Universidad de Córdoba

Escuela Politécnica Superior

Grado en Ingeniería Informática

Especialidad Computación

Trabajo de Fin de Grado

Colección de videojuegos de destreza multiplataforma

Autor:
Juan Martos Cáceres

Director:
Manuel Jesús Marín Jiménez

Córdoba - 28 de Junio de 2018

Dr. Manuel Jesús Marín Jiménez profesor del Departamento de Informática y Análisis Numérico de la Universidad de Córdoba.

CERTIFICA

Que el proyecto fin de carrera titulado: *"Colección de videojuegos de destreza multiplataforma"*, ha sido realizado bajo mi dirección por Juan Martos Cáceres, cumpliendo, a mi juicio, con los requisitos exigidos en este tipo de proyectos.

Fdo. **Dr. Manuel Jesús Marín Jiménez**

Córdoba, 28 de Junio de 2018

El alumno Juan Martos Cáceres con D.N.I. 30.999.954-W ha realizado el proyecto presentado en esta memoria junto con la dirección de Manuel Jesús Marín Jiménez.

Fdo. **Juan Martos Cáceres**

Córdoba, 28 de Junio de 2018

Índice general

Capítulo 1

Introducción

La tecnología móvil evoluciona a pasos agigantados y nos presenta en la actualidad el llamado teléfono inteligente (o smartphone en inglés). Los teléfonos inteligentes se caracterizan por el hecho de ser computadores de bolsillo con unas posibilidades cada vez más cercanas a los computadores de sobremesa.

Esta nueva generación de teléfonos inteligentes ha dado paso a las tablets o tabletas táctiles, unos dispositivos ligeros que han tratado de integrar las mejores funcionalidades de un teléfono móvil y de un ordenador.

Son varios los sistemas operativos que existen para teléfonos inteligentes y tablets, sin embargo, iOS [7] y Android [6] está creciendo cada vez más, estos sistemas operativos dan multitud de facilidades tanto a desarrolladores que quieran desarrollar aplicaciones, como a usuarios que quieren gestionar su información sin restricciones.

La curiosidad por estos nuevos dispositivos (y todos los gadgets que estos incorporan), que cada vez están más presentes en la sociedad, la curiosidad por el S.O. de Google y de Apple junto con la idea de desarrollar un videojuego es lo que me ha llevado a tratar este tema en mi trabajo de fin de grado.

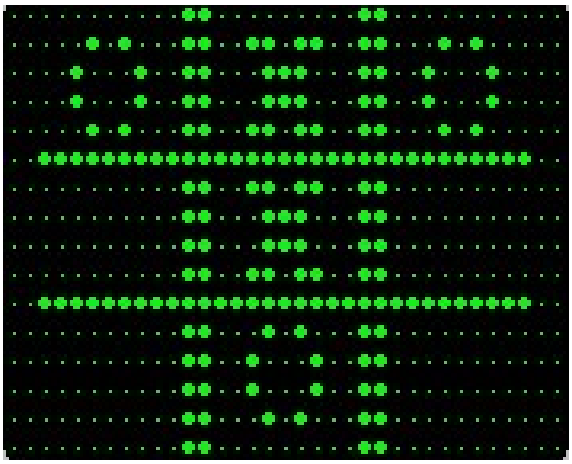
Hoy en día los dispositivos móviles se han transformado en una gran plataforma para el mundo de los videojuegos. Gracias a la evolución del hardware y de los recursos disponibles, actualmente los móviles son capaces de ofrecernos una enorme variedad de juegos. El desarrollar un videojuego para móviles que funcione en la gran mayoría de dispositivos es todo un reto para el desarrollador, por ello existen diversas librerías y herramientas que van dirigidas a resolver este problema, permitiendo desarrollar videojuegos y portarlos a las plataformas principales de móviles existentes.

El objetivo general de este trabajo de fin de grado es realizar un videojuego multiplataforma para dispositivos móviles, para web y para escritorio empleando los conocimientos adquiridos a lo largo de la carrera, diseñándolo de forma adecuada para su funcionamiento en diferentes plataformas.

1.1. Historia de los videojuegos

1.1.1. Inicio

Durante bastante tiempo ha sido complicado señalar cual fue el primer videojuego, principalmente debido a las múltiples definiciones de este que se han ido estableciendo, pero se puede considerar como primer videojuego el Nought and crosses, también llamado OXO(Fig. 1.1.1), desarrollado por Alexander S.Douglas en 1952. El juego era una versión computerizada del tres en raya que se ejecutaba sobre la EDSAC y permitía enfrentar a un jugador humano contra la máquina.



(a) OXO - El primer videojuego

En 1958 William Higginbotham creó, sirviéndose de un programa para el cálculo de trayectorias y un osciloscopio, Tennis for Two (tenis para dos): un simulador de tenis de mesa para entretenimiento de los visitantes de la exposición Brookhaven National Laboratory. Este videojuego fue el primero en permitir el juego entre dos jugadores humanos.

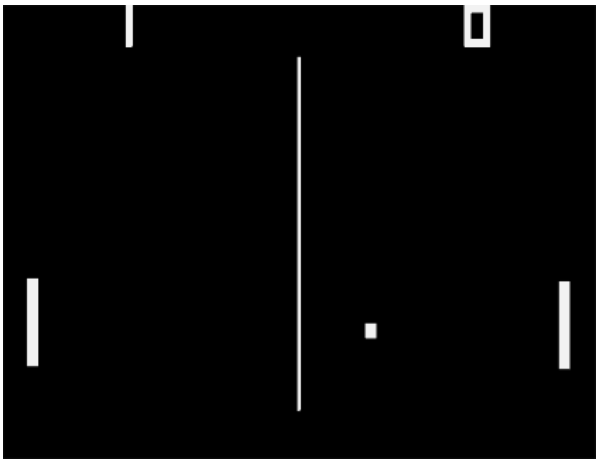
Cuatro años más tarde Steve Russell, un estudiante del Instituto de Tecnología de Massachussets, dedicó seis meses a crear un juego para computadora usando gráficos vectoriales: Spacewar. En este juego, dos jugadores controlaban la dirección y la velocidad de dos naves espaciales que luchaban entre ellas. El videojuego funcionaba sobre un PDP-1 y fue el primero en tener un cierto éxito, aunque apenas fue conocido fuera del ámbito universitario. En 1966 Ralph Baer empezó a desarrollar junto a Albert Maricon y Ted Dabney, un proyecto de videojuego llamado Fox and Hounds dando inicio al videojuego doméstico. Este proyecto evolucionaría hasta convertirse en la Magnavox Odyssey, el primer sistema doméstico de videojuegos lanzado en 1972 que se conectaba a la televisión y que permitía jugar a varios juegos pregrabados.

1.1.2. 1970-1979: La eclosión de los videojuegos

Un hito importante en el inicio de los videojuegos tuvo lugar en 1971 cuando Nolan Bushnell comenzó a comercializar Computer Space, una versión de Space War, aunque otra versión recreativa de Space War como fue Galaxy War puede que se le adelantara a

principios de los 70 en el campus de la universidad de Standford.

La ascensión de los videojuegos llegó con la máquina recreativa Pong(Fig. 1.1.2) que es considerada la versión comercial del juego Tennis for Two de Higginbotham. El sistema fue diseñado por Al Alcom para Nolan Bushnell en la recién fundada Atari.



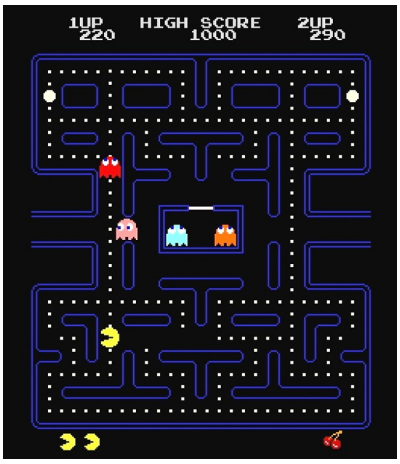
(b) Pong

El juego se presentó en 1972 y fue la piedra angular del videojuego como industria. Durante los años siguientes se implantaron numerosos avances técnicos en los videojuegos (destacando los microprocesadores y los chips de memoria). Aparecieron en los salones recreativos juegos como Space Invaders (Taito) o Asteroids (Atari).

1.1.3. 1980-1989: La década de los 8 bits

Los años 80 comenzaron con un fuerte crecimiento en el sector del videojuego alentado por la popularidad de los salones de máquinas recreativas y de las primeras videoconsolas aparecidas durante la década de los 70.

Durante estos años destacan sistemas como Oddyssey 2 (Phillips), Intellivision (Mattel), Colecovision (Coleco), Atari 5200, Commodore 64, Turbografx (NEC). Por otro lado en las máquinas recreativas triunfaron juegos como el famoso Pacman (Namco) (Fig. 1.1.3), Battle Zone (Atari), Pole Position (Namco), Tron (Midway) o Zaxxon (Sega).



(c) Pac-Man

El negocio asociado a esta nueva industria alcanzó grandes cosas en estos primeros años de los 80, pero sin embargo, en 1983 comenzó la llamada crisis del videojuego, afectando principalmente a Estados Unidos y Canadá, y que no llegaría a su fin hasta 1985.

Japón apostó por el mundo de las consolas con el éxito de la Famicom (llamada en occidente como Nintendo Entertainment System), lanzada por Nintendo en 1983 mientras en Europa se decantaba por los microordenadores como el Commodore 64 o el Spectrum. A la salida de su particular crisis los norteamericanos continuaron la senda abierta por los japoneses y adoptaron la NES como principal sistema de videojuegos. A lo largo de la década fueron apareciendo nuevos sistemas domésticos como la Master System (Sega), el Amiga (Commodore) y el 7800 (Atari) con juegos hoy en día considerados clásicos como el Tetris.

A finales de los 80 comenzaron a aparecer las consolas de 16 bits como la Mega Drive de Sega y los microordenadores fueron lentamente sustituidos por las computadoras personales basadas en arquitecturas de IBM. En 1985 apareció Super Mario Bros, que supuso un punto de inflexión en el desarrollo de los juegos electrónicos, ya que la mayoría de los juegos anteriores sólo contenían unas pocas pantallas que se repetían en un bucle y el objetivo simplemente era hacer una alta puntuación. El juego desarrollado por Nintendo supuso un estallido de creatividad. Por primera vez teníamos un objetivo y un final en un videojuego. En los años posteriores otras compañías emularon su estilo de juego. La evolución definitiva de las portátiles como plataformas de videojuego llegó en 1989 con el lanzamiento de la Game Boy (Nintendo).

1.1.4. 1990-1999: La revolución de las 3D y los primeros juegos moviles

A principios de los años 90 las videoconsolas dieron un importante salto técnico gracias a la competición de la llamada "generación de 16 bitsçompuesta por la Mega Drive, la Super Nintendo Entertainmet de Nintendo, la PC Engine de NEC, conocida como Turbografx en occidente y la CPS Changer de (Capcom).

Esta generación supuso un importante aumento en la cantidad de jugadores y la introducción de tecnologías como el CD-ROM, una importante evolución dentro de los diferentes géneros de videojuegos, principalmente gracias a las nuevas capacidades técnicas.

Mientras tanto diversas compañías habían comenzado a trabajar en videojuegos con entornos tridimensionales, principalmente en el campo de los PC, obteniendo diferentes resultados desde las "2D y media" de Doom(Fig. 1.1.4), 3D completas de 4D Boxing a las 3D sobre entornos pre-renderizados de Alone in Dark. Referente a las ya antiguas consolas de 16 bits, su mayor y último logro se produciría por el SNES mediante la tecnología 3-D de pre-renderizados de SGI, siendo su máxima expresión juegos como Donkey Kong Country y Killer Instinct. También surgió el primero juego poligonal en consola, la competencia de la SNES, Mega-Drive, lanzó el Virtual Racing, que tuvo un gran éxito ya que marcó un antes y un después en los juegos 3D en consola.

(d) Doom

Rápidamente los videojuegos en 3D fueron ocupando un importante lugar en el mercado, principalmente gracias a la llamada "generación de 32 bits."[en] las videoconsolas: Sony PlayStation y Sega Saturn (principalmente en Japón); y la "generación de 64 bits" en las videoconsolas: Nintendo 64 y Atari jaguar. En cuanto a los ordenadores, se crearon las aceleradoras 3D.

La consola de Sony apareció tras un proyecto iniciado con Nintendo, que consistía en un periférico para SNES con lector de CD. Al final Nintendo rechazó la propuesta de Sony, puesto que Sega había desarrollado algo parecido sin tener éxito, y Sony lanzó independientemente PlayStation.

Por su parte los arcades comenzaron un lento pero imparable declive según aumentaba el acceso a consolas y ordenadores más potentes.

Por su parte los videojuegos portátiles, producto de las nuevas tecnologías más poderosas, comenzaron su verdadero auge, uniéndose a la Game Boy máquinas como la Game Gear (Sega), Linx (Atari) o la Neo Geo Pocket (SNK), aunque ninguna pudo hacerle frente a la popularidad de la Game Boy, siendo esta y sus descendientes (Game Boy Pocket, Game Boy Color, Game Boy Advance, Game Boy Advance SP) las dominadoras del mercado.

Hacia finales de la década la consola más popular era la PlayStation con juegos como Final Fantasy VII (Square), Resident Evil (Capcom), Winning Eleven 4 (Konami), Gran Turismo (Polyphony Digital) y Metal Gear Solid (konami).

