requisitos funcionales y no funcionales del videojuego.

Capítulo 3. Diseño: muestro las especificaciones del videojuego, tales como género y objetivos, mecánica del juego, control del juego, diseño de sus pantallas. Además indico el diseño para el sonido, la puntuación, los datos a persistir, la arquitectura y el método de desarrollo utilizado para su implementación.

Capítulo 4. Desarrollo: explico los elementos propios de la codificación del videojuego. Como el entorno de desarrollo utilizado, arquitectura entrando en detalles de codificación, lógica de funcionamiento del código, aspectos más relevantes de la codificación y explicación del código en general.

Capítulo 5. Pruebas: indico cómo he probado el videojuego, sobre qué dispositivos y los resultados obtenidos.

Capítulo 6. Valoración económica: gastos ocasionados por el desarrollo del proyecto y los beneficios obtenidos.

Capítulo 7. Viabilidad del producto: explicar si puede tener salida como producto y si será útil para la comunidad.

Capítulo 8. Conclusiones: lecciones aprendidas con la elaboración del proyecto, reflexión crítica sobre el logro de los objetivos planteados, análisis crítico sobre el seguimiento de la planificación y metodología a lo largo del desarrollo del producto y las líneas de trabajo pendientes.

2. Análisis

Al ser un trabajo docente e individual, tiene la peculiaridad de que los requisitos no se extraen de ninguna fuente externa, sino que debo inferirlo yo mismo. Por lo tanto, son requisitos que quedarán perfectamente definidos desde el inicio del proceso de desarrollo y no cambiarán.

2.1 Requisitos funcionales

Cómo dije anteriormente, el contexto de los videojuegos para dispositivos móviles es diferente al de otros tipos de dispositivos. Teniendo ello en cuenta, concluyo que el videojuego debe satisfacer los siguientes requisitos:

- Mecánica de juego sencilla.
- Intuitivo y fácil de jugar.
- Partidas cortas.
- Pausable.
- Que incite a la competencia y superación de objetivos. Para ello debe persistir datos en el dispositivo.

2.1 Requisitos no funcionales

Uno de los requisitos a tener más en cuenta es la compatibilidad con el mayor número de dispositivos posibles, para ello hay que tener en cuenta dos factores:

- Las diferentes versiones de Android.

He tomado como versión objetivo la más popular de Android, y ésta según fuentes de Google [5] es la versión Jelly Bean, presente en el 59,1 % de los dispositivos. Por lo tanto, el videojuego está optimizado para dicha versión. Es la que tiene instalada el dispositivo de pruebas que he utilizado durante todo el desarrollo.

- Los diferentes tipos y tamaños de dispositivos (Smartphones y Tablets).

He utilizado una metodología de implementación ofrecida desde la versión 3.2 de Android. Esta metodología permite adaptar las aplicaciones Android a los diferentes tipos y tamaños de dispositivos, de una forma más eficiente y sofisticada que la que se venía utilizando hasta dicha versión. Las dos metodologías son incompatibles y excluyentes.

Utilizo la metodología más reciente, ya que asegura un mejor funcionamiento. El inconveniente es que se vuelve excluyente para los dispositivos con versiones de Android inferiores a la 3.2. He consultado datos actuales ofrecidos por Google [5], y sólo el 22, 5% de los dispositivos utilizan versiones de Android inferiores a la 3.2. Y con la lógica tendencia de que vayan

disminuyendo progresivamente. En consecuencia, creo que la mejor opción es utilizar la metodología más reciente, aunque esto suponga excluir un pequeño porcentaje de dispositivos.

En principio el videojuego es compatible con todas las versiones de Android desde la 3.2 en adelante, y con todo tipo y tamaños de dispositivos; ofreciendo un óptimo resultado de ejecución en la versión más popular.

Los requisitos no funcionales que debe satisfacer son los siguientes:

- Compatibilidad con el mayor número de versiones de Android.
- Compatibilidad con el mayor número de diferentes tipos y tamaños de dispositivos.
- El idioma será español.
- Disponer de un Log con información sobre la ejecución.
- No utilizar ningún motor de desarrollo.

3. Diseño

En el actual capítulo presento los detalles del diseño del videojuego, definiendo su arquitectura general, la representación de la interfaz, la estructura de datos y la mecánica en general. Una vez finalizado el diseño, tendremos más o menos claro, qué es lo que se debe implementar.

3.1 Especificaciones del videojuego

En esta sección describiré los aspectos más superficiales, dando una idea global sobre el videojuego y su funcionamiento.

Género y objetivos del juego

Por la propia área de proyecto, el videojuego debe ser educativo, y dentro de este amplio rango, Kamikawa pertenece al género de *puzzles*. Está basado en el videojuego Letris, aunque con un aspecto más dinámico e interactivo.

El objetivo del juego es llegar y superar su último nivel, consiguiendo la máxima puntuación posible. Para ello, debemos superar cada nivel, permaneciendo el tiempo de duración del mismo sin sobrepasar el número máximo de letras permitidas en pantalla.

Mecánica del juego

El juego consiste básicamente en formar palabras en inglés. Para ello disponemos de un escenario, en el cual, van apareciendo letras del abecedario. Éstas aparecerán en forma de simpáticos muñequitos que se desplazan por la pantalla en las cuatro direcciones posibles, izquierda, derecha, arriba y abajo. Caminan de forma aleatoria hasta llegar a un límite de pantalla, cambiando su dirección. Los muñequitos o letras van apareciendo individualmente, de forma progresiva y a un ritmo constante. La frecuencia de aparición y la velocidad de movimiento, será determinado por el nivel en el que se encuentre el videojuego, a mayor nivel, mayor velocidad de movimiento y frecuencia de aparición.

El modo de formar una palabra es haciendo pulsaciones con el dedo sobre cada una de las letras que la representen. El juego cuenta con un *display* el cual visualiza la palabra que se está formando en cada momento. Si la palabra actual es reconocida como palabra válida en la lengua inglesa, el *display* cambiar al color verde, sino continua en rojo. Para finalizar su formación y confirmación de la misma, hay que hacer una pulsación sobre el *display* que visualiza la palabra, cuando la palabra sea reconocida como válida. Con ello, eliminamos de la pantalla a los agentes letras que formen la palabra, disminuyendo el número de estos en pantalla y recibiendo la puntación que proceda en función a la longitud de la palabra. Si se desea, es posible desmarcar letras de una palabra inválida, mediante una pulsación sobre el *display*.

Para superar un nivel hay que permanecer el tiempo de duración del mismo, formando palabras sin exceder en ningún momento el número máximo de agentes letras permitidos en la pantalla. El tiempo de duración de cada nivel es de 2 minutos, y el número máximo de agentes letras permitidos simultáneamente en la pantalla es de 25, para los niveles desde el 1 hasta el 5, ambos inclusive; y de 50 para el resto.

La dificultad en cada nivel vendrá determinada por la frecuencia de aparición y la velocidad de movimiento de los agentes letras, en el primer nivel estos parámetros tendrán los valores mínimos posibles e irán aumentando progresivamente a medida que avancen los niveles. El videojuego cuenta con 10 niveles, una vez superado llegaremos al final del juego.

La puntuación se consigue a través de las palabras que se vayan formando. El número de puntos asociados a cada palabra dependerá de la longitud de éstas, a mayor longitud de palabra mayor puntuación. El aumento de la puntuación es exponencial en base dos, calculado a partir de la longitud. Esto se indicará con mayor detalle en el apartado *Puntuación*.

Niveles

El juego consta de 10 niveles, al superar el nivel décimo llegaremos al final del juego. En un principio contemplé la posibilidad de implementar infinitos niveles, ya que no es muy difícil; tan solo habría que ir aumentando la frecuencia de aparición de los agentes letras y la velocidad de movimiento progresivamente hasta el infinito.

Ello no lo hice porque considero más atractiva la idea de tener un final e intentar llegar hasta él. Aunque me he encargado a conciencia, de que no sea tarea fácil, no es imposible, pero si difícil.

A continuación muestro en una tabla dos datos relevantes acerca del juego en cada nivel. La segunda columna indica la frecuencia de aparición de los agentes letras, es decir, cada cuanto tiempo expresado en milisegundos aparece un agente en pantalla. Y la segunda columna indica el número de agentes letras que se crearán durante el transcurso de la partida.

Nivel	Frecuencia aparición	Letras creadas
1	3600	33
2	3200	37
3	2800	42
4	2400	50
5	2000	60
6	1600	75
7	1400	85
8	1200	100
9	1000	120
10	800	150

Control del juego

Kamikawa se controla exclusivamente desde la pantalla táctil del dispositivo, mediante pulsaciones sobre los botones de interacción y sobre los agentes letras. La interacción es fácil e intuitiva.

Sonido

En el videojuego existen dos tipos de sonidos que por defecto estarán activados. Podremos desactivarlo/activarlo desde la opción *Configuración* del menú principal, y podremos ajustar su volumen con los mismos botones que ajustan el volumen en el dispositivo. Estos sonidos son los siguientes:

- Sonido ambiental: compuesto por la música de fondo mientras se juega una partida y la música del final del juego (cuando superamos el nivel 10).
- Sonido de efectos: compuesto por sonidos de corta duración producidos mediante la interacción con los elementos interactivos del juego. A continuación indico cuales son los eventos o interacciones que provocan sonidos de efectos, estas son las siguientes:

  - Al pulsar sobre cualquier opción o botón de las diferentes pantallas.
  - Al pulsar sobre un agente letra.
  - Al pulsar el botón de *pause*.
  - Al pulsar el botón *acelerar.*
  - Al pulsar el *display* para borrar una palabra.
  - Al pulsar el *display* para validar una palabra.

  Existen otros sonidos de efectos que son sonidos de animación. Estos no son provocados de forma directa con algún tipo de interacción sino que los produce el propio videojuego bajo ciertas circunstancias, estas son las siguientes:

  - Cuando superamos un nivel.
  - Cuando perdemos una partida.

Los recursos de audio han sido obtenidos de páginas que ofrecen archivos de audio de forma libre y gratuita [11].

Puntuación

Los puntos se obtienen formando y validando palabras. La puntuación conseguida es directamente proporcional a la longitud de éstas. El aumento es

exponencial en base dos, tomando como exponente la longitud de la palabra. Creo que es un sistema justo de valoración, premiando de forma exponencial la longitud de las palabras. Por lo tanto, el valor de las palabras en función a su longitud sería el siguiente:

Longitud en caracteres	1	2	3	4	5	6	…
Valor en puntos	2	4	8	16	32	64	…
Forma de calcular puntuación	2^1	2^2	2^3	2^4	2^5	2^6	…

3.2 Diseño de la interfaz gráfica

En este apartado ilustraré el diseño de la interfaz gráfica, mostrando cada una de las pantallas que componen el videojuego y la transición entre estas. Antes de comenzar, indicaré algunas nociones sobre las pantallas:

- Una pantalla es una unidad independiente que ocupa por completo el visor del dispositivo y que es responsable de una parte del videojuego.

- Pueden estar compuesta por varios componentes como por ejemplo, botones, etiquetas, imágenes etc.

- Permiten al usuario interactuar con los elementos del videojuego. Estas interacciones pueden iniciar transiciones entre pantallas.

Una vez sabida las nociones anteriores, pasaré a especificar las pantallas del videojuego:

Pantalla Inicial de carga

Primera pantalla mostrada cuando iniciamos el videojuego. En ella aparece el nombre de este y un texto con la palabra “Cargando”. Su permanencia es temporal, una vez procesados y cargados en memoria los datos necesarios para iniciar el videojuego, se produce de forma automática la transición a la siguiente pantalla, la cual mostrará el menú principal.

Figura 2: Pantalla Inicial de carga

Pantalla Menú principal

Muestra el menú principal con todas las opciones que ofrece el video juego. Desde ella podemos navegar a cada una de las diferentes opciones.