En PC eran muy populares los FPS (juegos de acción en primera persona) como

Quake (id Softare), Unreal (Epic Megagames) o Half-Life (Valve), y los RTS (juegos de estrategia en tiempo real) como Command & Conquer (Westwood) o Starcraft (Blizzard). Además, conexiones entre ordenadores mediante internet facilitaron el juego multijugador, convirtiéndolo en la opción predilecta de muchos jugadores, y fueron las responsables del nacimiento de los MMORPG (juegos de rol multijugador online) como Ultima Online (Origin). Finalmente en 1998 apareció en Japón la Dreamcast (Sega) y daría comienzo a la "generación de los 128 bits".

A finales de los años 1990 los teléfonos móviles todavía eran aparatos que solo servían para llamar. Algunos fabricantes como Nokia decidieron ofrecer algún tipo de entretenimiento en esos pequeños dispositivos que tenían botones y una pantalla LCD en blanco y negro, compañías como Nokia o Philips introdujeron pequeños juegos basándose en las primeras arcade y consolas de principios de los años 1980; jamás pensaron que esto revolucionaría el mundo de los videojuegos portátiles. Estos juegos fueron evolucionando y algunos ofrecían la posibilidad de desbloquear niveles pagando al operador o conectándose a Internet.

Mientras, en Japón se lanzaron los primeros móviles programables con tecnología Java I-mode-doja. Hasta ese momento los videojuegos en los móviles estaban integrados en el teléfono y programados directamente en código máquina y grabados en la memoria ROM del móvil, pero con los móviles programables, había una zona de memoria donde se podían grabar datos, y se podía utilizar un lenguaje de desarrollo como Java y un cable USB o una conexión a Internet para introducir el programa en el móvil.

La finalidad de esto era el desarrollo de pequeñas aplicaciones tipo calculadoras, notas, y no videojuegos. Pero aun así algunas empresas como la francesa Gameloft desarrollaron videojuegos en blanco y negro para esas pequeñas pantallas y resultaron un éxito.

Los móviles fueron evolucionando y con ellos la memoria, potencia y los lenguajes de programación de los mismos, Symbian OS, J2ME 2.0, doja 1.5...

1.1.5. Desde el 2000: El comienzo del nuevo siglo

En el 2000 Sony lanzó la anticipada PlayStation 2 y Sega lanzó otra consola con las mismas características técnicas de la Dreamcast, nada más que venia con un monitor de 14 pulgadas, un teclado, altavoces y los mismos mandos llamados Dreamcast Drivers 2000 Series CX-1.

Microsoft entra en la industria de las consolas creando la Xbox en 2001.

Nintendo lanzó el sucesor de la Nintendo 64, la Gamecube, y la primera Game Boy completamente nueva desde la creación de la compañía, la Game Boy Advance. Sega viendo que no podría competir, especialmente con una nueva máquina como la de Sony, anunció que ya no produciría hardware, convirtiéndose sólo en desarrolladora de software en 2002.

El ordenador personal PC es la plataforma más cara de juegos pero también la que permite mayor flexibilidad. Esta flexibilidad proviene del hecho de poder añadir al ordenador componentes que se pueden mejorar constantemente, como son tarjetas gráficas o de sonido y accesorios como volantes, pedales y mandos, etc. Además es posible actualizar los juegos con parches oficiales o con nuevos añadidos realizados por la compañía que creó el juego o por otros usuarios. En 2008 fue lanzado el sistema operativo Android, un sistema

operativo que ha revolucionado el mundo de los smartphones y de los videojuegos para móviles.

(e) Final Fantasy IX

1.1.6. Actualidad de los videojuegos - Octava generación

La industria de los videojuegos ha experimentado en los últimos años un increíble crecimiento, debido al desarrollo de la computación, la mejora en la capacidad de procesamiento, el alcance de imágenes más reales y la estrecha relación entre el cine y los videojuegos, con lo cual los consumidores reconocen los títulos antes.

En la historia de los videojuegos, la octava generación de videoconsolas es un termino que describe la generación de consolas de videojuegos que sucede a la séptima generación (PlayStation 3 de Sony, Xbox 360 de Microsoft y Wii de Nintendo), también se incluyen las unidades de juegos portátiles lanzado en el marco de tiempo similar. La octava generación comienza con el lanzamiento de Nintendo 3DS, el 25 de febrero de 2011, seguido después por el de PlayStation Vita de Sony, que fue lanzada el 17 de diciembre del mismo año. Actualmente sólo Nintendo ha anunciado la sucesora de su consola de sobremesa, la Wii U que se estrenara en 2012. Varios periodistas han calificado este sistema como el inaugurador de la octava generación.

(f) The Last Guardian

1.2. Los videojuegos en dispositivos móviles

El fácil acceso a las tecnologías para desarrollar y la implantación de este sistema operativo en la mayoría de teléfonos móviles han sido las causas para que iOS y Android hayan crecido mucho en estos años y junto a ellos un inmenso catalogo de aplicaciones/videojuegos para este sistema operativo.

Los videojuegos son tan importantes para un dispositivo móvil o tablet que gran parte del éxito de un S.O. depende de las facilidades que éste dé para desarrollar videojuegos y, por tanto, del catálogo que éste ofrezca a sus usuarios.

Los videojuegos móviles han sabido abrirse paso en el mercado gracias a que son sencillos de jugar y muy intuitivos. Si echamos un vistazo a los videojuegos para consolas de sobremesa o portátiles, nos ofrecen un amplio catálogo de juegos y una gran variedad, sin embargo, nos es algo tan común llevarlas siempre encima, al contrario que los móviles, que se han convertido hoy en día en una herramienta muy necesaria y que siempre tenemos a mano, por lo que en cualquier momento y en cualquier lugar podemos echar unas partidas.

(g) Pokemon GO

1.3. Motivación

Para un programador siempre es llamativa la idea de ser desarrollador de videojuegos, poder crear un mecanismo de juego, un escenario, un personaje, etc., y que los usuarios puedan llegar a divertirse y disfrutar con él, de aquí la idea de este proyecto.

Dentro del mundo de los videojuegos tenemos que mencionar los minijuegos, juegos en los que la duración de una partida es bastante corta, la dificultad va en función de la habilidad del jugador y son bastante adictivos.

La mezcla de los minijuegos con el concepto táctil puede ser muy interesante debido a que se pueden desarrollar juegos en los que la interacción táctil sea continua y para eso los dispositivos móviles y tablets son geniales para este tipo de juegos.

Capítulo 2

Identificación del problema

2.1. Identificación del problema real

Lo que se pretende con este proyecto es la realización de un videojuego multiplataforma, con el fin de aprender toda la gestión del diseño, desarrollo y programación de un videojuego desde cero.

Dicho videojuego, permitirá la realización de tests de destreza mediante la realización de varias pruebas en las categorías de lógica, memoria, matemáticas, visual, secuencias y pesos, esta prueba devolverá una puntuación y un perfil basado en dicha puntuación.

Puesto que el videojuego se basa en la realización de un test con distintas pruebas se proporcionara también un modo de entrenamiento donde el usuario podrá practicar las pruebas de forma individual con el fin de mejorar en vista a la realización del test.

El videojuego dará la posibilidad de comparar la puntuación con el fin de atraer a los máximos usuarios posibles.

Para intentar que el videojuego llegue al mayor número de usuarios posibles, se incorpora como idioma adicional el inglés además del español. De esta forma no se restringe el uso del videojuego a usuarios hispanohablantes.

2.2. Identificación del problema técnico

Una vez descrito el problema real pasamos a analizar el problema técnico. Utilizaremos la metodología PDS[11] (Product Design Specification), la cual establece las pautas para asegurar que se cumplan las necesidades del cliente.

2.2.1. Funcionamiento

El funcionamiento general del videojuego se puede resumir en las siguientes funcionalidades:

- **Explicación del test.** El videojuego proporcionara al usuario una animación donde se mostrara el proceso para realizar un test.
- **Realización del test.** El videojuego deber permitir al usuario la realización de un test.
- **Guardar perfil y puntuación.** El videojuego deber permitir al usuario almacenar la puntuación y el perfil resultante de la realización del test.
- **Practica individual de las pruebas.** El videojuego debe permitir la realización de pruebas individuales en las distintas categorías de lógica, memoria, matemáticas, visual, secuencias y pesos.
- **Guardar puntuación de las pruebas individuales.** El videojuego deber permitir al usuario almacenar la puntuación de la prueba individual.
- **Comparar puntuaciones.** El videojuego debera permitir comparar las puntuaciones, tanto de la realización del test como de las pruebas individuales.
- **Vista de puntuación.** El videojuego deber permitir al usuario la consulta de las puntuaciones conseguidas.

2.2.2. Entorno

Puesto que el videojuego está destinado a ser multiplataforma, utilizaremos una librería gráfica multiplataforma, la librería es libGDX[4], puesto que esta librería esta basada en Java [12], el **lenguaje de programación** utilizado es Java [12].

El **entorno de desarrollo** usado es Android Studio [9] junto con el SDK de Android y el SDK de libGDX, este entorno nos servira para compilar las aplicaciones para Android, para escritorio y para web.

Un segundo **entorno de desarrollo** usado es Xcode [13] junto con el SDK para iOS, este entorno nos proporcionara la compilación para iOS.

El **entorno hardware** usado se puede consultar en la parte de recursos hardware, sección 7.

Los **usuarios finales** a los que va destinado este videojuegos son aquellos que dispongan de un dispositivo cuya versión del sistema sea igual o superior a Android 4.0 *"ICE CREAM SANDWICH"*, igual o superior a iOS 9.0, que contenga un navegador web o que tenga Java integrado.

2.2.3. Esperanza de vida

La vida esperada del proyecto es difícil de determinar, pero se prevee que sea larga puesto que al ser multiplataforma y funcionar en entornos con un gran uso mundial y que se actualizan constantemente, por lo que si en futuras versiones no se alteran los recursos usados actualmente, el videojuego seguirá siendo funcional.

2.2.4. Ciclo de mantenimiento

El ciclo de mantenimiento de la aplicación, dependerá de las futuras versiones del SDK de libGDX puesto que el videojuego podrá recibir actualizaciones para adecuarse a las nuevas versiones de los sistemas operativos.

2.2.5. Competencia

Actualmente hay algunos videojuegos de destreza con ciertos parecidos que podrían competir con este videojuego. Sin embargo no hay ninguno que reúna todos las distintas pruebas en uno solo y que se pueda compartir en las redes sociales.

Por lo tanto, con lo anteriormente expuesto, podemos concluir que la competencia en cuanto a funcionalidad específica de la aplicación es bastante baja.

2.2.6. Aspecto externo

La entrega definitiva de la aplicación se realizará en un CD-ROM que incluirá la aplicación lista para su instalación, así como el código fuente, los correspondientes manuales (manual técnico, de código y de usuario) y la distribución binaria del videojuego vía web/Google Play/escritorio.

2.2.7. Estandarización

Se generará un código fuente suficientemente claro y documentado para que puedan analizarse las técnicas utilizadas y realizar cambios si se estima oportuno. Para ello se utilizarán los estilos de codificación más utilizados a nivel mundial en Java. Además, se seguirán las recomendaciones de desarrollo en Java.

Se ha utilizado la librería libGDX que no es estándar pero su uso está creciendo.

2.2.8. Calidad y fiabilidad

La calidad y fiabilidad están garantizadas siempre que el dispositivo cumpla con los requisitos mínimos exigidos y el usuario haga un uso correcto del videojuego. Si estos requisitos no se cumplen, el funcionamiento del videojuego puede ser incorrecto. Para garantizar la calidad y fiabilidad se realizarán pruebas exhaustivas para detectar posibles errores y subsanarlos.

2.2.9. Fases de desarrollo

Para especificar las diferentes fases de desarrollo se han analizado los distintos modelos existentes en la Ingeniería del Software con la intención de adaptar el desarrollo del proyecto a uno de ellos [17]. De esta manera, hemos procedido a adoptar un modelo iterativo incremental, en el que a medida que se vaya avanzando se producirán versiones

intermedias con la finalidad de detectar sus errores y depurarlos al mismo tiempo que se avanza en la creación de otros módulos.

Fase 1 – Investigación

En esta etapa se recopilará toda la información requerida para cumplir los objetivos del proyecto, aplicaciones, desarrollos previos, etc. Esta fase se puede dividir en los siguientes apartados:

- Analizar el entorno de desarrollo de libGDX, así como familiarizarme con la SDK del mismo y, en concreto, con las posibles clases relacionadas con el proyecto. Para ello se realizarán pequeños prototipos a modo de aprendizaje.
- Analizar los videojuegos realizados con la librería libGDX para multiplataforma con el fin de extraer funcionalidades para el proyecto.
- Consultar los distintos manuales y libros más conocidos para aprender los conceptos básicos del SDK de desarrollo facilitándome así la tarea de aprendizaje.
- Analizar el porcentaje de dispositivos móviles con cada versión de Android y con cada versión de iOS para así decantarme por la versión mínima necesaria para estas plataformas, teniendo en cuenta los requisitos a implementar y las posibles restricciones que generen.

Fase 2 – Especificación de Requisitos

En esta fase se definirá como se implementarán y obtendrán los distintos objetivos funcionales definidos en el apartado 3.1. De manera que tendremos los siguientes puntos:

- Determinar las principales entidades que intervienen en el dominio de la aplicación y las relaciones que hay entre ellos y entidades externas mediante casos de uso.
- Creación de un diagrama de clases a partir de los casos de uso que hayamos generado para utilizarlos en fases posteriores en el diseño e implementación de los módulos.

Fase 3 – Diseño de Módulos

En esta fase se utilizará todo el conocimiento acumulado en la Fase 1 (apartado 4) y los casos de uso y diagramas de clases obtenidos durante la Fase 2 (apartado 4) para definir los siguientes puntos:

- Diseñar la interfaz de usuario, especificando el aspecto externo, así como de las distintas pantallas que tendrá el videojuego.
- Diseñar el modelo de datos que se empleará para representar las diferentes estructuras que intervendrán en el videojuego.
- Diseñar las distintas pruebas de destreza que contendrá el videojuego.

Fase 4 – Implementación de los módulos

En esta fase implementaremos las funciones y métodos requeridos para cumplir los puntos definidos en la Fase 3 (apartado 4). Se añadirán una serie de comentarios en el código generado a modo de documentación estándar (cabeceras de funciones, comentarios puntales, etc.). En este apartado también implementaremos todo lo relacionado con el aspecto visual del videojuego.

Fase 5 – Creación de versiones intermedias

Esta fase consistirá en ir empaquetando cada cierto tiempo todo el código y recursos necesarios para la distintas plataformas, un tipo de archivo ejecutable válido para el sistema Android, un tipo de archivo ejecutable válido para escritorio, una carpeta con varios recursos para web y un tipo de archivo ejecutable válido para el sistema iOS. Así, por cada nueva funcionalidad añadida a la aplicación, se generará una nueva versión de prueba de la misma.

El ejecutable generado podrá ser instalado en cualquier dispositivo cuya versión del sistema sea igual o superior a Android 4.0 *"ICE CREAM SANDWICH"*, igual o superior a iOS 9.0, que contenga un navegador web o que tenga Java integrado y que cumpla con los requisitos de versión de la aplicación.

Fase 6 – Pruebas de ejecución

En esta fase se pondrá a prueba todo lo que hemos implementado hasta el momento, instalando el videojuego en varios dispositivos distintos para garantizar que la aplicación sea compatible con todo tipo de hardware. Durante estas pruebas, se comprobarán todas y cada una de las funcionalidades implementadas y se probará que su funcionalidad sea la deseada y que, además, no haya ningún tipo de problema con la interfaz de usuario en ningún tipo de pantalla.

Tras esta fase pasaremos de nuevo a la Fase 3 (apartado 4) para diseñar nuevos aspectos de la aplicación. Podrá darse el caso en el que no tengamos nada más que estudiar en términos de diseño y pasemos directamente a la Fase 4 (apartado 4) para continuar implementando nuevas funciones y escenarios.

Una vez terminados todos los procesos, se procederá a una prueba global de todo el videojuego. Para ello, se procederá a probar todas las distintas pruebas del videojuego asi como la realización del test, comprobando que se realice bien la carga de las pruebas y de los distintos estados del videojuego.

De existir algún tipo de error, se procedería a erradicar dicho problema y generar una nueva versión final. Este proceso lo llamaremos como beta-testing. Para aligerar este proceso, se instalará la aplicación en varios dispositivos diferentes que serán utilizados por otras personas. De este modo, cualquier error que haya en la aplicación, saldrá a la luz de una forma mucho más rápida que solo utilizando uno o dos dispositivos únicamente.

Fase 7 – Documentación

Albergará todo el período de realización del proyecto. El objetivo de esta fase es plasmar el proceso seguido en la realización del proyecto, incluyéndose, además, un manual de uso de la aplicación desarrollada y el código fuente de la misma convenientemente comentado. Esta fase se realizará de forma simultánea a las otras fases del proyecto. Cuando se finalice esta fase, se obtendrá el videojuego final, así como el manual técnico donde se recoge el diseño del proyecto, el manual de usuario que especifica como usar el videojuego y un manual de código donde se comentará el código fuente del videojuego.

2.2.10. Pruebas

Durante el desarrollo del videojuego se irán realizando diversas pruebas para comprobar el funcionamiento del mismo. Estas pruebas las clasificaremos en pruebas de unidad (caja blanca) y pruebas funcionales (caja negra). Para llevarlas a cabo, un grupo de personas probara el videojuego y se recogerán los posibles errores para poder subsanarlos.

2.2.11. Seguridad

Puesto que el videojuego no requiere de ningún sistema de autenticación o similar para el uso de la misma, no se puede analizar la seguridad de la misma. En cuanto a la protección de copia indebida de la aplicación, tampoco se puede realizar un análisis puesto que la aplicación es totalmente gratuita y no tiene sentido introducir sistemas anti-copia.

Capítulo 3

Objetivos

3.1. Objetivos principales

Los principales objetivos a cumplir por la aplicación son los siguientes:

1. Desarrollar una colección de videojuegos de destreza en el que el usuario pueda divertirse mediante la realización de varias pruebas de lógica que sea compatible con cualquier dispositivo cuya versión del sistema sea igual o superior a Android 4.0 *"ICE CREAM SANDWICH"*, igual o superior a iOS 9.0, que contenga un navegador web o que tenga Java integrado.
2. Analizar y estudiar el entorno de desarrollo y las herramientas internas o externas que el sistema operativo Android y el sistema operativo iOS tiene para el desarrollo de videojuegos.
3. Estudiar el funcionamiento interno de Android y de iOS así como la estructura de un videojuego.
4. Estudiar el funcionamiento de la librería libGDX para el desarrollo de videojuegos en varias plataformas.
5. Desarrollar un videojuego en dos dimensiones utilizando las herramientas seleccionadas en el análisis.

El objetivo principal de la aplicación es el de permitir al usuario la posibilidad de realizar un test que contendrá distintas pruebas de destreza entre las categorías de lógica, memoria, visual y matemáticas. Este test una vez realizado le dará una puntuación y un perfil.

Además, para poder practicar a la hora de realizar este test, la aplicación incorpora un modo libre para que el usuario pueda practicar las pruebas de forma individual.

3.2. Objetivos personales

Al elegir este proyecto tengo como objetivo alcanzar las siguientes metas:

1. Hacer frente a un proyecto real para asentar los conocimientos adquiridos durante los años de carrera.
2. Aprender a desarrollar un videojuego desde cero con el fin de poder hacer frente a futuros proyectos.
3. Realizar un módulo para la comunicación en redes sociales.
4. Familiarizarme con el entorno de desarrollo multiplataforma libGDX, un sistema cada día más usado y con un amplio futuro.
5. Obtener la capacidad de acotar un coste en tiempo y dinero para la realización de un videojuego multiplataforma a través de la experiencia adquirida con la realización del proyecto.

Capítulo 4

Fases de desarrollo

Para especificar las diferentes fases de desarrollo se han analizado los distintos modelos existentes en la Ingeniería del Software con la intención de adaptar el desarrollo del proyecto a uno de ellos [17]. De esta manera, hemos procedido a adoptar un modelo iterativo incremental, en el que a medida que se vaya avanzando se producirán versiones intermedias con la finalidad de detectar sus errores y depurarlos al mismo tiempo que se avanza en la creación de otros módulos.

Fase 1 – Investigación

En esta etapa se recopilará toda la información requerida para cumplir los objetivos del proyecto, aplicaciones, desarrollos previos, etc. Esta fase se puede dividir en los siguientes apartados:

- Analizar el entorno de desarrollo de Android, así como familiarizarme con la SDK del mismo y, en concreto, con las posibles clases relacionadas con el proyecto. Para ello se realizarán pequeños prototipos a modo de aprendizaje.
- Analizar el entorno de desarrollo de iOS, así como familiarizarme con la SDK del mismo y, en concreto, con las posibles clases relacionadas con el proyecto. Para ello se realizarán pequeños prototipos a modo de aprendizaje.
- Analizar el entorno de desarrollo de libGDX, así como familiarizarme con la SDK del mismo y, en concreto, con las posibles clases relacionadas con el proyecto. Para ello se realizarán pequeños prototipos a modo de aprendizaje.
- Analizar los videojuegos realizados con la librería libGDX con el fin de extraer funcionalidades para el proyecto.
- Consultar los distintos manuales y libros más conocidos para aprender los conceptos básicos del SDK de desarrollo facilitándome así la tarea de aprendizaje.
- Analizar el porcentaje de dispositivos móviles con cada versión de Android para así decantarme por la versión mínima necesaria, teniendo en cuenta los requisitos a implementar y las posibles restricciones que generen.

Fase 2 – Especificación de Requisitos

En esta fase se definirá como se implementarán y obtendrán los distintos objetivos funcionales definidos en el apartado 3.1. De manera que tendremos los siguientes puntos:

- Determinar las principales entidades que intervienen en el dominio de la aplicación y las relaciones que hay entre ellos y entidades externas mediante casos de uso.
- Creación de un diagrama de clases a partir de los casos de uso que hayamos generado para utilizarlos en fases posteriores en el diseño e implementación de los módulos.

Fase 3 – Diseño de Módulos

En esta fase se utilizará todo el conocimiento acumulado en la Fase 1 (apartado 4) y los casos de uso y diagramas de clases obtenidos durante la Fase 2 (apartado 4) para definir los siguientes puntos:

- Diseñar la interfaz de usuario, especificando el aspecto externo, así como de las distintas pantallas que tendrá el videojuego.
- Diseñar el modelo de datos que se empleará para representar las diferentes estructuras que intervendrán en el videojuego.
- Diseñar las distintas pruebas de destreza que contendrá el videojuego.

Fase 4 – Implementación de los módulos

En esta fase implementaremos las funciones y métodos requeridos para cumplir los puntos definidos en la Fase 3 (apartado 4). Se añadirán una serie de comentarios en el código generado a modo de documentación estándar (cabeceras de funciones, comentarios puntales, etc.). En este apartado también implementaremos todo lo relacionado con el aspecto visual del videojuego.

Fase 5 – Creación de versiones intermedias

Esta fase consistirá en ir empaquetando cada cierto tiempo todo el código y recursos necesarios para la aplicación en un tipo de archivo ejecutable válido para el sistema Android, para el sistema iOS, para navegador web y para aplicación de escritorio. Así, por cada nueva funcionalidad añadida a la aplicación, se generará una nueva versión de prueba de la misma.

El ejecutable generado podrá ser instalado en cualquier dispositivo móvil, navegador web o en el escritorio siempre y cuando cumpla con los requisitos de versión. Además, también se tendrá la posibilidad de instalar la aplicación en un dispositivo virtual creado en el ordenador gracias a las herramientas de desarrollo de Android y las herramientas de desarrollo para iOS.

Fase 6 – Pruebas de ejecución

En esta fase se pondrá a prueba todo lo que hemos implementado hasta el momento, instalando el videojuego en varios dispositivos móviles distintos para garantizar que la aplicación sea compatible con todo tipo de hardware. Durante estas pruebas, se comprobarán todas y cada una de las funcionalidades implementadas y se probará que su funcionalidad sea la deseada y que, además, no haya ningún tipo de problema con la interfaz de usuario en ningún tipo de pantalla.

Tras esta fase pasaremos de nuevo a la Fase 3 (apartado 4) para diseñar nuevos aspectos de la aplicación. Podrá darse el caso en el que no tengamos nada más que estudiar en términos de diseño y pasemos directamente a la Fase 4 (apartado 4) para continuar implementando nuevas funciones y escenarios.

Una vez terminados todos los procesos, se procederá a una prueba global de todo el videojuego. Para ello, se procederá a probar todas las distintas pruebas del videojuego así como la realización del test, comprobando que se realice bien la carga de las pruebas y de los distintos estados del videojuego.

De existir algún tipo de error, se procedería a erradicar dicho problema y generar una nueva versión final. Este proceso lo llamaremos como beta-testing. Para aligerar este proceso, se instalará la aplicación en varios dispositivos diferentes que serán utilizados por otras personas. De este modo, cualquier error que haya en la aplicación, saldrá a la luz de una forma mucho más rápida que solo utilizando uno o dos dispositivos únicamente.

Fase 7 – Documentación

Albergará todo el período de realización del proyecto. El objetivo de esta fase es plasmar el proceso seguido en la realización del proyecto, incluyéndose, además, un manual de uso de la aplicación desarrollada y el código fuente de la misma convenientemente comentado. Esta fase se realizará de forma simultánea a las otras fases del proyecto. Cuando se finalice esta fase, se obtendrá el videojuego final, así como el manual técnico donde se recoge el diseño del proyecto, el manual de usuario que especifica como usar el videojuego y un manual de código donde se comentará el código fuente del videojuego.

Capítulo 5

Antecedentes

Antes de adentrarnos en las funcionalidades debemos de tener claro el concepto de videojuego, este concepto es bastante difícil de definir debido a que es algo bastante abstracto, los videojuegos en sí son el software que funcionan sobre una plataforma electrónica, pero también son definidos de esta manera al componente tecnológico, en otras palabras, el videojuego designa tanto al software como al hardware.

En nuestro caso la definición puede ser bastante sencilla, un videojuego será el software que crearemos para dar entretenimiento y poder usarlo en un hardware móvil o tablet.

En los siguientes apartados se citarán las funcionalidades que deberá cumplir el videojuego a desarrollar. Para la elaboración del videojuego hemos examinado el mercado en busca de videojuegos similares con el fin de poder realizar algo novedoso, hemos recopilado varias funcionalidades de algunos juegos muy famosos.