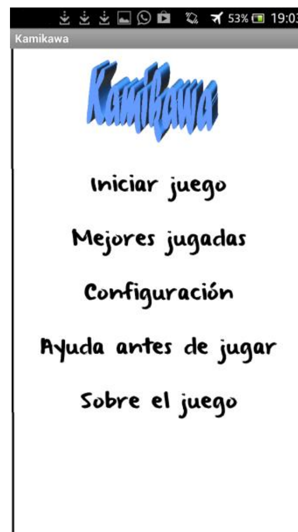


Figura 3: Pantalla Menú principal

Pantalla Iniciar juego

Muestra los niveles disponibles para iniciar el juego. Inicialmente no habrá niveles disponibles, en consecuencia estaremos obligados a iniciar en el nivel 1. Para ello, pulsamos sobre el botón *iniciar.* A medida que vayamos superando niveles, estos estarán disponibles en dicha pantalla, y podremos iniciar desde cualquier nivel que ya se haya abierto o jugado.

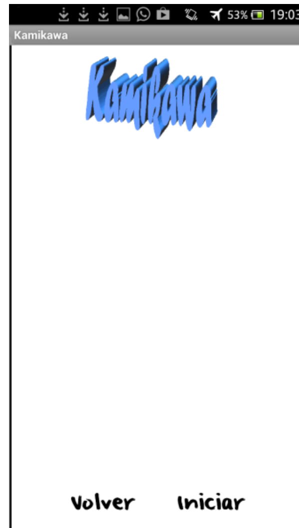


Figura 4: Pantalla Iniciar juego

Pantalla Juego

Es donde transcurre la partida del juego. En ella diferenciamos 3 partes:

- HUD: parte superior de la pantalla donde mostramos información relevante para el transcurso de la partida.
- Parte inferior: es la zona donde están ubicados los botones de interacción; *pause*, *display* y botón *acelerar*.
- Parte central o zona de movimiento: es la parte comprendida entre las dos anteriores, y es el área de pantalla por la que se mueven los agentes letras.

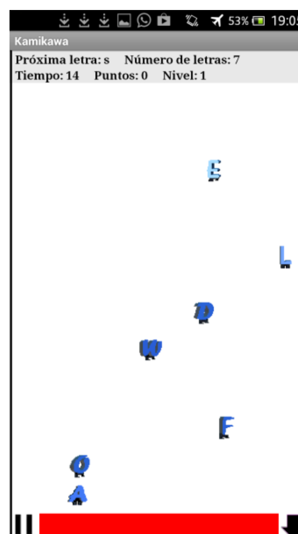


Figura 5: Pantalla Juego

Pantalla Mejores Jugadas

Visualiza una lista con las 10 mejores jugadas. El parámetro utilizado para evaluar y clasificar una jugada es la puntuación. Inicialmente la lista estará vacía, y a medida que vayamos jugando se irá completando dicha lista, ordenando las jugadas de mejor a peor, manteniendo en la lista sólo las diez mejores.

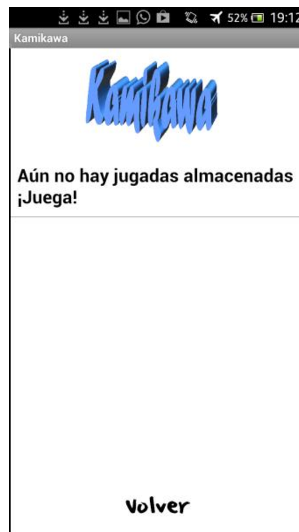


Figura 6: Pantalla Mejores jugadas

Pantalla Configuración

Permite desactivar o activar, de forma independiente, los dos tipos de sonidos: sonido de efectos y sonido ambiental.

Figura 7: Pantalla Configuración

Pantalla Ayuda antes de jugar

Muestra un texto explicando cómo jugar, es decir, las reglas del juego y cómo interactuar con los elementos.

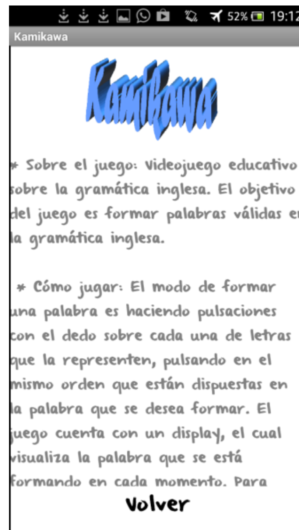


Figura 8: Ayuda antes de jugar

Pantalla Sobre el juego

Visualiza información sobre el videojuego: nombre, versión y tipo de licencia.

Figura 9: Pantalla Sobre el juego

Diagrama de transición entre pantallas:

Una vez vistas las diferentes pantallas, muestro las transiciones entre estas mediante un diagrama:

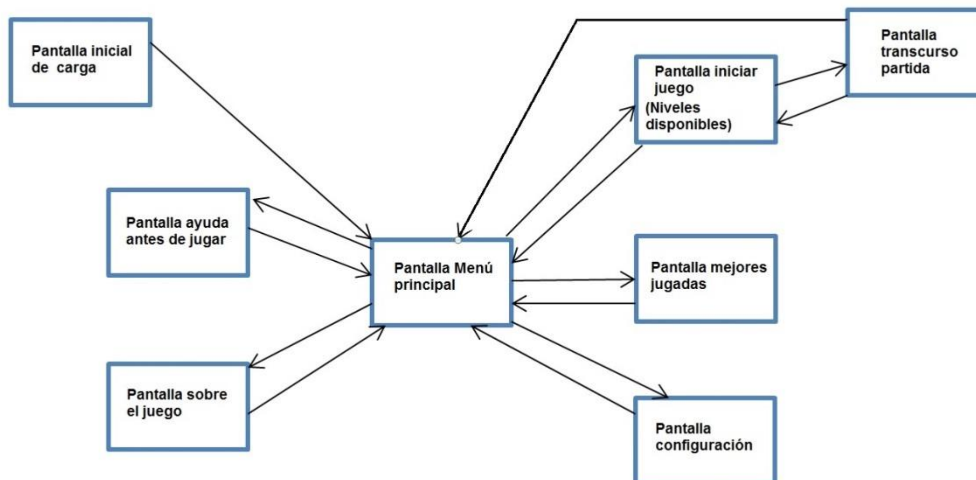


Figura 10: Diagrama de transición entre pantallas

3.3 Diseño de datos

El video juego necesita persistir en el dispositivo los siguientes datos:

- Lista con las diez mejores jugadas.
- El nivel máximo disponible.

Dada la sencillez de los datos a persistir, el videojuego no requiere de estructuras de datos complejas ni de un esquema de base de datos.

3.4 Diseño de la arquitectura

Como dije anteriormente, dada mi inexperiencia en el desarrollo de videojuegos, antes de empezar con la implementación, no diseñé prácticamente nada, sino que empecé desde la implementación mínima funcional, y desde ahí fui añadiendo funcionalidades.

La arquitectura de dicha implementación mínima funcional es la siguiente:

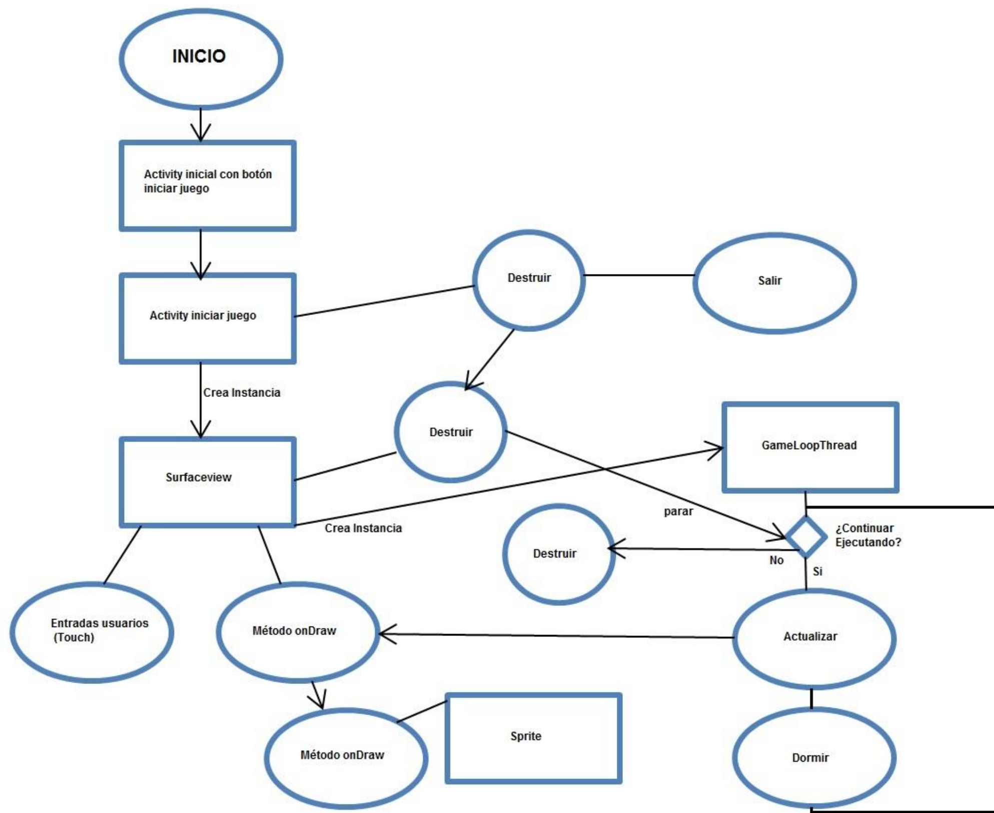


Figura 11: Arquitectura básica funcional

4. Desarrollo

Una vez establecida las especificaciones de diseño, tengo claro qué debo desarrollar, e inicio la implementación del videojuego. Esta fase ha sido la más costosa, ya que he tenido que ir aprendiendo a programar un videojuego sobre la marcha.

En el presente capítulo expondré la lógica de funcionamiento de la codificación. Comentando y explicando los aspectos más relevantes de la codificación, sólo a nivel funcional, no entraré en detalles de código. Éste está generosamente comentado, creo que se puede entender y seguir la lógica de funcionamiento con facilidad.

4.1 Entorno de desarrollo utilizado

Los recursos software necesario fueron nombrados en el apartado “1.4.1 Recursos necesarios”, aunque los vuelvo a recordar:

- IDE de Eclipse: entorno de trabajo desde el que edito el código fuente, compilo y ejecuto.
- JDK: interprete de Java, necesario para programar utilizando el Framework de Android.
- SDK: kit de desarrollo de Android.
- Plugin ADT: se instala desde Eclipse.

Una vez instalado y configurado el entorno de desarrollo, comenzamos con las labores de codificación.

4.2 Inicio del desarrollo

Como dije anteriormente, apliqué la metodología o principio de “**desarrollo ágil de software**”. Donde tuviese rápidamente una versión inicial ejecutable y con algún tipo de resultado. Y desde esa base inicial, mediante iteraciones, añadir funcionalidades hasta obtener el producto inicial.

Para implementar la base inicial, busqué y leí por internet cual es el “game loop” recomendado [6] [7]. Las recomendaciones coincidían en las diferentes fuentes consultadas, así que las tomé en cuenta y las implementé. La base inicial, a groso modo consistía en lo siguiente:

- Una pantalla inicial implementada mediante una *Activity*, con un botón que al pulsarlo transiciona a una segunda *Activity.*
- La segunda *Activity* contiene una clase que extiende de *SurfaceView*, esta representa la pantalla del juego, y se utiliza para dibujar en ella. Entre otros métodos contiene el método *OnDraw* para dibujar en la

pantalla, y el método *onTouchEvent* para capturar las pulsaciones realizadas sobre la pantalla. Esta clase, a su vez instancia otra que extiende de *Thread*, y es la que implementa el bucle de ejecución del juego, implementado en el método *run*.