El primer videojuego examinado es "Who Has The Biggest Brain?" [1], videojuego que causo mucha adicción en la red social Facebook y que está desarrollado por la compañía Play Fish, de este videojuego obtenemos la idea de la metodología del juego, éste consistirá en una partida en el que podremos ir realizando diferentes pruebas de lógica y al final nos devuelva una puntuación en base a las pruebas realizadas.

El segundo videojuego examinado es "Big Brain Academy" [3], este videojuego está desarrollado por la mejor compañía de videojuegos hasta el momento que es Nintendo. Fue uno de los primeros juegos que salió junto con la primera Nintendo DS, de "Big Brain Academy" obtenemos la idea de un segundo modo de juego, éste consistirá en un modo libre en el que podremos realizar las pruebas de manera individual con el que mejoraremos nuestra destreza para afrontar las pruebas del modo normal.

Finalmente el último videojuego examinado es "Brain Training" [2], juego desarrollado también por Nintendo. Este juego tiene la culpa de que Nintendo DS se hiciese tan famosa, es un juego muy adictivo, uno de sus atractivos era que podíamos hacer un test en el que realizabas una serie de pruebas y te devolvía una edad mental, esta edad mental se almacenaba y no podías intentar mejorarla hasta el día siguiente, mientras esperabas al día siguiente podías practicar distintas pruebas para ir mejorando. De "Brain Training" obtenemos la idea de guardar reportes diarios con la posibilidad de comparar nuestras puntuaciones en el tiempo.

Además de videojuegos sobre lógica, se ha examinado uno de los videojuegos de ejemplo que aporta el autor de la librería libGDX [5] denominado "Pax-Britannica" con el fin de ver cómo organizar de manera correcta nuestro proyecto.

En definitiva, la aplicación contendrá todas las funcionalidades anteriormente citadas, también incorporaremos una posibilidad de poder compartir nuestras puntuaciones en las redes sociales para hacer más interesante el videojuego.

Este trabajo final de grado ha sido elegido por un motivo bastante claro, el aprendizaje, desarrollo y programación de un videojuego es una idea muy atractiva que me puede abrir posibilidades en un futuro para trabajar en este sector.

Capítulo 6

Restricciones

A continuación se expondrán todas aquellas restricciones existentes en el ámbito del diseño y que condicionan la elección de una u otra alternativa.

Los factores dato corresponden a aquellas restricciones que fueron extraídas directamente de la solicitud del cliente siendo estas inamovibles, mientras que los factores estratégicos son aquellas decisiones que fueron tomadas durante el desarrollo del proyecto.

6.1. Factores dato

Los factores dato que se tienen son:

- El sistema operativo sobre el que se debe ejecutar la aplicación, debe ser Android 4.0 *"ICE CREAM SANDWICH"*, igual o superior a iOS 9.0, que contenga un navegador web o que tenga Java integrado.
- Los lenguajes utilizados serán Java, puesto que es el lenguaje en que el esta basada la librería gráfica libGDX. Los lenguajes de programación se consideran factores dato debido a que la librería gráfica libGDX lo limita a dichos lenguajes.
- El videojuego debe proporcionar una animación explicativa para la realización del test.
- El videojuego debe proporcionar la realización del test de destreza.
- El videojuego debe proporcionar la realización de pruebas individuales.
- El videojuego debe almacenar las puntuaciones y perfiles logradas en el test y las puntuaciones logradas en las pruebas.
- El videojuego debe mostrar las puntuaciones conseguidas.
- La aplicación debe ser compatible con el mayor número de dispositivos posibles.

6.2. Factores estratégicos

Los factores estratégicos elegidos para el desarrollo del proyecto son los siguientes:

- El entorno de programación utilizado es AndroidStudio puesto que es el que se recomienda para trabajar con libGDX. En AndroidStudio se ha integrado el SDK de libGDX mediante varios plugin para la compilación de las distintas plataformas que soporta libGDX. Una alternativa a AndroidStudio sería Eclipse pero se ha elegido AndroidStudio debido a que es el recomendado por libGDX y, además, ya se tenía experiencia en el uso de este entorno.
- Se usa la librería gráfica libGDX para la programación del videojuego. Otra librería que podría usarse seria AndEngine pero se ha elegido libGDX debido a que esta mucho mas activa y actualizada.
- Se restringe la versión mínima del sistema operativo Android a 4.0 *"ICE CREAM SANDWICH"* puesto que de esta forma la aplicación estaría disponible para el 99 % de los dispositivos Android y se restringe la versión mínima del sistema operativo iOS a 9.0 puesto que de esta forma la aplicación estaría disponible para el 98 % de los dispositivos iOS.
- Se utilizará una base de datos local para almacenar las puntuaciones y perfiles obtenidos por el usuario al realizar el test y las pruebas individuales. Una alternativa a la base de datos seria la utilización de XML pero se ha elegido el almacenamiento mediante base de datos a la hora de la facilidad de realizar ordenamientos y comparaciones.

Capítulo 7

Recursos

En este capítulo se exponen los recursos que se van a emplear durante el desarrollo del proyecto. Estos recursos se dividen en recursos humanos y recursos materiales (en este caso, recursos de hardware y de software).

7.1. Recursos humanos

Autor: Juan Martos Cáceres.

Alumno de 4º de Grado en Ingeniería Informática.

Director: Dr. Manuel Jesús Marín Jiménez.

Profesor del Departamento de Informática y Análisis Numérico.

7.2. Recursos software

- Sistema operativo macOS Sierra 64 bits
- Entorno de Desarrollo Android Studio 2.2.3 [9] + Plugin ADT 25.0 (Android Development Tools) [6]
- SDK de Android en su versión 4.0 y superiores [8]
- SDK de iOS en su versión 9.0 y superiores [7]
- Librería libGDX en su ultima versión.
- Editor de imágenes GIMP 2.8 [10]
- Generador de documentos LaTeX Kile [**?**]
- Editor de LaTeX online ShareLatex [16]

7.3. Recursos hardware

Para la realización de este proyecto se dispondrá de los siguientes ordenadores:

- Ordenador portátil
  - Procesador: 2,4 GHz Intel Core i5
  - Caché:
    - L1: 32 Kb + 32 Kb por núcleo
    - L2: 256 Kb por núcleo
    - L3: 3 Mb
  - Memoria RAM: 8 Gb (1600 MHz DDR3 SDRAM)
  - Intel Iris 1536 MB
  - Disco Duro: 256 SSD

Puesto que nuestro videojuego es multiplataforma también deberemos contar con un entorno móvil para probarlo. Dispondremos de los siguientes teléfonos móviles:

- Nexus 5
  - Procesador: Qualcomm MSM8974 Snapdragon 800
  - Memoria RAM: 2 GB
  - Gráficos: Adreno 330
  - Pantalla: 5.0 pulgadas con una resolución de 21920x1080 píxeles (Corning Gorilla Glass 3)
  - Almacenamiento: 16 GB
  - Versión de Android: Android 6.0 *"Marshmallow"*
- iPhone 5
  - Procesador: 1.3Ghz Dual-Core Swift ARM v7
  - Memoria RAM: 1GB LPDDR2
  - Gráficos: PowerVR SGX543 MP3
  - Pantalla: 4.0 pulgadas con una resolución de 640 x 1136 píxeles Gorilla Glass
  - Almacenamiento: 16 GB
  - Versión de iOS: iOS 9.0

Además de estos dos dispositivos móviles se utilizaran todos los móviles posibles de otras personas para acelerar el proceso de búsqueda y corrección de errores así como probar la aplicación en distintos tamaños y resoluciones de pantalla para poder adaptarla a la mayor variedad de hardware posible.

Puesto que también queremos que la aplicación este optimizada para tablets realizaremos pruebas en la siguiente tablet:

- Nexus 7
  - Procesador: NVIDIA Tegra 3 quad-core 1.3GHz
  - Memoria RAM: 1 GB
  - Gráficos: GPU ULP GeForce
  - Pantalla: 7.0 pulgadas con una resolución de 800 x 1280 píxeles LCD IPS touchscreen capacitivo
  - Almacenamiento: 16 Gb Externos
  - Versión de Android: Android 6.0 *"Marshmallow"*

Capítulo 8

Especificación de requisitos

8.1. Descripción general

En este capítulo se detallan la especificación de requisitos así como toda la documentación correspondiente del análisis de la aplicación. A partir de este análisis estableceremos un diseño que se ajuste a los requisitos definidos y realizaremos un estudio de reutilización de componentes.

Los requisitos principales que debe cumplir el videojuego que se va a desarrollar son los siguientes:

- Robustez, control de fallos y recuperación de éstos cuando sea posible. La aplicación debe administrar los fallos y las situaciones inesperadas, avisar al usuario y proporcionar mecanismos para su control.
- Eficiencia, optimización y versatilidad del código. Se deben realizar todas las funciones de la aplicación con la mayor optimización posible.
- Gestión de la memoria RAM. EL videojuego debe gestionar esta memoria de la mejor manera para evitar cuelgues o lentitud en el dispositivo.
- Diseño de una interfaz clara y sencilla de utilizar que permita usar la aplicación a todo tipo de usuarios sin necesidad de conocimientos informáticos.
- Generación de una documentación completa del funcionamiento de la aplicación.

8.2. Descripción de la información

Entidades

Una entidad es un objeto concreto o abstracto que presenta interés para el sistema y sobre el que se recoge información la cual va a ser representada en un sistema de base de datos.

La base de datos cuenta con dos entidades:

- Entidad Play
  Se encarga de almacenar las puntuaciones correspondientes a las partidas del modo jugar.

Atributo	Descripción	Dominio	Nulo	Tipo
id	Identificador de partida realizada	Número entero	No	Clave primaria
date	Fecha de realización de la partida	Fecha	No	Atributo simple
score	Puntuación obtenida en la partida	Número entero	No	Atributo simple

Tabla 8.1: Entidad Play

- Entidad Games
  Se encarga de almacenar las puntuaciones correspondientes a las partidas del modo juegos.

Atributo	Descripción	Dominio	Nulo	Tipo
id	Identificador de partida realizada	Número entero	No	Clave externa
date	Fecha de realización de la partida	Fecha	No	Atributo simple
score	Puntuación obtenida en la partida	Número entero	No	Atributo simple
game	Juego seleccionado en la partida	Cadena	No	Atributo simple
difficulty	Dificultad seleccionada en la partida	Número entero	No	Atributo simple

Tabla 8.2: Entidad Ejercicio

Esquema Entidad

A continuación se muestra el esquema Entidad obtenido a partir del estudio de los requisitos.

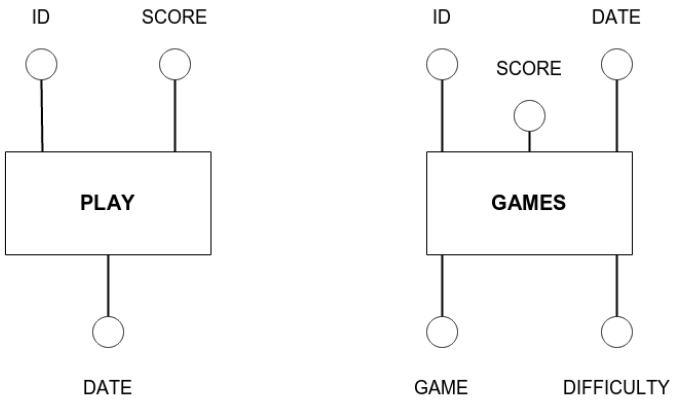


Figura 8.1: Esquema Entidad

8.3. Descripción funcional

Para la descripción funcional del sistema se utilizarán los diagramas de casos de uso[20]. Los casos de uso pretenden ser herramientas simples para describir el comportamiento del software o de los sistemas. Un caso de uso contiene una descripción textual especificando cómo podrán trabajar los distintos usuarios con el software o sistema. Los casos de uso no describen ninguna funcionalidad interna, ni tampoco exponen cómo se implementará.

Como se ha dicho, estos diagramas se utilizan para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios o con otros sistemas. Es decir, en un diagrama de caso de uso se muestra la relación entre usuarios del sistema y las tareas o usos que pueden desempeñar en dicho sistema. También ilustran cómo reacciona un sistema a eventos que se producen en su ámbito.

Los diagramas de casos de uso pueden incorporar gran cantidad de elementos, símbolos y nomenclaturas para definir incluso los sistemas más complejos. Para el diseño de la aplicación se utilizarán diagramas simples que ayuden a la comprensión de cada una de las tareas. Los elementos que se van a utilizar en el modelado de caso de uso se presentan a continuación.

- **Actor.** Especifica un rol que cierta entidad adopta cuando interactúa con un sistema directamente. Puede representar un rol de una persona o el de otro sistema o hardware. Una misma persona física, por ejemplo, puede desempeñar los roles de distintos actores simultáneamente. Los actores siempre son externos al sistema.

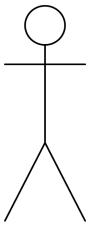


Figura 8.2: Actor

- **Sujeto o límite del sistema.** Es una parte muy importante del modelo. Indica qué forma parte del sistema y que es externo a éste. Está definido por quién utiliza el sistema (los actores) y qué beneficios les ofrece el sistema (casos de uso).

Figura 8.3: Límite del sistema

- **Caso de uso.** Es algo que el actor quiere que el sistema haga. Siempre son iniciados por un actor y se escriben desde el punto de vista de éste. Se dibujan dentro de los límites del sistema, pues son los elementos clave de éste.

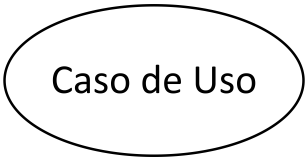


Figura 8.4: Caso de uso

- **Comunicación.** Es la relación presente entre un actor y un caso de uso. Se indica simplemente con una línea recta que conecta el actor y el caso de uso.
- **Inclusión.** Es una relación que se da entre dos casos de uso. En la Figura 8.5 se puede observar como el caso A incluye el comportamiento del caso B. Se representa con una flecha discontinua que apunta al comportamiento incluido en el caso base y se incluye la palabra clave *«include»* al lado.
- **Exclusión.** Es una relación que se da entre dos casos de uso. En la Figura 8.5 se puede observar como el caso C extiende su comportamiento al caso D. Esto es, el caso de uso C incluye en algunas ocasiones el comportamiento del caso D. Se representa con una flecha discontinua que apunta al caso al que se le añade el nuevo comportamiento (caso base) y se incluye la palabra clave *«extends»* al lado.

A continuación se muestra un ejemplo que incluye todos los elementos comentados anteriormente.

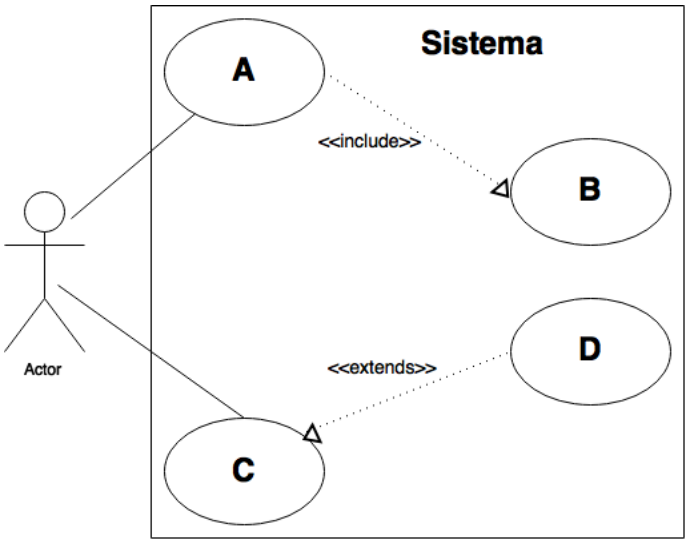


Figura 8.5: Ejemplo de caso de uso con todas las notaciones

Además de los diagramas explicativos, para cada caso de uso se anexa una tabla descriptiva, llamada especificación del caso de uso. UML[18] no propone ningún modelo estándar para estas tablas y la cantidad de información a exponer acerca del caso de

uso puede ser muy variada. En función de la complejidad del caso de uso, estas tablas contendrán más o menos información. Para el proyecto se ha decidido utilizar la plantilla de la Tabla 8.3.

Nombre	Nombre del caso de uso
Descripción	Breve comentario que resume el objetivo del caso de uso
Actores	Actores presentes en el caso de uso
Precondiciones	Restricciones que definen el estado del sistema antes del comienzo del caso de uso
Flujo principal	Pasos de un caso de uso que se listan como un flujo de eventos
Postcondiciones	Estado del sistema tras la ejecución del caso de uso

Tabla 8.3: Plantilla para la especificación de los casos de uso

A continuación se expondrán los diagramas para cada caso de uso de nuestra aplicación así como la especificación descriptiva de cada uno de ellos.

Inicio de la aplicación

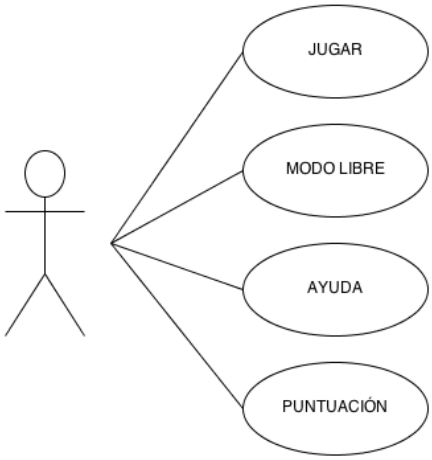


Figura 8.6: Caso de Uso: Inicio de la aplicación

Nombre	Inicio de la aplicación
Descripción	Se inicia la aplicación para que el usuario elija la acción a realizar
Actores	Usuario
Precondiciones	Tener instalada la aplicación en el dispositivo
Flujo principal	1. El usuario abre la aplicación 2. El usuario selecciona la acción a realizar
Postcondiciones	La aplicación lleva a cabo las operaciones necesarias para realizar la acción seleccionada por el usuario

Tabla 8.4: Caso de Uso: Inicio de la aplicación

Jugar

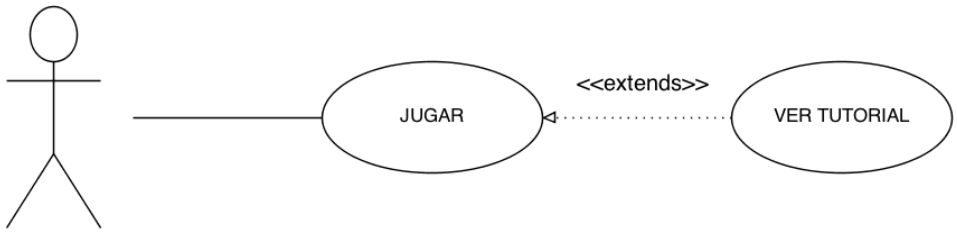


Figura 8.7: Caso de Uso: Jugar

Nombre	Jugar
Descripción	Permite al usuario la opcion de empezar una partida
Actores	Usuario
Precondiciones	Tener instalada la aplicación en el dispositivo
Flujo principal	1. La aplicación utiliza el caso de uso Ver tutorial 2. Permite al usuario empezar una partida
Postcondiciones	Comienza el juego

Tabla 8.5: Caso de Uso: Jugar

Ver tutorial

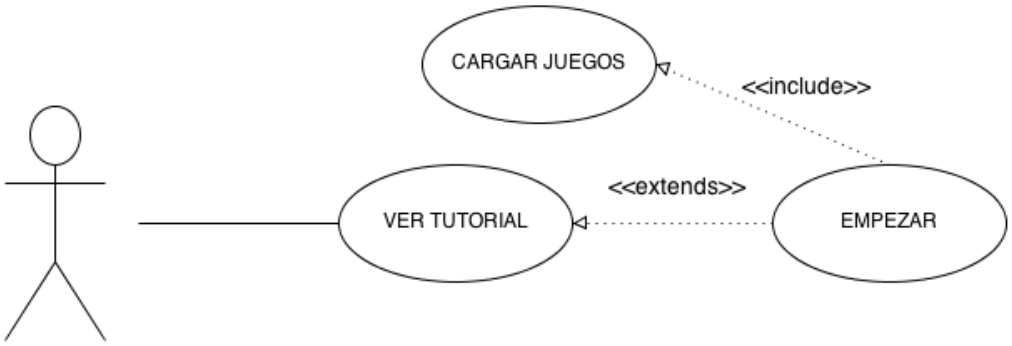


Figura 8.8: Caso de Uso: Ver tutorial

Nombre	Ver tutorial
Descripción	Muestra al usuario un tutorial de como funciona el videojuego
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Jugar
Flujo principal	El usuario visualiza el tutorial
Postcondiciones	Se inicia el tutorial

Tabla 8.6: Caso de Uso: Ver tutorial

Empezar

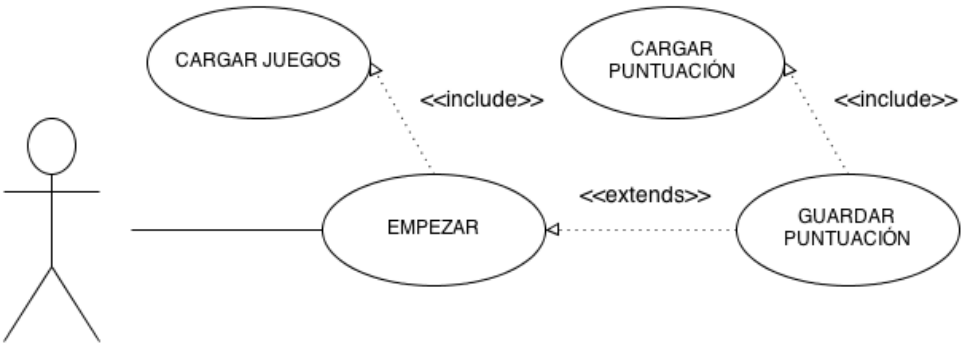


Figura 8.9: Caso de Uso: Empezar

Nombre	Empezar
Descripción	Empieza la partida, el usuario empieza a jugar
Actores	Usuario
Precondiciones	El usuario ha tenido que visualizar el tutorial
Flujo principal	Se utiliza el caso de uso Cargar juegos
Postcondiciones	Empieza la partida

Tabla 8.7: Caso de Uso: Empezar

Guardar puntuación

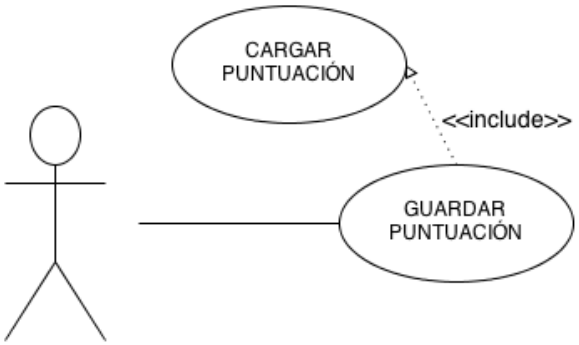


Figura 8.10: Caso de Uso: Guardar puntuación

Nombre	Guardar puntuación
Descripción	Guarda la puntuación obtenida en la partida
Actores	Usuario
Precondiciones	El usuario ha tenido que terminar la partida
Flujo principal	Se cargan las puntuaciones
Postcondiciones	Se guarda la puntuación

Tabla 8.8: Caso de Uso: Guardar puntuación

Juegos

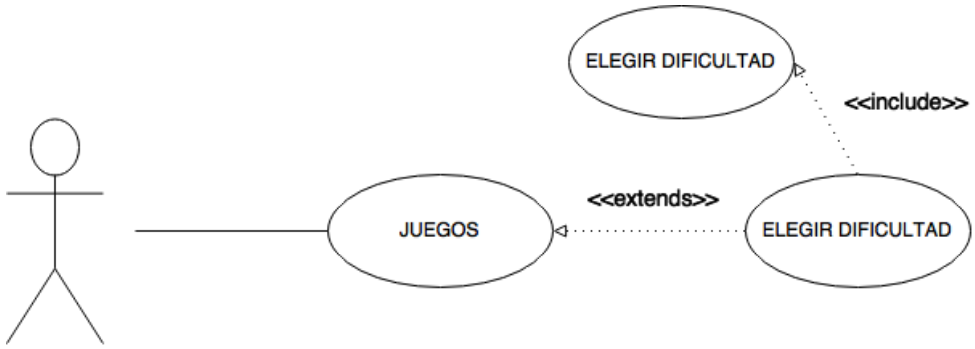


Figura 8.11: Caso de Uso: Juegos

Nombre	Juegos
Descripción	Permite al usuario realizar las pruebas de una manera libre
Actores	Usuario
Precondiciones	Tener instalada la aplicación en el dispositivo
Flujo principal	1. El usuario abre la aplicación 2. El usuario selecciona la acción a realizar
Postcondiciones	La aplicación lleva a cabo las operaciones necesarias para realizar la acción seleccionada por el usuario

Tabla 8.9: Caso de Uso: Juegos

Elegir dificultad

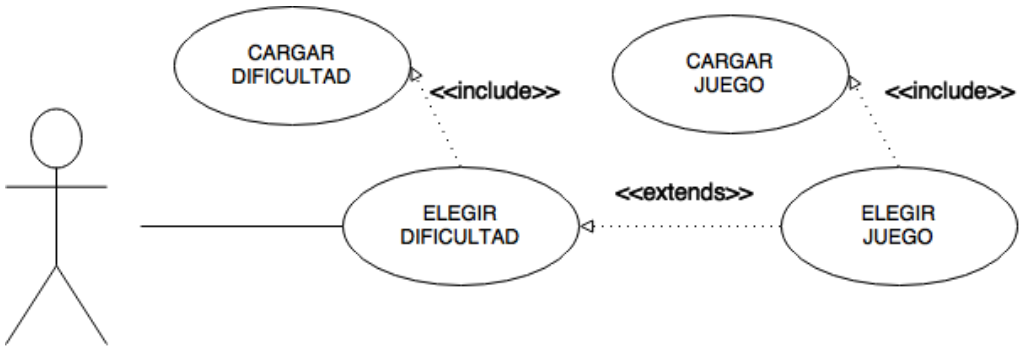


Figura 8.12: Caso de Uso: Elegir dificultad

Nombre	Elegir dificultad
Descripción	Se selecciona la dificultad previa al videojuego
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Juegos
Flujo principal	El usuario elige la dificultad
Postcondiciones	Se ha elegido la dificultad