Esta es la base de implementación tomada, y a partir de ella, fui añadiendo funcionalidades.

4.3 Representación visual de funcionalidades y elementos del código

A continuación incluyo una imagen en la que podemos visualizar los elementos de codificación del videojuego:

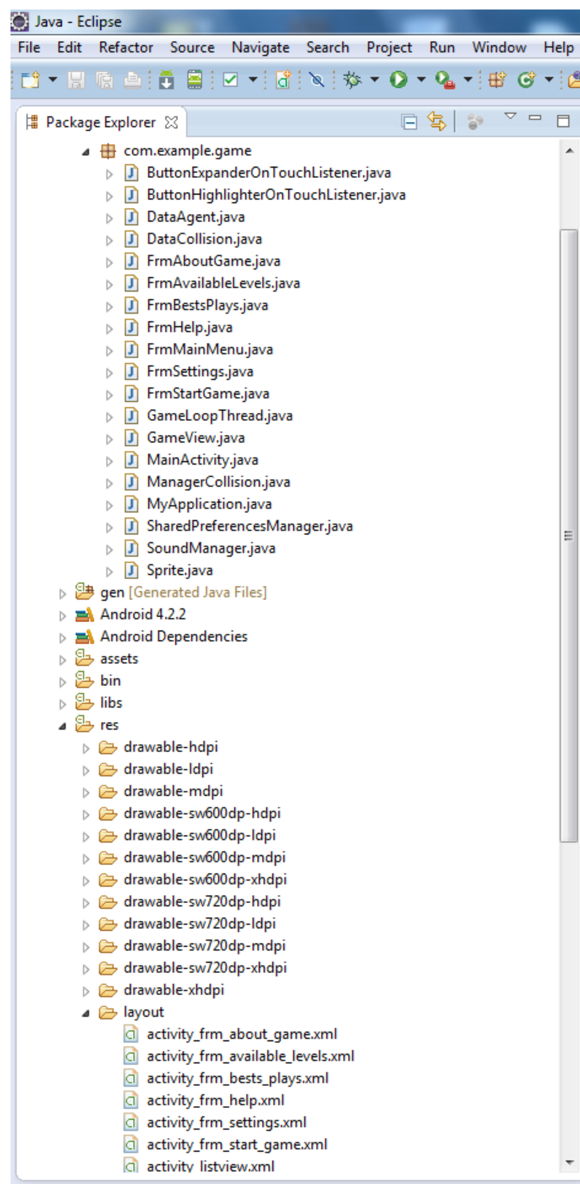


Figura 12: Elementos de codificación

Seguidamente, en otra imagen muestro la relación entre algunos elementos del código y las funcionalidades (o más bien, pantallas del videojuego). Es para hacernos una idea, rápida y visual, sobre qué elementos del código implementa cada funcionalidad o pantalla.

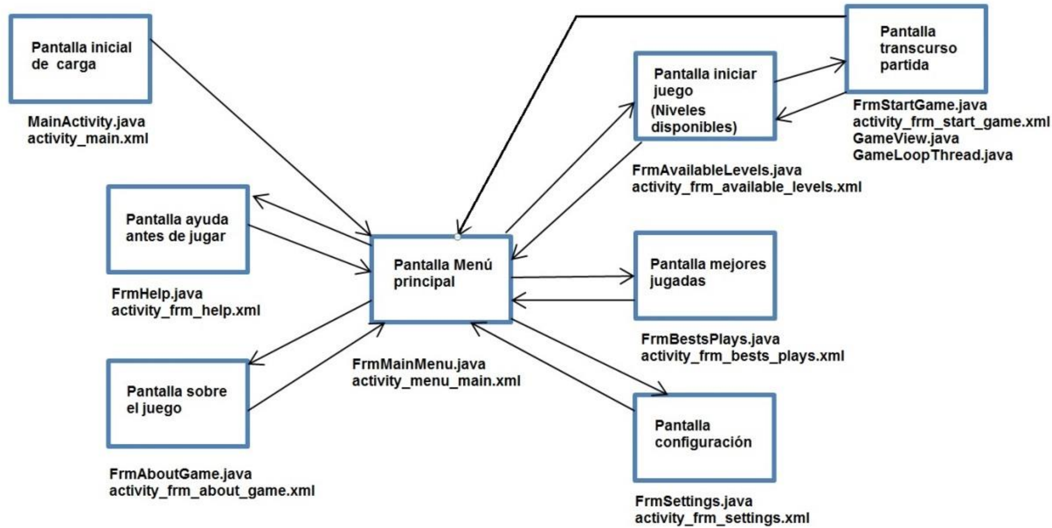


Figura 13: Pantallas y elementos de la codificación

4.4 Lógica de funcionamiento

Explicaré de forma general, sin entrar en detalles –los detalles los daré en el apartado *Codificación*-, cual es la lógica de funcionamiento del código implementado:

El punto de inicio de ejecución es la actividad *MainActivity*, esta inicializa datos que serán utilizados posteriormente y carga las palabras válidas de la lengua inglesa a memoria; mientras se están cargando, muestra en pantalla el mensaje de “Cargando”. Una vez cargadas las palabras a memoria, pasa automáticamente a la siguiente actividad, *FrmMainMenu.*

Esta muestra en pantalla las diferentes opciones ofrecidas por el videojuego, redireccionando a la opción que se elija, estas pueden ser las siguientes:

- *Iniciar juego:* muestra una pantalla con los niveles disponibles actualmente, esta pantalla es representada gráficamente por el elemento de codificación *activity_frm_available_levels.xml.* Es el encargado del aspecto visual de dicha pantalla. Y la lógica de funcionamiento es implementada por la clase *FrmAvailableLevels.java*. Una vez seleccionado el nivel por el que deseamos comenzar la partida, cambia a la pantalla que representa o visualiza la partida del juego. Esta pantalla es representada gráficamente por el elemento *activity_frm_start_game.xml.* La lógica de funcionamiento es implementada mediante las clases *FrmStartGame.java*, *GameView.java*

y *GameLoopThread.java.* El funcionamiento de cada una de estas clases se comentará en el apartado *Codificación.*

- Mejores jugadas: muestra en pantalla una lista con las 10 mejores jugadas, clasificadas de mayor a menor según la puntuación obtenida. Es representada por el elemento *activity_frm_bests_plays.xml,* y la lógica de funcionamiento implementada por la clase *FrmBestsPlays.java*.

- Configuración: en esta pantalla activamos/desactivamos el sonido del videojuego, como sabemos, disponemos de dos tipos de sonidos: sonido ambiental y sonido de efectos. La pantalla es representada gráficamente por el elemento *activity_frm_settings.xml*, y la lógica de funcionamiento la implementa la clase *FrmSettings.java.*

- Ayuda antes de jugar: muestra el texto explicando en qué consiste y cómo jugar al juego. Representada gráficamente por *activity_frm_help-xml*, y la lógica implementada por *FrmHelp.java*.

- Sobre el juego: muestra información propia del videojuego, como el nombre, la versión y el tipo de licencia. Representación gráfica *activity_frm_about_game.xml*, y la lógica *FrmAboutGame.java*.

Todas las pantallas disponen de un botón “*volver*” -excepto la pantalla inicial de ejecución, pantalla con el menú principal, y la pantalla de la partida-, para volver a la pantalla anterior. Esta funcionalidad ya es ofrecida implícitamente por el SO de Android, mediante el botón “*back*”, aún así he creído conveniente incluir dicha función en las pantallas. Así el usuario la tiene presente de forma explícita en cada pantalla, y no tiene que saber o intuir que puede hacer lo mismo desde el botón “*back*” del dispositivo.

4.5 Aspectos más relevantes de la codificación

A continuación comento cómo he resuelto las tareas o funcionalidades más determinantes o influyentes en la elaboración del videojuego. La explicación será en lenguaje natural, los aspectos técnicos sobre la codificación pueden verse en detalle en el código del proyecto.

4.5.1 Menú del juego

La pantalla principal con el menú del juego, y las restantes pantallas que representan a las diferentes opciones; su lógica de funcionamiento es implementada mediante clases que extienden de *Activity* (interfaz ofrecida por Android para implementar pantallas o ventanas), y la parte gráfica es implementada en un fichero XML; veámoslo con algo más de detalles:

En la representación de una pantalla, se hacen dos distinciones:

- Parte gráfica: es donde se definen todos los aspectos gráficos de la pantalla; características propias (ancho, alto, fondo…) y elementos

gráficos que tienen (como etiquetas, botones, etc.). Ello se define en un fichero XML, dentro del directorio *layout*, contenido en la estructura de directorios de un proyecto Android.

- Parte lógica: es donde se programa su funcionalidad, es decir, qué hace dicha pantalla. Ello se define en una *Activity* contenida en un fichero ".java", dentro de algún paquete y en el directorio *src.*

Por lo tanto, cada pantalla tiene asociado dos ficheros de los tipos comentados anteriormente.

La pantalla que representa al menú principal, contiene un botón por cada opción del menú. Al pulsar sobre uno de ellos, transiciona a la pantalla que representa a la opción seleccionada.

4.5.2 Bucle del juego

Considero que es la esencia del videojuego, y todo lo demás gira en torno a él. El bucle está implementado en la clase *GameLoopThread,* y se apoya en otras dos: *GameView* y *FrmStartGame.*

El bucle de ejecución del juego se implementa en el método *run*, y la lógica de funcionamiento es la siguiente:

- La frecuencia de ejecución es de 10 veces por segundo, es decir, los FPS son 10.

- Lo primero que comprueba es si el juego está pausado, en caso de estar pausado, no hace nada, si no está pausado, modifica el estado del juego.

- A continuación comprueba si debe crear un agente letra, ello lo hace mediante un contador de tiempo, si este ha llegado o superado al tiempo mínimo establecido para crear agentes, crea un nuevo agente y reinicia a 0 dicho contador, en caso contrario, volverá a comprobarlo en la siguiente pasada del bucle.

- Luego, actualiza la pantalla, es decir, para ese instante pinta a todos los agentes letras contenidos en la pantalla en la posición que corresponda. Para pintar hace uso del método *onDraw* de la clase *GameView*, y esta a su vez utilizada el método *onDraw* de la clase *Sprite*, para cada agente contenido en la pantalla.

- Calcula cuanto debe dormir para que se complete la vuelta en el tiempo establecido, y no antes. Esto es esencial para marcar el ritmo de ejecución del bucle, y en consecuencia el del juego. Supongo que una vuelta del bucle siempre se completa antes del tiempo establecido (100 milisegundos), por ello, siempre debe dormir el tiempo que le falte hasta completar 100 milisegundos, o en caso de excederlo, al menos duerme

10 milisegundos. Está lógica para controlar la frecuencia va bien si se cumple la suposición anterior. En mi caso hice pruebas, y una vuelta se completaba en 7 milisegundos. Esta prueba la hice desde el emulador virtual, es decir, utilizando la capacidad de procesamiento del ordenador, no de un Smartphone -Aunque se supone que en la emulación también simula la capacidad de procesamiento del dispositivo-. Aún así, con el tiempo anterior, si el dispositivo es 10 veces más lento o incluso 100 (cosa que dudo, porque cuando probaba el videojuego en el dispositivo real de pruebas, iba mucho más fluido que en el emulador virtual), aún estaría dentro del margen de ejecución supuesto… En el caso de que no estuviese, lo peor que puede ocurrir es que no cumpla con la frecuencia establecida y vaya un poco más lento de 10 ejecuciones por segundo. Pero esos límites ya los pone el propio dispositivo. En mi caso, lo he probado con varios dispositivos, alguno de gama media, y en todos va fluido.

- Vuelve a ejecutar el bucle.