Tabla 8.10: Caso de Uso: Elegir dificultad

Elegir juego

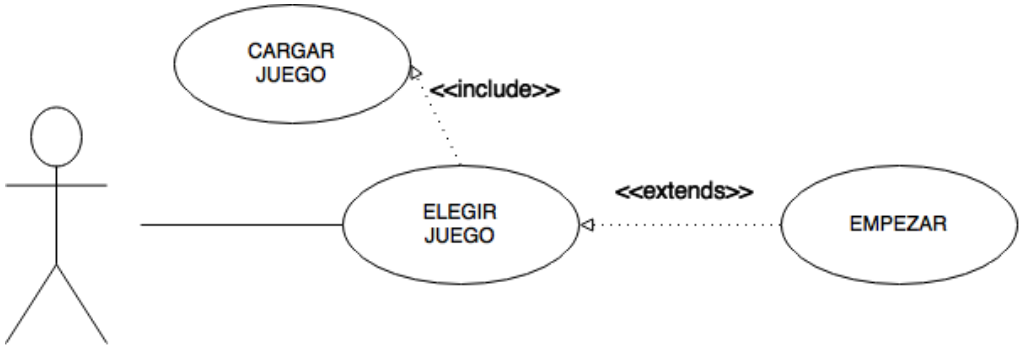


Figura 8.13: Caso de Uso: Elegir juego

Nombre	Elegir juego
Descripción	Se selecciona el juego
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Elegir dificultad
Flujo principal	El usuario elige el juego
Postcondiciones	Se ha elegido el juego

Tabla 8.11: Caso de Uso: Elegir juego

Empezar

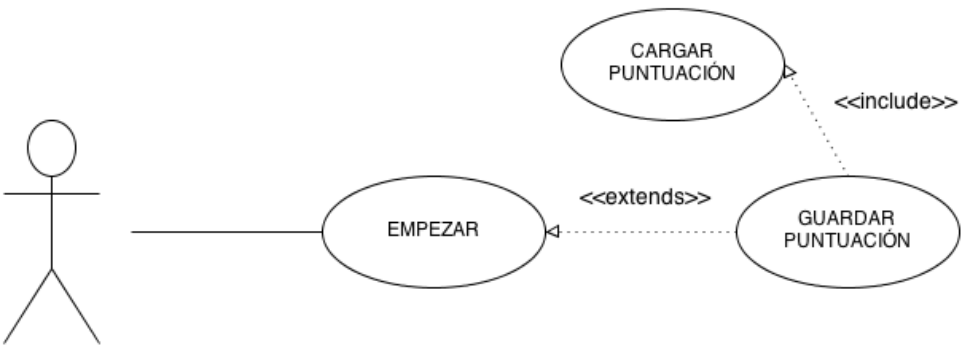


Figura 8.14: Caso de Uso: Empezar

Nombre	Empezar
Descripción	Empieza la partida con el juego y la dificultad seleccionada
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Elegir juego
Flujo principal	El usuario empieza a jugar
Postcondiciones	El usuario ha jugado

Tabla 8.12: Caso de Uso: Empezar

Guardar puntuación

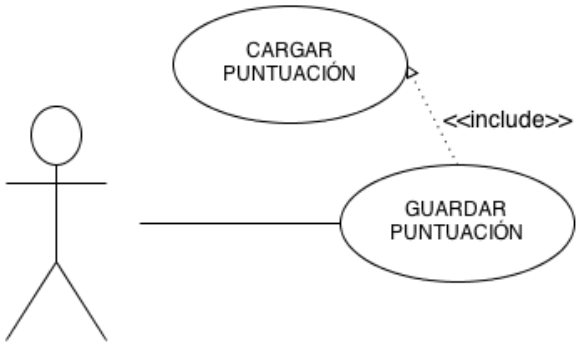


Figura 8.15: Caso de Uso: Guardar puntuación

Nombre	Guardar puntuación
Descripción	Guarda la puntuación obtenida en la partida en juegos
Actores	Usuario
Precondiciones	El usuario ha tenido que terminar la partida
Flujo principal	Se cargan las puntuaciones
Postcondiciones	Se guarda la puntuación

Tabla 8.13: Caso de Uso: Guardar puntuación

Puntuaciones

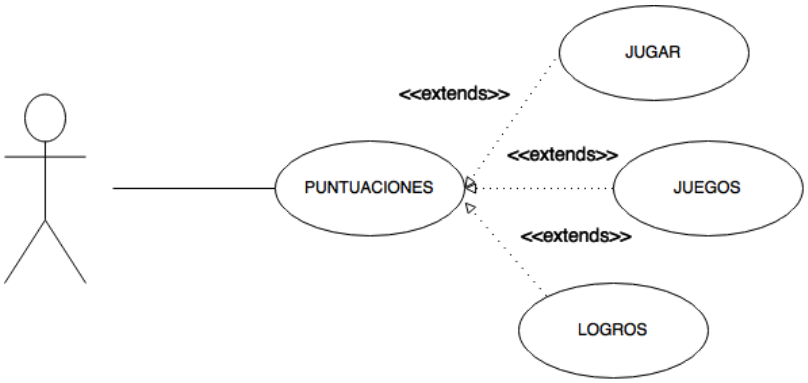


Figura 8.16: Caso de Uso: Puntuaciones

Nombre	Puntuaciones
Descripción	Se permite al usuario consultar las puntuaciones
Actores	Usuario
Precondiciones	Tener instalada la aplicación en el dispositivo
Flujo principal	1. El usuario abre la aplicación 2. El usuario selecciona la acción a realizar
Postcondiciones	La aplicación lleva a cabo las operaciones necesarias para realizar la acción seleccionada por el usuario

Tabla 8.14: Caso de Uso: Puntuaciones

Jugar

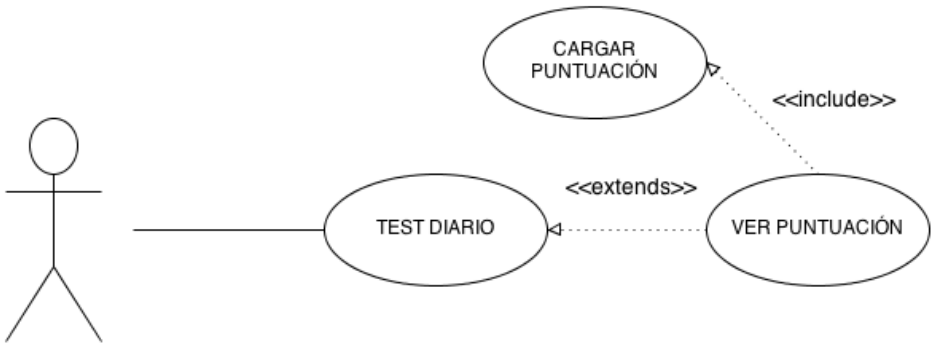


Figura 8.17: Caso de Uso: Jugar

Nombre	Jugar
Descripción	Puntuaciones obtenidas por todos los usuarios en el modo Jugar
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Puntuaciones
Flujo principal	El usuario selecciona las puntuaciones del modo Jugar
Postcondiciones	Se ha seleccionado las puntuaciones del modo Jugar

Tabla 8.15: Caso de Uso: Jugar

Ver puntuaciones

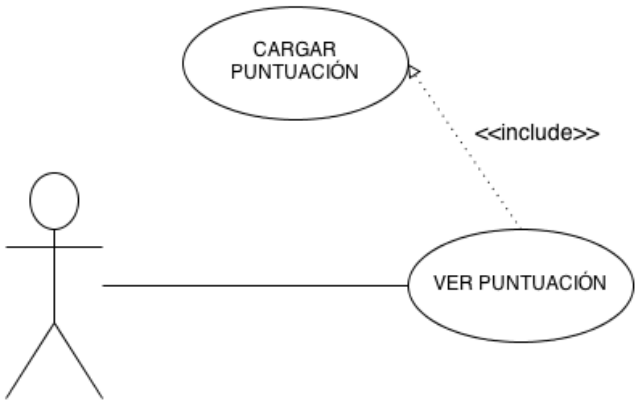


Figura 8.18: Caso de Uso: Ver puntuaciones

Nombre	Ver puntuaciones
Descripción	Muestra las puntuaciones obtenidas en el test diario
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Jugar
Flujo principal	El usuario consulta las puntuaciones del modo Jugar
Postcondiciones	Se ha consultado las puntuaciones del modo Jugar

Tabla 8.16: Caso de Uso: Ver puntuaciones

Juegos

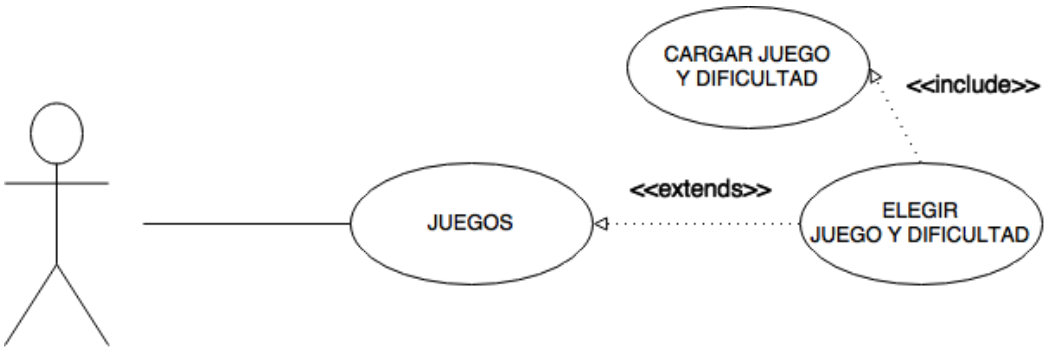


Figura 8.19: Caso de Uso: Juegos

Nombre	Juegos
Descripción	Puntuaciones obtenidas por juegos y dificultad
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Puntuaciones
Flujo principal	El usuario selecciona las puntuaciones por juego y dificultad
Postcondiciones	Se ha seleccionado las puntuaciones por juego y dificultad

Tabla 8.17: Caso de Uso: Juegos

Elegir juego y dificultad

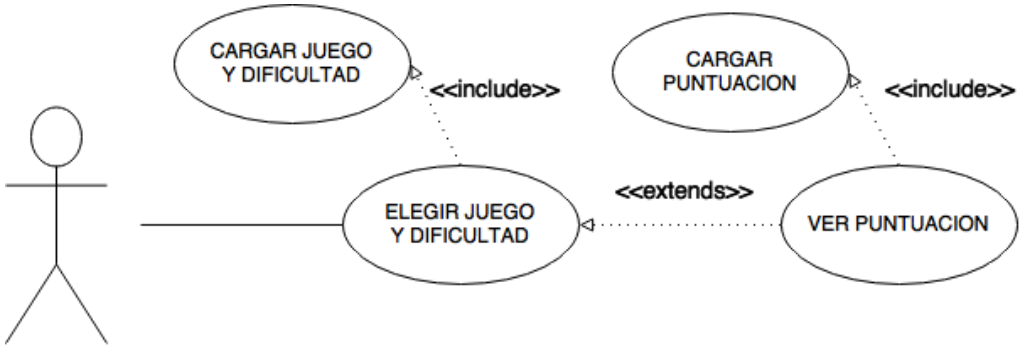


Figura 8.20: Caso de Uso: Elegir juego y dificultad

Nombre	Elegir Juego y dificultad
Descripción	Se selecciona de un listado de botones el juego y la dificultad que queramos consultar.
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Juego
Flujo principal	El usuario elige juego y dificultad
Postcondiciones	Se ha elegido juego y dificultad

Tabla 8.18: Caso de Uso: Elegir juego y dificultad

Ver puntuaciones

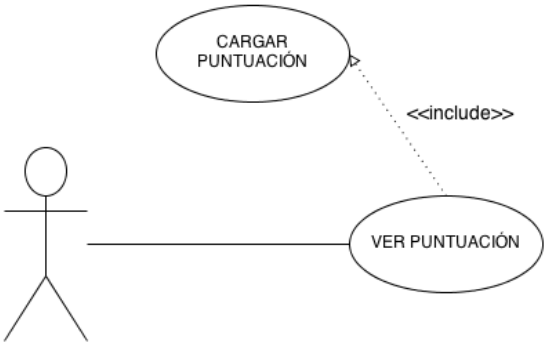


Figura 8.21: Caso de Uso: Ver puntuaciones

Nombre	Ver puntuaciones
Descripción	Muestra las puntuaciones obtenidas por categorías
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Elegir juego y dificultad
Flujo principal	El usuario consulta la puntuación por categoría, juego y dificultad
Postcondiciones	Se ha consultado la puntuación por categoría, juego y dificultad

Tabla 8.19: Caso de Uso: Ver puntuaciones

Ayuda

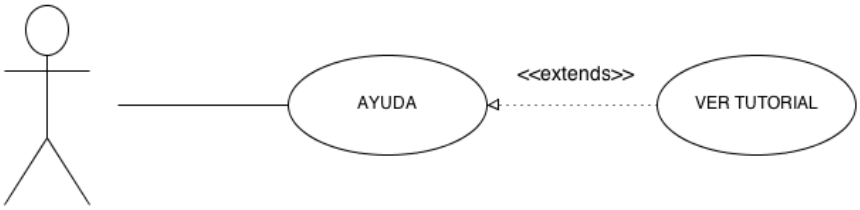


Figura 8.22: Caso de Uso: Ayuda

Nombre	Ayuda
Descripción	Muestra al usuario una sección con ayuda para poder jugar
Actores	Usuario
Precondiciones	Tener instalada la aplicación en el dispositivo
Flujo principal	1. El usuario abre la aplicación 2. El usuario selecciona la acción a realizar
Postcondiciones	La aplicación lleva a cabo las operaciones necesarias para realizar la acción seleccionada por el usuario

Tabla 8.20: Caso de Uso: Ayuda

Ver tutorial

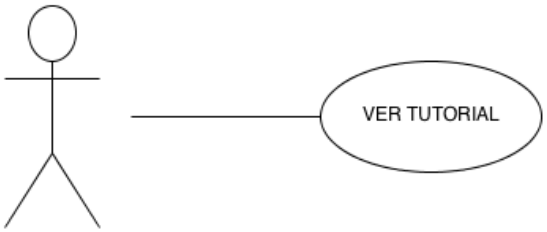


Figura 8.23: Caso de Uso: Ver tutorial

Nombre	Ver tutorial
Descripción	Muestra al usuario un tutorial de como funciona el videojuego
Actores	Usuario
Precondiciones	El caso de uso anterior debe ser Jugar
Flujo principal	El usuario visualiza el tutorial
Postcondiciones	Se inicia el tutorial

Tabla 8.21: Caso de Uso: Ver tutorial

Capítulo 9

Especificación del Sistema

Debido a que el sistema se basará en el paradigma de la programación orientada a objetos, se empleará el diagrama de clases [14] para presentar todas las clases usadas por la aplicación y las relaciones que hay entre las mismas. Además, se explicará la finalidad de las mismas y de los métodos que componen cada una de las clases.