Cómo contabilizo el tiempo:

Programo una tarea para que se ejecute cada segundo, y en la ejecución de dicha tarea, con periodicidad de 1 segundo, incremento en una unidad el contador utilizado para contabilizar el tiempo.

Pausar el juego:

La pausa se realiza a través de un botón situado en la esquina inferior izquierda de la pantalla. Al pulsarlo se ejecuta el método *OnClick* asociado a dicho botón, en él indicamos el estado de *pausa*, y realizamos otras operaciones como: parar la reproducción de la música de fondo, desactivar los controles de interacción del juego, y mostrar un submenú con las opciones de *Continuar* y *Salir*.

El bucle de ejecución del juego, lo primero que hace es comprobar si el juego está pausado, al pausarlo hemos indicado el estado de *pausa*, ello es detectado al inicio del bucle, y este no hace nada. Po lo tanto, no modifica el estado de la pantalla y esta se queda inmóvil.

Otro punto relevante donde se comprueba si el juego está pausado, es en la actividad que utilizo para contabilizar el tiempo, en ella hago dicha comprobación antes de hacer nada. Y en caso de estar pausado no hace nada, es decir, no contabiliza el tiempo que transcurra en estado de *pausa*.

Para revertir el estado de *pausa* y volver a la actividad en el juego, podemos hacerlo mediante el propio botón de pausa o a través del botón *Continuar* que se mostró en la pantalla cuando se inició la *pausa*. Ambos tienen el mismo efecto. Y es el siguiente: indica que se ha salido del estado de *pausa*, vuelve a reproducir la música de fondo, activa los controles de interacción del juego y oculta el submenú asociado al estado de *pausa.*

Con ello el bucle del juego, detecta que se ha salido del estado de *pausa* y retoma la actividad, lo mismo ocurre en la actividad utilizada para contabilizar el tiempo.

Paso de niveles:

La lógica para pasar de un nivel a otro está implementada en la clase *GameLoopThread*. Pasamos de un nivel a otro cuando llegamos a los 120 segundos sin exceder el número máximo de agentes letras permitidos en pantalla.

Para detectar si paso de nivel hago periódicas comprobaciones sobre el tiempo de partida transcurrido, dichas comprobaciones las realizo en la actividad antes mencionada utilizada para contabilizar el tiempo, y que se ejecuta de manera automática cada segundo. En ella compruebo si he llegado a los 120 segundos, en tal caso, indico que el bucle de ejecución del juego debe parar, y el motivo de parada.

Posteriormente, el bucle de ejecución del juego detecta que debe finalizar, cesa en su ejecución y comprueba cuál ha sido el motivo de parada. En este caso, la finalización de un nivel; e invoca al método que se encarga de procesar todas las tareas correspondientes a la finalización de un nivel, como:

- Parar la música de fondo, dejar de contabilizar el tiempo e inhabilitar todos los controles de interacción.

- Mostrar en pantalla un mensaje para indicar que el nivel se ha superado, con el correspondiente efecto sonoro.

- Escribir los puntos totales conseguidos hasta el momento, contabilizar el número de letras que han quedado en pantalla, restar un punto por cada una de ellas y mostrar los puntos totales tras la resta anterior.

- Comprobar si debe incluir la jugada entre las diez mejores, y en tal caso incluirla.

- Comprobar si el siguiente nivel es mayor que el nivel máximo disponible, y si es así, indicar dicho nivel como nivel disponible para iniciar partidas desde él.

- Indico en pantalla mediante un mensaje el inicio del próximo nivel y comienza el siguiente nivel, volviendo a ejecutar el bucle de ejecución del juego.

Perder una partida:

La lógica es prácticamente igual que la anterior, sólo difiera la comprobación a realizar y el motivo de parada del bucle del juego. Veamos la lógica:

La lógica para detectar que se ha perdido una partida está implementada en la clase *GameLoopThread.* Perdemos una partida cuando superamos el número máximo de agentes letras permitidos en pantalla simultáneamente.

Para detectar tal situación, hago periódicas comprobaciones sobre el número de agentes letras acumulados en pantalla, dichas comprobaciones las realizo en la actividad utilizada para contabilizar el tiempo, y que se ejecuta de manera automática cada segundo. En ella compruebo si se cumple la condición para perder una partida, en tal caso, indico que el bucle de ejecución del juego debe parar, y el motivo de parada.

Posteriormente, el bucle de ejecución del juego detecta que debe finalizar, cesa en su ejecución y comprueba cuál ha sido el motivo de parada. En este caso, la pérdida de la partida; e invoca al método que se encarga de procesar todas las tareas correspondientes a la pérdida de una partida, como:

- Parar la música de fondo, dejar de contabilizar el tiempo e inhabilitar todos los controles de interacción.

- Mostrar en pantalla un mensaje indicando que se ha perdido la partida, con sus correspondientes efectos sonoros.

- Hace desaparecer a los agentes letras que quedaron en pantalla.

- Escribe los puntos totales conseguidos.

- Comprobar si debe incluir la jugada entre las diez mejores, y en tal caso incluirla.

- Dirige al juego a la pantalla con el menú principal.

<u>Finalizar el juego:</u>

Una vez detectado que se ha superado un nivel y se invoca al método que se encarga de procesar todas las tareas correspondientes a la finalización de un nivel; lo primero que se hace en dicho método es comprobar si se ha superado el último nivel, es decir, el nivel diez. En tal caso, hemos llegado al final del juego, y debe ejecutarse el método correspondiente, este método realiza las siguientes tareas:

- Parar la música de fondo, dejar de contabilizar el tiempo e inhabilitar todos los controles de interacción.

- Inicia la reproducción de la música correspondiente al final del juego.

- Hace desaparecer a los agentes letras que quedaron en pantalla.

- Escribe en pantalla un mensaje, dando la enhorabuena.

- Escribe los puntos totales conseguidos.
- Comprobar si debe incluir la jugada entre las diez mejores, y en tal caso incluirla.
- Dirige al juego a la pantalla con el menú principal.

Las comprobaciones anteriores para detectar cuando superamos un nivel o perdemos una partida, inicialmente, puede parecer que el mejor lugar para realizarlas es al final de una ejecución del bucle. Y así, determinar si debemos parar su ejecución o iniciar una nueva iteración. Pero si lo hiciésemos de dicha manera, comprobaríamos las condiciones diez veces por segundo, que es la frecuencia con la que se ejecuta el bucle. Creo que sería excesivo e innecesario, y podría perjudicar sobre la fluidez de ejecución. Por ello, realizo las comprobaciones en la actividad programada para contabilizar el tiempo, así se comprueba una sóla vez cada segundo. Creo que ésta si es una buena frecuencia de comprobación.

Relaciones con las clases GameView y FrmStartGame:

Las clases *FrmStartGame*, *GameView* y *GameLoopThread* son las que implementa básicamente el transcurso de una partida, e interaccionan entre ellas para conseguir tal fin. La clase *FrmStartGame* es la *Activity* donde transcurre, y es el vínculo de acceso a los recursos gráficos contenidos en la pantalla. La clase *GameView* es la encargada de pintar en la pantalla donde transcurre la partida, a través de ella, pinto a los agentes letras. Y la clase *GameLoopThread*, como ya sabemos, contiene el bucle de ejecución del juego. Estas son sus funciones más relevantes, veremos algunas más en el próximo apartado cuando comente el funcionamiento de cada clase.

4.5.3 Los agentes letras

A continuación comentaré los recursos y lógica utilizada para representar a los agentes letras que aparecen y se mueven por la pantalla. Comentaré cómo los represento, la lógica implementada para hacer que aparezcan de forma aleatoria, cómo se mueven y la lógica desarrollada para detectar colisiones.

Cómo los represento:

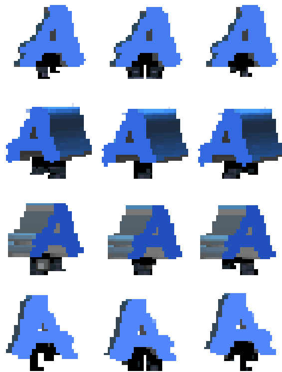


Figura 14: Agente letra azul

Figura 15: Agente letra rojo

Cada agente letra es representado mediante dos imágenes con extensión ".png"; una para cada color. El color por defecto es azul, y cuando los seleccionamos cambian al color rojo.

Cada imagen tiene 4 filas, cada fila representa una de las cuatro posibles direcciones en las que el agente se puede desplazar. Y 3 columnas, éstas representan las posiciones en las que puede estar el agente en cada dirección.

Las imágenes han sido creadas "manualmente", es decir, no he utilizado ningún recurso que de forma directa me genere cada imagen. Para su elaboración he utilizado las siguientes herramientas o aplicaciones:

- *Xara 3D Maker,* esta aplicación genera letras con aspecto tridimensional. Con ella he generado el cuerpo de cada letra, utilizando diferentes perspectivas para cada fila.

- *Paint,* una vez generadas las letras para cada fila, he utilizado ésta aplicación para colocarlas en forma de matriz y añadirle las patitas a cada una de ellas. Una vez que las tenía en azul, para obtenerlas en rojo, simplemente le pintaba el cuerpo en rojo.

El proceso de creación ha sido "manual" y con un toque de artesanía, ya que cada letra ha sido generada individualmente y con un tope un poco personal. No son todas exactamente iguales con las patitas uniformes.

Busqué herramientas para generar Sprites de forma automática, y ahorrarme el laborioso proceso anterior, pero no encontré ninguna que se ajustase a mis necesidades, así que tuve que hacerlo mediante el proceso descrito anteriormente.

Generación aleatoria:

He supuesto que en las palabras de la lengua inglesa, de forma general, el 45% de éstas están formadas por vocales, y el 55% restante por consonantes. Creo que es una suposición representativa de la realidad.

Partiendo de la suposición anterior, he implementado la lógica para que los agentes letras aparezcan en pantalla con los porcentajes de probabilidades mencionados. Para implementarla hago uso de lo siguiente:

Utilizo una tabla de 26 posiciones, cada posición representa una de las 26 letras del alfabeto de la lengua inglesa. Las 5 primeras posiciones las utilizo para las vocales, y las restantes para las consonantes. Ordenadas alfabéticamente, aunque el orden no influye en nada.

Hago uso de la función *Random* para generar números aleatoriamente entre 0 y 999. Para cada generación hago la siguiente clasificación:

- Si el número está comprendido en el intervalo [0, 450), debo generar una vocal.
- Sino (estará en el intervalo [450, 999]), genero una consonante.

Mediante lo anterior consigo generar letras de forma aleatoria, de las cuales el 45% serán vocales y el 55% consonantes

Una vez clasificadas entre vocales y consonantes, debo volver a generar aleatoriamente una vocal o una consonante. Para ello vuelvo a utilizar la función *Random* generando números aleatorios entre [0, 4] para las vocales, y entre [5, 25] para las consonantes. Dicho número lo utilizo como índice de la tabla mencionada anteriormente, y al acceder a la posición de la tabla indicada por el índice, ésta me determina qué letra concreta debo generar.

Cómo se mueven:

Los dos aspectos claves en el movimiento de los agentes letras son: las direcciones de movimiento y la velocidad de desplazamiento. Veámoslo con más detalles:

Direcciones de movimiento: La pantalla se representa mediante los ejes cartesianos X e Y, con una leve anomalía, Y positivo hacia abajo, Y negativo hacia arriba. La X igual que por normal general.

El desplazamiento se consigue con variaciones en el eje X e Y. Estas variaciones tendrán siempre la misma magnitud para los dos ejes, es decir, si varío en una unidad la coordenada del eje X, también vario en una unidad la coordenada del eje Y. Por lo tanto, el movimiento siempre será en diagonal.

Tenemos 4 diagonales posibles de movimiento, en consecuencia, 4 direcciones posibles de movimiento, y para cada dirección de movimiento muestro al

agente letra en una perspectiva diferente. Esto último lo consigo utilizando las 4 filas de la matriz imagen, una para cada dirección de movimiento.

Las variaciones en los ejes de coordenadas determinan en la dirección que se mueve, y a qué velocidad se mueven. La dirección viene determinada por el signo, y la velocidad por la magnitud. A continuación, detallo las 4 direcciones posibles en función al signo de las variaciones:

- (+, -) incremento positivo en el eje X y negativo en el eje Y, iría hacia arriba, representado por la fila 3 de la matriz imagen.
- (-, -), iría hacia la izquierda, representado por la fila 1.
- (+, +), iría hacia la derecha, representado por la fila 2.
- (-, +), iría hacia abajo, representado por la fila 0.

Inicialmente, cuando creo al agente, comienza con incrementos positivos en ambos ejes, por lo tanto, siempre inicia el movimiento hacia la derecha. Luego irá variando en función a las colisiones, que las veremos en el próximo punto.

Velocidad de desplazamiento: Hemos dicho anteriormente que el desplazamiento se consigue con variaciones en el eje X e Y. Pues bien, la velocidad depende de la magnitud de dichas variaciones. La magnitud, y por tanto la velocidad de movimiento es directamente proporcional al nivel del juego. A mayor nivel, mayor velocidad de movimiento.

Detección de colisiones:

Los agentes están expuestos a colisiones de dos tipos: con los límites de la pantalla y entre ellos. Explicaré la lógica utilizada para detectar ambos tipos de colisiones, y en caso de colisionar, corregir el movimiento.

Colisiones con los límites de la pantalla: En la pantalla existen cuatro límites, dos determinados por el ancho (laterales) y otros dos por el alto (superior e inferior).

Un agente letra habrá llegado a los límites laterales, cuando el valor de su coordenada X sea cero o menor que cero (límite lateral izquierdo), o su coordenada X sea igual o superior al ancho de la pantalla (límite lateral derecho). En ambos caso para corregir el movimiento debo invertir el sentido en el eje X, es decir, las variaciones tendrán la misma magnitud pero con signo inverso.

Un agente letra habrá llegado a los límites superior e inferior, cuando el valor de su coordenada Y sea cero o menor que cero (límite superior), o su coordenada Y sea igual o superior al alto de la pantalla (límite inferior). En ambos caso para corregir el movimiento debo invertir el sentido en el eje Y, es decir, las variaciones tendrán la misma magnitud pero con signo inverso.

Estas comprobaciones las realizo para cada iteración del bucle del juego, es decir, 10 veces por segundo. Como el procesamiento para detectar colisiones

con los límites de la pantalla es sencillo, lo he dejado con dicha frecuencia, no creo que influya en la fluidez de ejecución del juego.

Colisiones entre agentes: El procesamiento para detectar colisiones entre agentes letras es más complejo y pesado que el anterior, en consecuencia, he reducido la frecuencia de comprobación. Lo compruebo cada tres iteraciones del bucle del juego.

Para comprobar si dos agentes colisionan, debo hacerlo con todas las combinaciones de agentes posibles. Ello se hace de la siguiente manera:

Supongamos que en la pantalla tenemos tres agentes: a1, a2 y a3. Pues las posibles combinaciones entre ellos serían: (a1, a2), (a1, a3) y (a2, a3). Esta lógica la adapto al número de agentes que haya en pantalla.

Cada agente es representado mediante una imagen contenida en un polígono rectangular. Y he concluido que dos agentes colisionan entre sí, cuando al menos un vértice del polígono que representa a uno de ellos está contenido dentro del área que ocupa el polígono del otro agente. Esta es la comprobación que utilizo para determinar si dos agentes colisionan. En caso de detectar colisión entre dos agentes aplico la siguiente lógica:

Obtengo y guardo su instante de colisión y compruebo si la colisión entre ellos es una colisión que deba corregir, y debo corregir una colisión cuando no esté entre las últimas 50 colisiones (utilizo una tabla donde almaceno las últimas 50 colisiones), y en caso de estar, que haya superado el *tiempo mínimo de colisión establecido para el nivel actual*.

Puede ocurrir que al generar un nuevo agente letra, aparezca en la pantalla en la misma posición en la cual ya existía otro. En este caso más que una colisión es un solapamiento, estos casos son susceptibles de crear complicaciones como bucles. Ya que al detectar que colisionan corregiremos su movimiento, y mientras se están alejando pero aún siguen solapados podemos volver a comprobar la colisión, y dará positiva, corrigiendo de nuevo su movimiento. Esto probablemente ocasione un bucle (en las pruebas lo he comprobado).

Por ello, al corregir una colisión debemos esperar un tiempo (tiempo mínimo de colisión establecido para el nivel actual) antes de intentar volver a corregir la misma colisión, el necesario para que se separen y no estar comprobando y corrigiendo de forma cíclica. Dicho tiempo de espera va decreciendo a medida que se avanza en los niveles, ya que la velocidad de movimiento es mayor y se separan antes.
Las colisiones las corrijo modificando la dirección de movimiento de uno de los agentes. La lógica me llevó a deducir que lo ideal es invertir la dirección de ambos agentes para que se repelan y alejen de forma óptima. Pero esta forma es susceptible de provocar bucles, ya que si estamos en un solapamiento y hemos invertido sus direcciones de movimiento, como se detecte una nueva colisión entre los mismos agentes y se dé por válida, volveríamos a invertir sus direcciones, provocando que se vuelvan a acercar y probablemente entrando en un bucle. Por ello, sólo modifico el movimiento de uno de los agentes.

Evitar las colisiones de forma radical creo que es casi imposible, ya que como comenté antes, algunos agentes aparecen en la pantalla superpuesto encima de otro, y éste es un caso diferente a una simple colisión. Las colisiones las corrige bastante bien, detecta que han colisionado y modifica la dirección de movimiento de uno de los agentes. Y cuando son solapamientos, también los corrige bien, uno de los agentes tiende a alejarse del otro hasta que se separen completamente.

La lógica para detectar colisiones entre agentes la he deducido e implementado completamente yo. Le eché una pensada, no lo vi muy difícil y lo intenté implementar. Tal vez, por internet existan algoritmos o lógicas más eficiente que la mía implementada, pero como he visto que los resultados son aceptables (al menos, desde mi punto de vista) la he dejado así.

4.5.4 Palabras válidas de la gramática inglesa y su uso

El juego reconoce y da por válida 58.110 palabras de la lengua inglesa. Creo que son suficientes para la magnitud y objetivos del videojuego. A continuación indico cómo hago uso de ellas.

<u>Modo de acceso a la información:</u>

Para acceder a la información disponemos de varios métodos, como crear una base de datos con la información que se necesita y hacer consultas sobre ella. Dado que la información a procesar es simple y ocupa poco espacio, no es rentable crear una base de datos.

Otro posible método podría ser tener la información en el disco duro y acceder a ella cada vez que fuese necesario, pero el acceso a disco es más lento que el acceso a memoria. En el juego, durante la partida, se realiza continuamente consultas para comprobar si la palabra actual es válida. Así que el número de consultas es un parámetro a tener en cuenta.

Por lo tanto, la mejor opción es tener la información cargada en memoria (ya que el fichero de texto con las palabras ocupa 588 KB) y acceder a ella cada vez que sea necesario.

<u>Estructura de datos utilizada para guardar la información:</u>

Inicialmente la estructura de datos utilizada para almacenar las palabras fue un *TreeSet* (árbol binario balanceado, donde la información está ordenada y las búsquedas y acceso son rápidos y eficientes). Pero tardaba demasiado tiempo en cargar las palabras desde el fichero a memoria, aproximadamente 85 segundos (el tiempo fue medido en un dispositivo virtual, en uno real tardaría algo menos). Es excesivo hacer esperar al usuario dicho tiempo para comenzar a jugar al videojuego.

Por ello, utilizo como estructura de almacenamiento un *ArrayList* (lista dinámica de objetos, en mi caso de cadenas), tarda aproximadamente ¼ del tiempo que

necesita el *TreeSet* para cargar las palabras. Ofreciendo un tiempo de espera bastante aceptable (cuando lo he probado en dispositivos reales tarda unos 5 segundos). El tiempo de espera antes de poder jugar, es un parámetro a tener muy en cuenta en los juegos para dispositivos móviles. Ya que el usuario no suele disponer de mucho cuando va a jugar una partida.

El inconveniente que presenta la estructura *ArrayList* es que las búsquedas implementadas en sus métodos nativos no son eficientes, y tarda demasiado tiempo en encontrar una palabra. En consecuencia, ello ralentizaba la ejecución del juego, iba a saltos y no se podía interactuar correctamente con él.

Para solventar dicho problema, he implementado una función que realiza búsquedas sobre el *ArrayList* eficientemente, la función es "*searchWord".* Utilizando dicha función las búsquedas son rápidas y eficientes, permitiendo una ejecución del juego fluida.

4.5.5 Persistir datos del juego

Los datos que necesita persistir el videojuego son simples y poca cantidad, ellos son los siguientes:

- *Nivel máximo disponible o alcanzado*: información utilizada para saber cuál es el nivel máximo disponible, es decir, el mayor nivel al que ha llegado el usuario.

- *Lista con las 10 mejores jugadas*: una lista con la puntuación de las 10 mejores jugadas.

Dicha información podemos guardarla como pares de datos, de la forma (clave, valor), por ejemplo: para el *nivel máximo disponible* podemos guardar el par (nivel, valor_numérico_nivel); donde *nivel* es la clave y *valor_numérico_nivel* es el valor asociado a dicha clave. En este caso *valor_numérico_nivel* indica el máximo nivel disponible.

Lo mismo para la *lista con las 10 mejores jugadas,* para cada jugada podemos guardar el par (número_jugada, puntuación); donde *número_jugada* es la clave (además ésta indica la posición en la lista) y *puntación* es el valor asociado a dicha clave. En este caso *puntación* sería un valor numérico indicando la puntuación.

La adaptación de la información a persistir a la forma de "pares de datos", se debe a que la guía para desarrolladores de Android recomienda utilizar para persistir pequeñas cantidades de datos una metodología [8] que hace uso de dicha representación de datos. Utilizando dicha metodología, puedo persistir datos en el dispositivo fácilmente, ya que ello es administrado por el Framework de Android.

4.6 Codificación

En el presente apartado comentaré brevemente los aspectos y elementos más relevantes de la codificación. Como: las clases implementadas y su funcionalidad, y la estructuración de los elementos para soportar y utilizar la lógica que ofrece el *framework* de Android para que sea compatible con los diferentes tipos y tamaños de dispositivos.

4.6.1 Clases implementadas y su funcionalidad

Nombraré cada clase y una breve descripción de la funcionalidad que implementa:

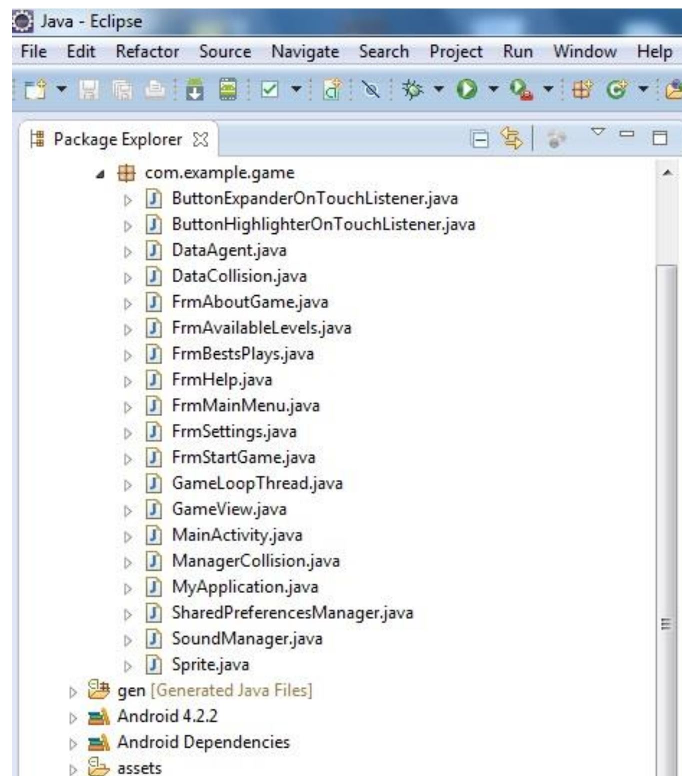


Figura 16: Clases del código

ButtonExpanderOnTouchListener

Clase utilizada para crear un efecto visual cada vez que pulsamos un botón en las diferentes pantallas del juego. El efecto consiste en expandir el texto del botón hasta 1,5 veces su tamaño original, mientras se pulsa el botón. Una vez finalizada la pulsación, el texto vuelve a su tamaño original.