9.1. Clases

El diagrama de clases es el representado en la figura (9.1).

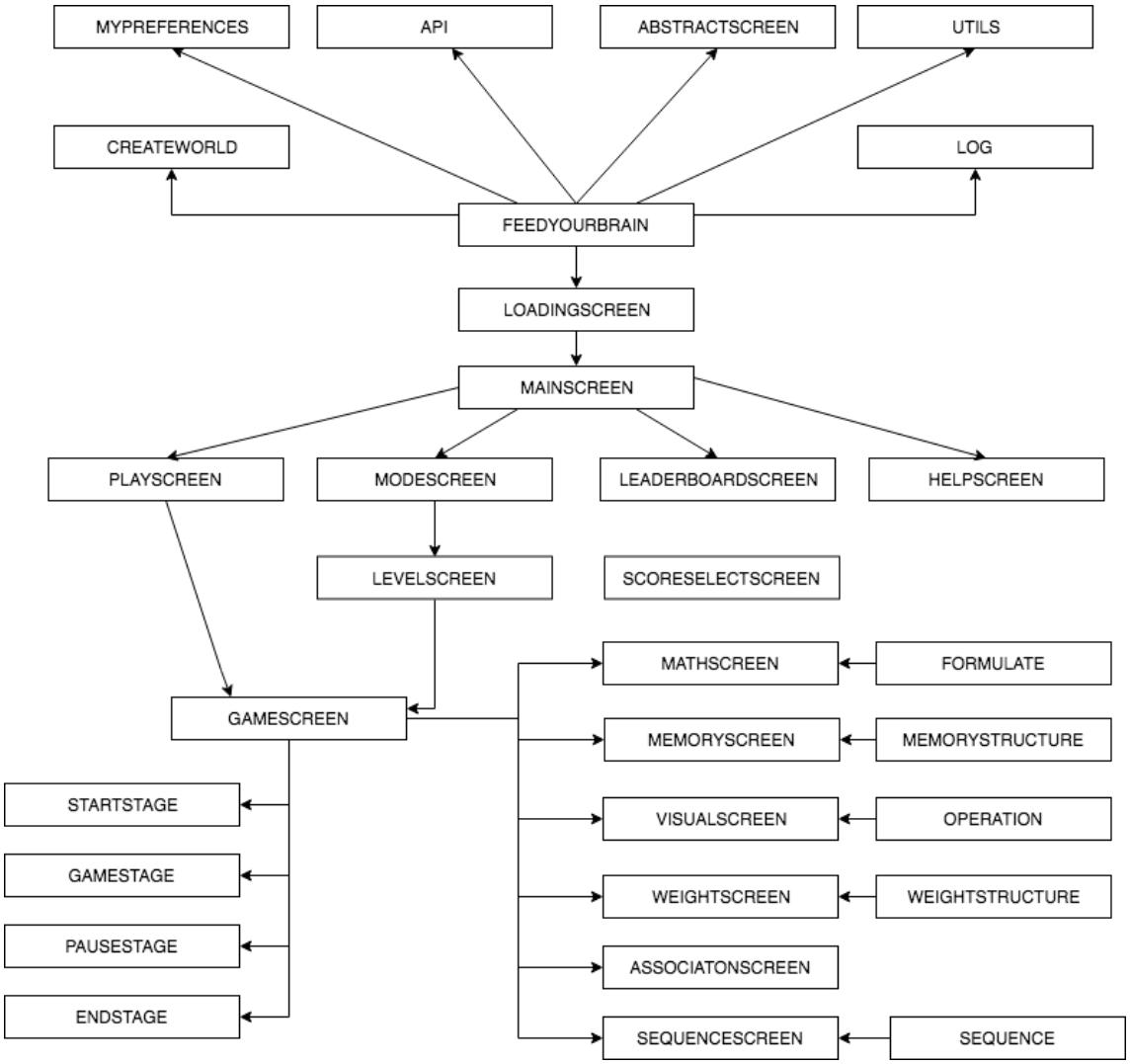


Figura 9.1: Diagrama de clases

9.1.1. FeedYourBrain

La clase FeedYourBrain extiende de la clase Game y es la clase principal del proyecto. Es la encargada de gestionar todos los recursos en memoria, instanciar la base de datos, realizar los cambios entre pantallas y reproducir los sonidos.

- **Atributos:**
  - **AbstractScreen mainScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla principal.
  - **AbstractScreen modeScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla de juegos.
  - **AbstractScreen helpScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla de ayuda.

- **AbstractScreen playScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla jugar.
- **AbstractScreen endScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla de fin de juego.
- **AbstractScreen levelScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla de selección de nivel.
- **AbstractScreen scoreSelectScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla de puntuación.
- **AbstractScreen leaderBoardScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla de tabla de lideres.
- **AbstractScreen creditsScreen.** Atributo privado de la clase AbstractScreen que almacena la pantalla de creditos.
- **GameScreen associationScreen.** Atributo privado de la clase GameScreen que almacena la pantalla del juego de asociación.
- **GameScreen visualScreen.** Atributo privado de la clase GameScreen que almacena la pantalla del juego de visual.
- **GameScreen mathScreen.** Atributo privado de la clase GameScreen que almacena la pantalla del juego matematicas.
- **GameScreen logicScreen.** Atributo privado de la clase GameScreen que almacena la pantalla del juego de logica.
- **GameScreen memoryScreen.** Atributo privado de la clase GameScreen que almacena la pantalla del juego de memoria.
- **GameScreen sequenceScreen.** Atributo privado de la clase GameScreen que almacena la pantalla del juego de secuencias.
- **GameScreen weightScreen.** Atributo privado de la clase GameScreen que almacena la pantalla del juego de pesos.
- **LoadingScreen screenLoading.** Atributo privado de la clase LoadingScreen que almacena la pantalla del juego.
- **AssetManager manager.** Atributo privado de la clase AssetManager que almacena el manejador de recursos.
- **SpriteBatch batch.** Atributo privado de la clase SpriteBatch que almacena el dibujador de texturas.
- **DatabaseFeedYourBrain databaseFeedYourBrain.** Atributo privado de la clase DatabaseFeedYourBrain que almacena el manejador de la base de datos.

▪ **Métodos públicos:**

- **create().** Método sobrescrito de la clase Game que es llamado cuando la clase es instanciada.
- **loadAssets().** Método que se encarga de cargar todas las recursos en memoria.
- **loadScreen().** Método que se encarga de cargar la primera pantalla.
- **dispose().** Método sobrescrito de la clase Game que se encarga de liberar toda la memoria de la aplicación.
- **render().** Método sobrescrito de la clase Game que es llamado cuando la aplicación se tiene que dibujar.

- **resize(entero width, entero height).** Método sobrescrito de la clase Game que se encarga de la redimensión de la pantalla, recibe por parámetro el ancho y el alto al que queremos redimensionar.
- **pause().** Método sobrescrito de la clase Game que es llamado cuando la aplicación entra en pausa.
- **resume().** Método sobrescrito de la clase Game que es llamado cuando la aplicación se reanuda desde un estado de pausa.
- **play().** Método que se encarga de iniciar el modo juego.
- **exitScreen(entero screen, entero type).** Método que se encarga de liberar la memoria y salir de la pantalla pasada en el primer parámetro, el segundo parámetro especifica si estamos en el modo jugar o en el modo juegos.
- **exitScreen(entero screen).** Método que se encarga de liberar la memoria y salir de la pantalla pasada por parametro.
- **nextGame(cadena nameChildClass, entero scoreNumber).** Método que se encarga de cambiar de juego en el modo jugar, cambia al juego especificado en el primer parámetro, el segundo parámetro especifica la puntuación conseguida en el juego anterior.
- **main().** Método que se encarga de llevarnos a la pantalla principal.
- **DatabaseFeedYourBrain getDatabaseFeedYourBrain().** Método que devuelve el manejador de la base de datos.
- **sound(entero type).** Método que se encarga de reproducir un sonido, recibe por parametro el tipo de sonido a reproducir.

9.1.2. LoadingScreen

La clase LoadingScreen implementa la interfaz Screen. Esta clase es la encargada de mostrar la animación de carga por pantalla mientras se realiza el volcado a memoria de los recursos.

- **Atributos:**
  - **cadena loading.** Atributo privado de la clase cadena que almacena la cadena a mostrar por pantalla en el proceso de carga.
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **boolean load.** atributo privado de tipo boolean que almacena el valor de si hemos terminado de cargar los recursos.
  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **ArrayList<Character>characters.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Character.
- **Métodos públicos:**

- **LoadingScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
- **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
- **dispose().** Método de la interfaz Screen que es llamado al liberar los recursos.
- **hide().** Método de la interfaz Screen que es llamado cuando esta pantalla no es la actual.
- **pause().** Método de la interfaz Screen que es llamado cuando la aplicación entra en pausa.
- **resize(entero arg0, entero arg1).** Método de la interfaz Screen que se encarga de la redimensión de la pantalla, recibe por parámetro el ancho y el alto al que queremos redimensionar
- **resume()..** Método de la interfaz Screen que es llamado cuando la aplicación se reanuda desde un estado de pausa.
- **show().** Método de la interfaz Screen que es llamado cuando esta pantalla pasa a ser la actual.

9.1.3. MainScreen

La clase MainScreen extiende de la clase AbstractScreen e implementa la interfaz Screen. Esta clase es la encargada de mostrar las opciones del menú principal.

- **Atributos:**
  - **Button mainButton.** Atributo privado de la clase Button que almacena el botón para acceder al modo juego.
  - **Button modeButton.** Atributo privado de la clase Button que almacena el botón para acceder al modo juegos.
  - **Button scoreButton.** Atributo privado de la clase Button que almacena el botón para acceder a las puntuaciones.
  - **Button helpButton.** Atributo privado de la clase Button que almacena el botón para acceder a la ayuda.
  - **Button exitButton.** Atributo privado de la clase Button que almacena el botón para salir de la aplicación.
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
- **Métodos públicos:**
  - **MainScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.

- **boolean keyDown(entero keycode).** Método de la interfaz InputProcessor que es llamado cuando una tecla es pulsada.
- **boolean keyUp(entero keycode) .** Método de la interfaz InputProcessor que es llamado cuando una tecla es liberada.
- **boolean keyTyped(char character).** Método de la interfaz InputProcessor que es llamado cuando una tecla es presionada.
- **boolean touchDown(entero screenX, entero screenY, entero pointer, entero button).** Método de la interfaz InputProcessor que es llamado cuando la pantalla es pulsada.
- **boolean touchUp(entero screenX, entero screenY, entero pointer, entero button) .** Método de la interfaz InputProcessor que es llamado cuando la pantalla es liberada.
- **boolean touchDragged(entero screenX, entero screenY, entero pointer).** Método de la interfaz InputProcessor que es llamado cuando arrastramos una pulsación en la pantalla.
- **boolean mouseMoved(entero screenX, entero screenY).** Método de la interfaz InputProcessor que es llamado cuando movemos un cursor con un ratón.
- **boolean scrolled(entero amount) .** Método de la interfaz InputProcessor que es llamado cuando realizamos un scroll.

- **Métodos privados:**
  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.
  - **goToLevel().** Método para acceder a la pantalla del modo juegos.
  - **goToPlay().** Método para acceder a la pantalla del modo jugar.
  - **goToScore().** Método para acceder a la pantalla de las puntuaciones.

9.1.4. PlayScreen

La clase PlayScreen extiende de la clase AbstractScreen e implementa la interfaz Screen. Esta clase es la encargada de mostrar las opciones antes de empezar el modo jugar.

- **Atributos:**
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **Description description.** Atributo privado de la clase Description que almacena el actor para representar la textura de la descripción del modo jugar.
  - **Title title.** Atributo privado de la clase Title que almacena el actor para representar la textura del titulo del modo jugar.
  - **MaxScore maxScore.** Atributo privado de la clase MaxScore que almacena el actor para representar la textura del texto de la máxima puntuación.
  - **Score scoreRight.** Atributo privado de la clase Score que almacena el actor para representar la textura de la cantidad de la máxima puntuación.

- **Button buttonStart.** Atributo privado de la clase Button que almacena el botón para empezar el modo jugar.
  item **Button buttonBack.** Atributo privado de la clase Button que almacena el botón para volver atrás.
- **ButtonClickListener buttonClickListener.** Atributo privado de la clase ButtonClickListener que almacena un Listener para los eventos de los botones.