ButtonHighlighterOnTouchListener

Cambia el aspecto del botón *acelerar* mientras se pulsa. Al pulsarlo cambia su imagen de fondo, y cuando finaliza la pulsación vuelve a restablecer su imagen original.

DataAgent

Almacena información sobre los *Sprites* letras, como: letra que representa el *Sprite*, vínculo a la imagen que lo representa en azul y un vínculo a la imagen que lo presenta en rojo.

DataCollision

Clase utilizada para almacenar los datos referentes a una colisión entre dos *Sprites.* Como: identificador del *Sprite* tomado como principal (posición que ocupa dentro de la lista de *Sprites*), identificador del *Sprite* tomado como secundario y el instante en el cual se produce la colisión.

FrmAboutGame

Activity que implementa la lógica de la pantalla *Sobre el juego.* Muestra información sobre el juego: nombre, versión y tipo de licencia.

FrmAvailableLevels

Activity que implementa la lógica de la pantalla *Iniciar juego.* En dicha pantalla, se muestra los niveles que pueden ser seleccionados antes de iniciar una partida. Por cada nivel conseguido o abierto, muestra un botón, y al pulsarlo inicia la partida en el nivel seleccionado. El máximo nivel alcanzado es un valor que persiste el juego en el dispositivo, por lo tanto, para determinar el máximo nivel disponible debe leerlo de la memoria del dispositivo.

FrmBestsPlays

Activity que implementa la lógica de la pantalla *Mejores jugadas.* Muestra una lista con las 10 mejores jugadas, ordenadas de mejor a peor. Dicha información la persiste el juego en el dispositivo, por lo tanto, antes de mostrar la lista debe leer previamente la información del dispositivo.

FrmHelp

Activity que implementa la lógica de la pantalla *Ayuda antes de jugar.* Muestra un texto con información sobre el juego y sus reglas.

FrmMainMenu

Activity que implementa la lógica de la pantalla *Menú principal*. En el menú principal disponemos de un botón por cada una de las diferentes opciones.

Ésta *Activity* identifica el botón pulsado, realizando la transición a la pantalla que corresponda.

FrmSettings

Activity que implementa la lógica de la pantalla *Configuración.* En dicha pantalla disponemos de dos botones, para activar o desactivar los dos tipos de sonidos: sonidos de efectos y sonido ambiental. La Activity identifica el botón pulsado, y para cada pulsación cambia el aspecto del botón y el estado del sonido en función de su estado actual. Es decir, si el sonido está activado lo desactiva y cambia el aspecto del botón a “Off”, y viceversa.

FrmStartGame

Activity que implementa parte de la lógica de la pantalla *Pantalla juego*. Dicha pantalla es donde transcurre la partida.

Entre las funcionalidades que implementa, destacan las siguientes:

- *Botón pause*: implementa la lógica del botón *pause*. Detecta la pulsación sobre éste, y comprueba el estado actual, en función a él activa el estado de *pausa* o lo desactiva. Por ejemplo, si el estado actual es “no pausado”, cambia dicho estado a “pausado”, para la reproducción de la música de fondo, cambia el aspecto del botón *pause*, desactiva los controles de interacción (botón *display* y *acelerar*) y muestra el submenú asociado al estado de *pausa*.

- *Botón acelerar*: implementa la lógica del botón *acelerar*. Detecta la pulsación sobre éste, e invoca al método *createAgent();* contenido en la clase *GameView* y utilizado para crea un nuevo agente en pantalla.

- *Botón display*: detecta la pulsaciones sobre el *display*, y para cada pulsación invoca al método *checkWord()*; contenido en la clase GameView. Dicho método comprueba si la palabra actual es válida en la lengua inglesa; si es así la valida eliminando los agentes letras que correspondientes y sumando los puntos que procedan. En caso contrario, borra la palabra del *display.*

- *Botón continuar del submenú asociado al botón pausa*: detecta la pulsación sobre éste y reanuda el juego.

- *Botón salir del submenú asociado al botón pausa*: detecta la pulsación sobre éste y sale de la partida, volviendo a la pantalla *Menú principal*.

- *Permitir la interacción con los elementos de control:* implementa métodos que permiten la interacción y cambiar el estado de los elementos gráficos (*textView*) del HUD y de otros elementos gráficos utilizados con fines informativos.

GameLoopThread

Clase que implementa al hilo con el bucle de control y ejecución del juego, por lo tanto, extiende de la clase *Thread.* Entre sus funcionalidades destacan las siguientes:

- *Tarea programada para contabilizar el tiempo:* programa una tarea para que se ejecute cada segundo, en ella contabiliza el tiempo y comprueba si se cumple alguna condición de parada del bucle de ejecución.

- *Bucle de ejecución del juego:* La funcionalidad del bucle la he comentado con en el punto 4.5.2.

- *Comprobación de condiciones para detener la ejecución del bucle:* comprueba si se cumple una de las dos condiciones para detener la ejecución del bucle. En dicho caso, indica que debe para el bucle junto con el motivo de la parada. Las condiciones para detener el bucle son:

    - Que haya transcurrido el tiempo de duración de un nivel.

    - Que se haya superado el número máximo de agentes permitidos en pantalla.

- *Superación de un nivel y paso al siguiente:* esta situación ocurre cuando ha transcurrido el tiempo de duración de un nivel sin exceder el número máximo de agentes permitidos. La funcionalidad está implementada en el método *finishedLevel()*, cuya funcionalidad ha sido comentada en el punto 4.5.2.

- *Perdida de un nivel:* ocurre cuando durante la partida superamos el número máximo de agentes permitidos en pantalla simultáneamente. La funcionalidad está implementada en el método *lostGame(),* y ha sido comentada anteriormente en el punto 4.5.2.

- *Finalizar el juego:* se ha llegado al final del juego, es decir, se ha superado el nivel 10. Funcionalidad implementada en el método *winGame(),* cuya funcionalidad ha sido comentada en el punto 4.5.2.

- *Inicio de un nuevo nivel:* anteriormente se superó el nivel y se ejecutó toda la funcionalidad asociada al método *finishedLevel().* A continuación se ejecuta la funcionalidad implementada en el método *startLevel(),* y es la siguiente:

    - Oculta el HUD y todos los mensajes anteriores informativos.

    - Muestra un mensaje indicando el nuevo nivel que se iniciará.

    - Inicializa algunos parámetros antes de iniciar la ejecución del nuevo nivel.

- Habilita los controles de interacción (botón *pausa*, *display* y botón *acelerar*), vuelve a mostrar el HUD con sus valores acorde al nuevo nivel, reanuda la reproducción de la música de fondo e invoca al método *run()* para iniciar la nueva partida.

- *Comprobar y actualizar el máximo nivel disponible:* dicha funcionalidad es ejecutada cuando se supera un nivel y se dispone a iniciar el siguiente. Antes de iniciar el nuevo comprueba si es superior al mayor nivel abierto y disponible hasta el momento. En tal caso, indica que el nuevo nivel que se va a iniciar es el mayor nivel disponible. Ésta funcionalidad está implementada en el método *checkForMaxLevelAvailable()*.

- *Al finalizar un nivel cuenta el número de letras acumuladas en la pantalla:* Una vez finalizado un nivel y desde el método *finishedLevel()*, se invoca al método *counterLettersLeft()*, y éste contabiliza el número de letras acumuladas en pantallas, restando un punto a la puntuación total por cada letra.

- *Dibujar el contenido de la pantalla en el momento actual:* ésta funcionalidad es implementada en el método *updateScreen()*, y lo que hace es pintar los elementos de la pantalla en el instante actual. Es decir, pinta a todos los agentes letras en la posición que corresponda para el instante actual. Para ello hace uso de un objeto de tipo *Canvas*, bloqueándolo y utilizándolo de forma exclusiva, y lo pasa como parámetro al método *OnDraw()* de la clase *GameView,* que es la que pinta realmente en la pantalla.

GameView

Clase utilizada para pintar en la pantalla donde transcurre la partida del juego, se encarga de pintar a los agentes letras en cada instante en la posición que proceda. Entre sus funcionalidades destacan las siguientes:

- *Pintar a los agentes letras en la pantalla:* pinta la pantalla con fondo blanco y sobre ella a los agentes letras en la posición que corresponda para cada uno de ellos. Ello lo hace en el método *OnDraw().*

- *Detectar pulsaciones sobre la pantalla:* detecta las pulsaciones efectuadas sobre la pantalla y comprueba si coincide con la posición de algún agente. Si coincide con la posición de algún agente lo marca como seleccionado, realizando las siguientes operaciones:

    - Comprueba que el número de agentes totales seleccionados que forman la palabra actual sea menor que 30. Si no es así, la selección no tiene efecto, en caso contrario, tiene efecto y realiza las operaciones indicadas a continuación.

- Reproduce efecto sonoro y cambia el color del agente a rojo.
- Incluye la letra del agente en la palabra actual.
- Comprueba si la palabra actual es válida, en tal caso, cambia el color del *display* a verde.

- *Detectar colisiones entre agentes:* Comprueba si hay colisiones entre agentes, y en tal caso, las corrige. La comprobación se realiza cada 3 *frames*, es decir, cada 300 milisegundos. La orden de comprobación se realiza desde el método *OnDraw()*, y el método que realiza dicha comprobación y corrección es *isCollisionSprites().* La forma de operar es la siguiente:

  - Comprueba para todos los agentes letras contenidos en la pantalla si está colisionando con algún otro.
  - En caso de colisión comprueba que sea permitida, es decir, que no esté entre las últimas 50 colisiones, y si está que haya superado el tiempo mínimo de colisión establecido -esto se comentó con más detalles en apartados anteriores-.
  - En caso de colisión permitida, modifica la dirección de movimiento de uno de ellos.
  - Almacena la colisión en la tabla de colisiones temporales.

- *Comprobar si una palabra es válida:* comprueba si la palabra actual pertenece a la lista de palabras reconocidas en la lengua inglesa, es decir, si la palabra está contenida en el *ArrayList* donde cargamos las palabras reconocidas. Dicha comprobación se realiza mediante los métodos: *wordIsCorrect(), listContainWord() y searchWord().* El funcionamiento de éstos se puede ver fácilmente sobre el código del proyecto.

- *Modificar la lista con las 10 mejores jugadas:* Dada una puntuación comprueba si debe ser incluida en la lista con las 10 mejores jugadas. En tal caso, la incluye en la posición que corresponda y reordena la lista. Los métodos que realizan dicha función son: *checkForBestPlays() y reOrder().*

- *Preparar próximo agente:* Determina cuál es el próximo agente que debe aparecer. Ello lo hace mediante generaciones aleatorias, con un 45% de probabilidad de que sea vocal y un 55% de probabilidad de que sea consonante. Para ello genera un valor numérico que consultando una tabla determina cuál es el agente letra asociado a dicho valor. Dicha funcionalidad está implementada en el método *prepareNextAgent().*

- *Crear agentes letras:* Una vez sabido el próximo agente en aparecer, debe crearlo. Para ello, consulta una tabla, la cual asocia para cada valor numérico un agente letra y toda la información necesaria para poder crearlo. Los métodos utilizados para crear agentes son *createSprites() y createSprite().*

MainActivity

Activity donde se inicia la ejecución del juego. Implementa la lógica de la pantalla *Inicial de carga.* Las principales funciones que realiza son las siguientes:

- Instancia la clase *SoundManager*, carga los sonidos de efectos que se utilizarán durante el juego en la instancia de dicha clase, y almacena la referencia en la clase *MyApplication*.

- Activa los dos tipos de sonidos: de efectos y ambiental.

- Define el tipo de fuente que se utilizará a lo largo del juego y lo almacena en la clase *MyApplication*.

- Instancia las clases *SharedPreferencesManager* y *ManagerCollision*, almacenando referencias a dichas instancias en la clase *MyApplication.*

- Carga las palabras reconocidas de la lengua inglesa a memoria. Ello lo hace mediante una tarea programada para que se ejecute cada segundo. La primera vez que se ejecute lanza la carga, y para las restantes comprueba si la carga ha finalizado, una vez finalizada desactiva la ejecución de la tarea programada y pasa a la siguiente pantalla.

ManagerCollision

Clase utilizada para gestionar las colisiones entre agentes letras. Sus funciones principales son:

- Mantener en una tabla las 50 últimas colisiones.

- Dado un nivel, determina cual es el tiempo mínimo (en milisegundos) que debe transcurrir entre dos colisiones producidas por los mismos agentes.

- Registrar una colisión en la tabla temporal de colisiones.

- Dada una colisión comprobar si es válida, es decir, que no esté entre las 50 últimas colisiones, y si está que haya transcurrido el tiempo mínimo establecido entre colisiones.

MyApplication

Clase donde almaceno referencias a instancias de otras clases, recursos y datos. Para que estén disponibles durante toda la ejecución del juego y accesibles desde cualquier lugar.

Contiene la definición de una clase estática llamada *Config*, en ella almaceno información de configuración del juego. La cual se consultará durante la ejecución del mismo.

Define otra clase estática llamada *Sounds*, para referencias los diferentes sonidos de efectos del juego, asociando a cada identificar de sonido un nombre.

SharedPreferencesManager

Clase implementada para persistir datos en la memoria del teléfono. Los datos se guardan por pares de la forma (*clave*, *valor*). Donde *clave* es una cadena identificativa y *valor* es un dato numérico de tipo entero asociado a *clave.* Provee de métodos para:

- Escribir un par (clave, valor) en memoria.
- Eliminar un par (clave, valor) de memoria, dada una clave.
- Obtener el valor asociado a una clave, dada la clave.

SoundManager

Clase programada para gestionar los sonidos de efectos y ambiental. Provee métodos para realizar las siguientes funciones:

- Añadir un sonido de efecto a la lista de sonidos disponibles.
- Reproducir un determinado sonido de efecto.
- Iniciar la reproducción de la música ambiental.
- Iniciar la reproducción de la música reproducida cuando se llega al final del juego.
- Pausar la reproducción de la música ambiental.
- Pausar la reproducción de la música de final del juego.
- Parar la reproducción de la música ambiental y liberar los recursos asociados al reproductor.
- Parar la reproducción de la música de final del juego y liberar los recursos asociados al reproductor.

- Comprobar si algún reproductor está reproduciendo, en tal caso, pausa o para su reproducción.

Sprite

Clase utilizada para representar a un agente letra. Entre sus funciones, las más destacadas son las siguientes:

- *Desplazar al agente:* cambia la posición del agente en la pantalla. Si el agente está en el extremo derecho o izquierdo de la pantalla, cambia su dirección, es decir, cambia el signo de las variaciones en el eje X. Lo mismo si el agente está en el extremo inferior o superior de la pantalla, cambia su dirección cambiando el signo de las variaciones en el eje Y. Actualiza el valor que indica dentro de la matriz imagen que representa al agente cuál columna utilizar. Todo ello se implementa en el método *update().*

- *Dibujar al agente en la pantalla:* pinta en la pantalla la imagen que representa al agente, sabiendo cuál de ellas debe elegir de la matriz imagen. Ello se implementa en el método *onDraw().*

- *Determinar la dirección de movimiento del agente:* El agente puede desplazarse en las 4 direcciones posibles (arriba, abajo, izquierda, derecha) pero siempre en diagonal. La dirección de movimiento es determinada por el signo de las variaciones en el eje X e Y. En función al signo determina en qué dirección va y qué fila de la matriz imagen representa dicha dirección. Dicha funcionalidad está implementada en el método *getAnimationRow().* Es útil para saber qué fila de la matriz imagen debe utilizar para pintar al agente.

- *Determinar si un punto colisiona con un agente:* Dado un punto determinado (coordenada X e Y), determina si el punto colisiona con el agente, es decir, está contenido dentro del área que ocupa el agente en la pantalla. Ello lo implementa el método *isCollision().*

- *Cambiar el color del agente a rojo:* Cambia el color de fondo del agente a rojo. Implementado en el método *changeColourToRed().*

- *Cambiar el color del agente al color por defecto:* Cambia el color de fondo del agente a azul. Implementado en el método *changeColourToDefault().*

- *Invertir movimiento de un agente:* Invierte la dirección de movimiento de un agente, es decir, cambia el signo de las variaciones para el eje X e Y. Implementado en el método *changeDirection().*

4.6.2 Estructuración de los elementos

Comentaré la estructuración de los elementos para soportar y utilizar la lógica que ofrece el *framework* de Android para que sea compatible con los diferentes tipos y tamaños de dispositivos.

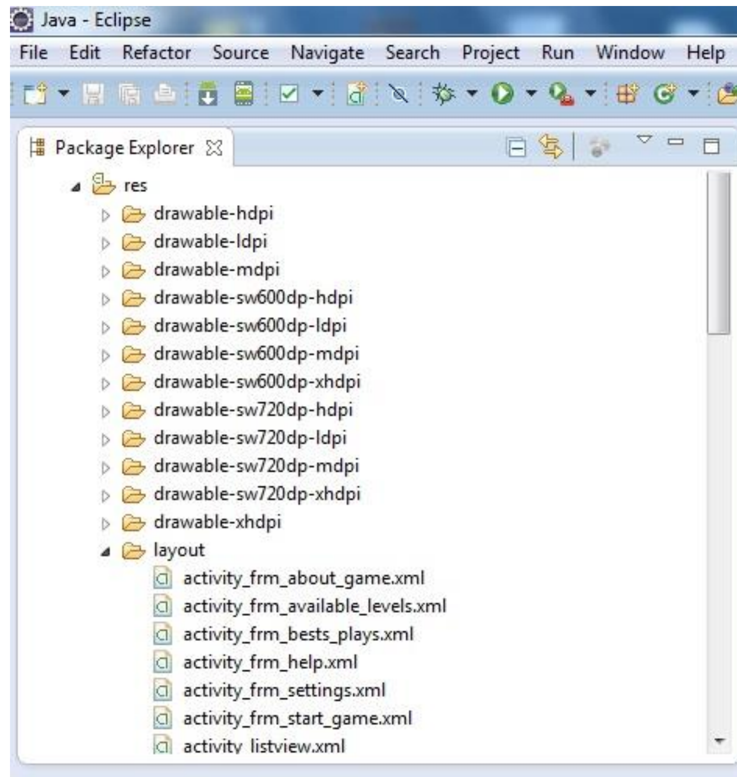


Figura 17: Otros elementos del código

La guía para desarrolladores de Android [9][10] recomienda compatibilizar las aplicaciones con los diferentes tipos de dispositivos teniendo en cuenta dos parámetros:

- Tamaño de la pantalla del dispositivo: clasifica a los dispositivos según el tamaño de su pantalla en pequeños, normales, largos y extra largos.
- Densidad de la pantalla del dispositivo: diferencia y clasifica a los dispositivos según la densidad de su pantalla en baja, media, alta y extra alta.

Teniendo ello en cuenta, clasifico los posibles dispositivos en tres grupos, según el tamaño de sus pantallas:

- Normales: desde los tamaños más pequeños posibles hasta pantallas inferiores a los 600 dp (pixeles independientes de la densidad) de ancho. Suelen ser smartphones y pequeñas tablets.

- Grandes: con tamaños desde 600 dp mínimo de ancho hasta menos de 720 dp. Para tablets de 7 pulgadas y superiores, pero inferiores de 10.

- Extra grandes: con tamaño mínimo de 720 dp de ancho. Para tablets de 10 pulgadas en adelante.

Con dichas distinciones creo que es suficiente para que el juego se adapte de forma aceptable a los diferentes tamaños de pantallas. En un principio, no hice distinción por tamaños de pantallas, y el juego estaba optimizado para una pantalla de 4,3 pulgadas. Pero cuando lo probé en una tablet de 10,1 pulgadas no se adaptaba bien a la pantalla. El tamaño de la fuente era demasiado pequeño, y la distribución de los elementos en la pantalla no aprovechaba todo el tamaño de ésta. Así que vi necesario hacer dicha distinción.

Para cada tipo de la clasificación anterior, tengo en cuenta los diferentes tipos de densidades de pantallas más comerciales; baja, media, alta y extra alta.

En consecuencia a todo lo comentado anteriormente, para conseguir que el juego se adapte a las diferentes pantallas teniendo en cuenta su tamaño y densidad, incluyo en el directorio *res* de mi proyecto los siguientes directorios:

- Para dispositivos con tamaño de pantalla normales:

    - *layout*: contiene los ficheros XML que definen la representación gráfica de las diferentes pantallas, optimizando su representación para pantallas con tamaños normales.

    - *drawable*-tipo_densidad: 4 directorios uno por cada tipo de densidad, y dentro de cada uno de ellos, las imágenes adaptadas al tipo de densidad.

    - *values*: aparte del fichero “strings.xml” con la definición de las cadenas utilizadas en el juego, contiene el fichero “styles.xml”, donde se definen estilos para los diferentes textos utilizados, indicando entre otros parámetros el tamaño de la fuente.

- Para dispositivos con tamaño de pantalla grandes:

    - *layout-sw600dp*: contiene los ficheros XML que definen la representación gráfica de las diferentes pantallas, optimizando su representación para pantallas con tamaños grandes.

    - *drawable*-sw600dp-tipo_densidad: 4 directorios uno por cada tipo de densidad, y dentro de cada uno de ellos, las imágenes adaptadas al tipo de densidad.

    - *values-sw600dp*: no es necesario volver a definir el fichero “strings.xml”, ya que el contenido de las cadenas es el mismo independientemente a tamaños y densidades de pantallas. Por ello, sólo indico el fichero “styles.xml”.

- Para dispositivos con tamaño de pantalla extra grandes:

  - *layout-sw720dp*: contiene los ficheros XML que definen la representación gráfica de las diferentes pantallas, optimizando su representación para pantallas con tamaños extra grandes.

  - *drawable*-sw720dp-tipo_densidad: 4 directorios uno por cada tipo de densidad, y dentro de cada uno de ellos, las imágenes adaptadas al tipo de densidad.

  - *values-sw720dp*: fichero “styles.xml”.

5. Pruebas

Las pruebas realizadas han consistido en comprobar que el videojuego funciona perfectamente en diferentes tipos de dispositivos, con diferentes tipos de tamaño de pantalla, densidad y capacidad de procesamiento.

En principio, durante el desarrollo realizaba las ejecuciones sobre un dispositivo virtual. Al poco tiempo, cambié y realizaba las ejecuciones sobre un dispositivo real, un Sony Xperia S. El cambio fue notorio, el juego iba mucho más fluido en el dispositivo real.

Dicho dispositivo tiene instalada la versión de Android más utilizada, *Jelly Bean,* y es un dispositivo de gama media-alta. Creo que representa el perfil de dispositivo más común en el cual se podría ejecutar el videojuego la mayoría de las veces. Por ello, he optimizado el juego para dicha versión de Android y para dispositivos como el comentado.

Aunque el juego en principio, debe funcionar correctamente en cualquier tipo de dispositivo. Para comprobarlo, he realizado pruebas en diferentes dispositivos, ellos son los siguientes:

- Smartphone Sony Xperia S con sistema operativo Android Jelly Bean 4.1.2 y una pantalla de 720 x 1280 pixeles y densidad extra alta.

- Smartphone LG Optimus L7 con sistema operativo Android Ice Cream Sandwich 4.0.3 y una pantalla de 480 x 800 pixeles y densidad media.

- Smartphone Samsung Galaxy S3 con sistema operativo Android Jelly Bean 4.1.2 y una pantalla de 720 x 1280 pixeles y densidad extra alta.

- Tablet Galaxy Tab 3 10.1 con sistema operativo Android 4.2.1 y una pantalla de 1280 x 800 pixeles y densidad media.

El resultado ha sido exitoso, el videojuego funciona y se adapta correctamente a los diferentes dispositivos, con diferentes características de capacidad de procesamiento, tamaño y densidad de pantalla.

En el vídeo de la presentación, mostraré el juego ejecutándose en los diferentes dispositivos. Además, para comprobar que se adapta a los diferentes tipos de tamaños y densidades de pantalla, hice algunas pruebas específicas. Comprobando que utiliza los recursos de representación gráfica (como layout y drawable) en función a las características del dispositivo.

Una de las pruebas consiste en comprobar que utiliza la capeta *drawable* correcta en función a la densidad de pantalla del dispositivo. Para ello, diferencio las imágenes de cada carpeta, y lo hago coloreando de un color diferente el botón *acelerar.* Utilizando los siguientes colores:

- Baja densidad: color rojo.
- Media densidad: color naranja.
- Alta densidad: color amarillo.
- Extra alta densidad: color violeta.

En el vídeo podremos comprobar cómo el videojuego utiliza unas imágenes u otras en función a la densidad de pantalla.

6. Valoración económica

Mi idea inicial era realizar el trabajo sin necesidad de hacer ninguna inversión económica. Sólo invertir mi tiempo y dedicación. Creo que lo he conseguido.

Los recursos de hardware utilizados, como el ordenador para el desarrollo y los dispositivos para las pruebas, han sido prestados por familiares.

Los recursos software utilizados para el desarrollo del videojuego son gratuitos. Otro tipo de software utilizado como software de Microsoft, he utilizado versiones que facilita la UOC para uso docente. Para la aplicación *Xara 3D Maker* he utilizado una versión de demostración temporal, gratuita. El resto de software es gratuito.

Para otros recursos del juego como ficheros de sonidos [11] y tipo de fuente utilizada [12], son recursos ofrecidos en internet de forma gratuita.

Podría intentar rentabilizar el tiempo dedicado, publicando el videojuego en la tienda de software en línea de *Google*, conocida como *Google Play.* Aunque no creo que sea muy buena idea. Ya que el objetivo de su desarrollo no era éste, y no lo he creado pensando en la competencia con el resto de videojuegos, sino en aprender el máximo posible. Mi videojuego está en clara desventaja, entre otros motivos, porque lo he implementado completamente desde cero, sin utilizar motores que faciliten el desarrollo.

Su valor es puramente docente y a nivel de conocimientos. Además lo he liberado utilizando licencias de GNU, para que pueda servir a la comunidad.

7. Viabilidad del producto

En cuanto a la viabilidad económica, dudo que el videojuego pueda aportar ingresos vendiéndolo como producto. Su fin es didáctico; aprender durante la realización del mismo. No se ha creado con ninguna pretensión económica.

En cuanto al uso como videojuego es perfectamente viable, ya que es completamente funcional y me atrevo a decir que hasta divertido. Mis familiares lo han probado y les parece divertido. Además, dado su carácter educativo es interesante jugar no sólo por el hecho de "pasar el rato", sino porque podemos practicar conocimientos de vocabulario sobre la lengua inglesa.

Además he liberado todo el proyecto: ejecutable, código y documentación. Puede servir a la comunidad para utilizarlo como entretenimiento o para fines didácticos, como referencia para elaborar un PFC, aprender sobre el desarrollo de aplicaciones para Android y para aprender algunas nociones básicas sobre videojuegos y su desarrollo.

8. Conclusiones

Comentaré las conclusiones inferidas tras la realización del proyecto, comentando algunos aspectos, éstos son:

Lecciones aprendidas:

En la realización de un proyecto de considerable envergadura, un factor importantísimo para el éxito y obtener el producto para la fecha prevista, es realizar una buena planificación e ir cumpliendo con todos los hitos en el tiempo estimado. Para realizar una buena planificación es imprescindible hacer una precisa estimación de la magnitud del resultado que podemos obtener en función al tiempo.

Una vez estimada la magnitud, dividir el trabajo en tareas y planificarlas en el tiempo. Esta creo que es la esencia para conseguir realizar el proyecto y no frustrarnos en el intento. Dicha esencia, ya me resulta familiar de las asignaturas varias sobre “gestión de proyectos”. Pero hasta que no realizas uno, no eres consciente de su importancia, y sueles tender a centrarte sobre todo en la “fase de codificación”.

En mi caso personal, siempre intento hacer las cosas lo mejor que puedo, y muchas veces incido más en “lo mejor” que en lo que “puedo”. Es decir, que las hago mejor de lo que realmente puedo con la consecuencia de no cumplir plazos, es decir, excediendo en el tiempo estimado para la tarea. He aprendido que las cosas siempre se pueden hacer mejor, pero hay que hacerlas lo mejor que se pueda bajo las circunstancias del momento de la realización; y el primer factor limitante es el tiempo.

En resumen he aprendido a realizar las tareas dentro del tiempo previsto, y a saber cortar la dedicación para cumplir plazos.

Logro de los objetivos planteados:

Creo y siento que he cumplido con los objetivos que tenía propuesto antes de realizar el proyecto.

El principal objetivo era aprender y adquirir práctica en el diseño y desarrollo de aplicaciones para Android. Creo que he conseguido dicho objetivo. Cuando empecé a realizar el proyecto, tenía serías dudas sobre si lo conseguiría o no. Ya que debía realizarlo en dos campos que no tenía ni idea, éstos son: desarrollo de aplicaciones para Android y desarrollo de videojuegos. Conseguir acabarlo y de forma digna, creo que ha sido un logro, y estoy satisfecho con el resultado.

En cuanto a los objetivos propios del juego también creo que he cumplido con todos los expuestos en el punto 1.2. El videojuego se adapta perfectamente al contexto de videojuego para dispositivos móviles, cumpliendo las principales características que deben tener.

Seguimiento de la planificación y metodología:

He seguido la planificación elaborada desde el inicio del proyecto. Aunque la fase más incierta y con riesgos de no cumplir los objetivos establecidos en ella, era la fase de implementación. Ya que no tenía ninguna experiencia previa en el desarrollo de aplicaciones para Android ni sobre videojuegos.

Por ello, en la fase de diseño del videojuego propuse unas funcionalidades mínimas a implementar y otras opcionales. La implementación de las opcionales dependería del tiempo disponible una vez realizadas las obligatorias. Con la implementación de las obligatorias consumí todo el tiempo disponible para la fase de desarrollo, así que no ha sido posible implementar ninguna funcionalidad opcional.

En cuanto a la metodología utilizada, creo que ha sido una buena utilizar la metodología de *desarrollo ágil de software.* Aplicando dicha metodología obtuve rápidamente un resultado ejecutable, con una funcionalidad mínima, y a partir de ahí fui añadiendo funcionalidades hasta conseguir el resultado.

Líneas de trabajo pendientes:

Como dije anteriormente, pensé en funcionalidades opcionales que serían interesantes implementarlas en el videojuego, éstas eran:

- *Incluir los agentes auditivos*: éstos representan las letras de forma sonora. Reproduce una letra del abecedario de forma sonora cuando el jugador se lo indique mediante algún tipo de interacción con él. La inclusión en el juego de este tipo de agentes es muy interesante, ya que ofrecería la posibilidad de practicar o aprender la fonética del abecedario de la gramática inglesa.
- *Agentes borradores:* Éstos aparecerían en el videojuego a partir de un determinado nivel de dificultad, y su velocidad de movimiento sería menor que la de los otros agentes letras. Su ingrata aportación sería "borrar" a un agente letra, es decir, si colisionan inhibiría para siempre la facultad de éste para formar palabras. Dejándolo vagante por la pantalla sin ninguna utilidad productiva, al contrario, contabiliza como un agente letra, pero no podemos utilizarlo. Su aspecto visual cambiaría, así son identificados por el jugador en todo momento. El jugador podría interactuar con los agentes de borrado para intentar inhibir o minimizar su acción de borrado. El efecto de su interacción sería cambiar la dirección de movimiento de éstos, consiguiendo así que se desplacen hacia direcciones que no perjudiquen. El modo de interactuar se decidirá e indicará si finalmente se implementa dicha funcionalidad

- Implementar infinitos niveles. Ésta última en realidad la habría podido implementar, no hubiese supuesto mucho esfuerzo desarrollarla. Pero preferí dejar el videojuego con 10 niveles.

Incluir dichas funcionalidades mejoraría el videojuego, en aspectos didácticos, de dinamismo y diversión, y durabilidad.

9. Glosario

PFC: Proyecto Final de Carrera.

SO: Sistema Operativo.

Framework: estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software.

API: del inglés Application Programming Interface, es el conjunto de funciones y procedimientos que ofrece cierta biblioteca o *framework* para ser utilizado por otro software como capa de abstracción.

IDE: Entorno de Desarrollo Integrado.

XML: del inglés eXtensible Markup Language, es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos de forma legible.

FPS: Frames por segundos.

KB: Kilobytes.

UOC: Universidad Oberta de Cataluña.

GNU: Proyecto o sistema operativo GNU/Linux. Dicha plataforma ha elaborado unas licencias de publicación para el software y la documentación, las cuales he utilizado para publicar el material de mi proyecto.

10. Bibliografía

[1] [Eclipse] Septiembre 2013
http://www.eclipse.org/downloads/

[2] [JDK Java] Septiembre 2013
http://www.oracle.com/technetwork/java/javase/downloads/index.html

[3] [SDK Android] Septiembre 2013
http://developer.android.com/sdk/index.html

[4] [Plugin ADT para Eclipse] Septiembre 2013
http://developer.android.com/intl/es/tools/sdk/eclipse-adt.html

[5] [Guía para programadores de Android] Octubre 2013
http://developer.android.com/about/dashboards/index.html

[6] [Desarrollo de juegos para Android] Octubre 2013
http://www.javacodegeeks.com/2011/07/android-game-development-game-loop.html

[7] [Desarrollo de juegos para Android] Octubre 2013
http://www.mysecretroom.com/www/programming-and-software/android-game-loops

[8] [Guía para programadores de Android]] Noviembre 2013
http://developer.android.com/training/basics/data-storage/shared-preferences.html

[9] [Guía para programadores de Android]] Enero 2014
http://developer.android.com/guide/practices/screens_support.html

[10] [Android: tamaños de pantallas y densidades] Enero 2014
http://stefan222devel.blogspot.com.es/2012/10/android-screen-densities-sizes.html

[11] [Recursos de audio] Diciembre 2014
http://www.wavsource.com/sfx/sfx3.htm

[12] [Recursos de fuentes] Diciembre 2014
http://www.1001freefonts.com/3d-fonts-3.php