- **Métodos públicos:**
  - **PlayScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **boolean keyDown(entero keycode).** Método de la interfaz InputProcessor que es llamado cuando una tecla es pulsada.
  - **boolean keyUp(entero keycode) .** Método de la interfaz InputProcessor que es llamado cuando una tecla es liberada.
  - **boolean keyTyped(char character).** Método de la interfaz InputProcessor que es llamado cuando una tecla es presionada.
  - **boolean touchDown(entero screenX, entero screenY, entero pointer, entero button).** Método de la interfaz InputProcessor que es llamado cuando la pantalla es pulsada.
  - **boolean touchUp(entero screenX, entero screenY, entero pointer, entero button) .** Método de la interfaz InputProcessor que es llamado cuando la pantalla es liberada.
  - **boolean touchDragged(entero screenX, entero screenY, entero pointer).** Método de la interfaz InputProcessor que es llamado cuando arrastramos una pulsación en la pantalla.
  - **boolean mouseMoved(entero screenX, entero screenY).** Método de la interfaz InputProcessor que es llamado cuando movemos un cursor con un ratón.
  - **boolean scrolled(entero amount) .** Método de la interfaz InputProcessor que es llamado cuando realizamos un scroll.

- **Métodos privados:**
  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadCustomView().** Método que se encarga de cargas las vistas propias de la clase.
  - **start().** Método para comenzar el modo jugar.
  - **back().** Método para volver atrás.

9.1.5. ModeScreen

La clase ModeScreen extiende de la clase AbstractScreen e implementa la interfaz Screen. Esta clase es la encargada de mostrar las opciones del modo juegos.

- **Atributos:**
  - **Button logicButton.** Atributo privado de la clase Button que almacena el botón para acceder al juego asociación.
  - **Button visualButton.** Atributo privado de la clase Button que almacena el botón para acceder al juego visual.
  - **Button mathButton.** Atributo privado de la clase Button que almacena el botón para acceder al juego matemáticas.
  - **Button memoryButton.** Atributo privado de la clase Button que almacena el botón para acceder al juego memoria.
  - **Button backButton.** Atributo privado de la clase Button que almacena el botón para volver a la pantalla de atrás.
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
- **Métodos públicos:**
  - **ModeScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **boolean keyDown(entero keycode).** Método de la interfaz InputProcessor que es llamado cuando una tecla es pulsada.
  - **boolean keyUp(entero keycode) .** Método de la interfaz InputProcessor que es llamado cuando una tecla es liberada.
  - **boolean keyTyped(char character).** Método de la interfaz InputProcessor que es llamado cuando una tecla es presionada.
  - **boolean touchDown(entero screenX, entero screenY, entero pointer, entero button).** Método de la interfaz InputProcessor que es llamado cuando la pantalla es pulsada.
  - **boolean touchUp(entero screenX, entero screenY, entero pointer, entero button) .** Método de la interfaz InputProcessor que es llamado cuando la pantalla es liberada.
  - **boolean touchDragged(entero screenX, entero screenY, entero pointer).** Método de la interfaz InputProcessor que es llamado cuando arrastramos una pulsación en la pantalla.
  - **boolean mouseMoved(entero screenX, entero screenY).** Método de la interfaz InputProcessor que es llamado cuando movemos un cursor con un ratón.
  - **boolean scrolled(entero amount) .** Método de la interfaz InputProcessor que es llamado cuando realizamos un scroll.
- **Métodos privados:**
  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadCustomView().** Método que se encarga de cargas las vistas propias de la clase.

- **goToVisual().** Método para acceder a la pantalla del juego visual.
- **goToLogic().** Método para acceder a la pantalla del juego asociación.
- **goToMath().** Método para acceder a la pantalla del juego matemáticas.
- **goToMemory().** Método para acceder a la pantalla de juego memoria.

9.1.6. LeaderBoardScreen

La clase LeaderBoardScreen extiende de la clase AbstractScreen y es la encargada de mostrar los botones para acceder a las tablas de lideres.

- **Atributos:**
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **Button backButton.** Atributo privado de la clase Button que almacena el botón para volver atrás.
- **Métodos públicos:**
  - **LeaderBoardScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
- **Métodos privados:**
  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.7. LevelScreen

La clase LevelScreen extiende de la clase AbstractScreen y es la encargada de mostrar los botones para la selección de nivel.

- **Atributos:**
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
- **Métodos públicos:**
  - **LeaderBoardScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.

  - **goToMode().** Método que se encarga de liberar la memoria de esta clase y volver a la pantalla de selección de juego.

- **Métodos privados:**

  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.8. ScoreSelectScreen

La clase ScoreSelectScreen extiende de la clase AbstractScreen. Esta clase es la encargada de mostrar las opciones para seleccionar las distintas puntuaciones.

- **Atributos:**

  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **Button mainButton.** Atributo privado de la clase Button que almacena el botón para acceder a las puntuaciones del modo jugar.
  - **Button modeButton.** Atributo privado de la clase Button que almacena el botón para acceder a las puntuaciones del modo juegos.
  - **Button generalButton.** Atributo privado de la clase Button que almacena el botón para acceder a los logros

- **Métodos públicos:**

  - **ScoreSelectScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.

- **Métodos privados:**

  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadCustomView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.9. GameScreen

La clase GameScreen implementa la interfaz Screen. Esta clase es la encargada de llevar los procesos de comienzo, pausa, reinicio y salida de todos los juegos de la aplicación así como de mostrar mensajes y gestionar las puntuaciones.

- **Atributos:**
  - **FeedYourBrain feedYourBrain.** Atributo protegido de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **StartStage startStage.** Atributo protegido de la clase StartStage que almacena el escenario donde mostraremos los actores de la pantalla de comienzo.
  - **Stage messageStage.** Atributo protegido de la clase Stage que almacena el escenario donde mostraremos los actores de los mensajes.
  - **Stage stage.** Atributo protegido de la clase Stage que almacena el escenario donde mostraremos los actores que hereden de esta clase.
  - **GameStage gameStage.** Atributo protegido de la clase GameStage que almacena el escenario donde mostraremos los actores de la pantalla de juego.
  - **PauseStage pauseStage.** Atributo protegido de la clase PauseStage que almacena el escenario donde mostraremos los actores de la pantalla de pausa.
  - **EndStage endStage.** Atributo protegido de la clase EndStage que almacena el escenario donde mostraremos los actores de la pantalla de fin de juego.
  - **Message message.** Atributo protegido de la clase Message que almacena el actor para representar mensajes en pantalla.
  - **entero state.** Atributo protegido de tipo entero que almacena el estado del juego, si es 0 esta sin empezar, si es 1 esta en mitad del juego y si es 2 esta en pausa.
  - **boolean end.** Atributo protegido de tipo boolean que almacena si la partida ha finalizado.
  - **InputMultiplexer multiplexer.** Atributo privado de la clase InputMultiplexer que almacena los distintos procesadores que tendrá nuestro juego.
  - **entero type.** Atributo privado de tipo entero que almacena el tipo de modo de juego, si es 0 estamos en modo jugar y si es 1 estamos en modo juegos.
- **Métodos públicos:**
  - **GameScreen(FeedYourBrain feedyourbrain, entero type).** Constructor parametrizado de la clase.
  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **start().** Método que se encarga de comenzar el juego..
  - **pause().** Método que se encarga de pausar el juego.
  - **resume().** Método que se encarga de reanudar el juego.
  - **restart().** Método que se encarga de reiniciar el juego.
  - **exit().** Método que se encarga de salir del juego
  - **finish().** Método que se encarga de finalizar el juego.

- **saveScore().** Método que se encarga de almacenar la puntuación obtenida en cada partida.
- **boolean onGame(flotante delta).** Método que se encarga de comprobar si estamos en el estado jugando. Si devuelve true estamos en el estado jugando y si devuelve false no.
- **boolean onStart().** Método que se encarga de comprobar si estamos en el estado empezar. Si devuelve true estamos en el estado empezar y si devuelve false no.
- **boolean onMessage().** Método que se encarga de comprobar si estamos en el estado mensaje. Si devuelve true estamos en el estado mensaje y si devuelve false no.
- **boolean onPause().** Método que se encarga de comprobar si estamos en el estado pausa. Si devuelve true estamos en el estado pausa y si devuelve false no.
- **boolean onEnd().** Método que se encarga de comprobar si estamos en el estado finalizado. Si devuelve true estamos en el estado finalizado y si devuelve false no.
- **increase().** Método que se encarga de incrementar la puntuación y de informar que se ha producido un acierto en un juego.
- **noincrease().** Método que se encarga de informar que se ha producido un error en un juego.
- **dispose().** Método de la interfaz Screen que es llamado al liberar los recursos.
- **hide().** Método de la interfaz Screen que es llamado cuando esta pantalla no es la actual.
- **resize(entero arg0, entero arg1).** Método de la interfaz Screen que se encarga de la redimensión de la pantalla, recibe por parámetro el ancho y el alto al que queremos redimensionar
- **show().** Método de la interfaz Screen que es llamado cuando esta pantalla pasa a ser la actual.

9.1.10. StartStage

La clase StartStage extiende de la clase Stage y es la encargada mostrar las opciones de comienzo de un juego en el modo juegos.

- **Atributos:**
  - **FeedYourBrain feedYourBrain.** Atributo de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **ActorGestureListener actorGestureListener.** Atributo privado de la clase ActorGestureListener que almacena un Listener para los eventos de los actores.

- **cadena nameChildClass.** Atributo privado de la clase cadena que almacena el nombre de la cual es instanciada esta clase.
- **StartGameTitle startGameTitle.** Atributo privado de la clase StartGameTitle que almacena el actor que representa la textura del titulo del juego.
- **StartGameDescription startGameDescription.** Atributo privado de la clase StartGameTitle que almacena el actor que representa la textura del titulo de la descripción del juego.
- **StartGameTitle startGameScoreMax.** Atributo privado de la clase StartGameTitle que almacena el actor que representa la textura del titulo de la máxima puntuación.
- **Button buttonStart.** Atributo privado de la clase Button que almacena el botón para empezar la partida.
- **Button buttonBack.** Atributo privado de de la clase Button que almacena el botón para volver atrás.

▪ **Métodos públicos:**

- **StartStage(FeedYourBrain fyb,ActorGestureListener actorGestureListener, cadena nameChildClass).** Constructor parametrizado de la clase.
- **loadActors().** Método para cargar los actores en pantalla.
- **remove().** Método para borrar los actores.

▪ **Métodos privados:**

- **loadView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.11. GameStage

La clase GameStage extiende de la clase Stage y es la encargada mostrar las opciones por defecto de un juego en el modo juegos.

▪ **Atributos:**

- **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
- **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
- **ActorGestureListener actorGestureListener.** Atributo privado de la clase ActorGestureListener que almacena un Listener para los eventos de los actores.
- **Time time.** Atributo privado de la clase Time que almacena un actor que reprensenta la textura para mostar el tiempo.
- **flotante seconds.** Atributo privado de la clase flotante que almacena el tiempo que llevamos en el juego.
- **Pause pause.** Atributo privado de la clase Pause que almacena el actor que representa la textura para mostrar la pausa.

- **Sound sound.** Atributo privado de la clase Sound que almacena el actor que representa la textura para mostrar el sonido.
- **Streak streak.** Atributo privado de la clase Streak que almacena el actor que representa la textura para mostrar la racha.
- **Score score.** Atributo privado de la clase Score que almacena el actor que representa la textura para mostrar la puntuación.
- **Ok ok.** Atributo privado de de la clase Ok que almacena el actor que representa la textura para mostrar un acierto en el juego.
- **No no.** Atributo privado de la clase No que almacena el actor que representa la textura para mostrar un error en el juego.
- **flotante TIME.** Atributo privado de la clase flotante que almacena el tiempo total de una partida.

- **Métodos públicos:**
  - **GameStage(FeedYourBrain fyb, ActorGestureListener actorGestureListener, entero type).** Constructor parametrizado de la clase.
  - **loadActors().** Método para cargar los actores en pantalla.
  - **update(flotante delta).** Método para actualizar el tiempo del juego.
  - **restart().** Método para reiniciar el juego.
  - **ok().** Método para mostrar por pantalla un acierto en el juego.
  - **no().** Método para mostrar por pantalla un error en el juego.
- **Métodos privados:**
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.12. PauseStage

La clase PauseStage extiende de la clase Stage y es la encargada mostrar las opciones de pausa en los juegos en cualquier modo.

- **Atributos:**
  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **ActorGestureListener actorGestureListener.** Atributo privado de la clase ActorGestureListener que almacena un Listener para los eventos de los actores.
- **Métodos públicos:**
  - **PauseStage(FeedYourBrain fyb, ActorGestureListener actorGestureListener, entero type).** Constructor parametrizado de la clase.
- **Métodos privados:**
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.13. EndStage

La clase EndStage extiende de la clase Stage y es la encargada mostrar las opciones de finalización de un juego en el modo juegos.

- **Atributos:**
  - **FeedYourBrain feedYourBrain.** Atributo de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **ActorGestureListener actorGestureListener.** Atributo privado de la clase ActorGestureListener que almacena un Listener para los eventos de los actores.
  - **cadena nameChildClass.** Atributo privado de la clase cadena que almacena el nombre de la cual es instanciada esta clase.
  - **StartGameTitle startGameTitle.** Atributo privado de la clase StartGameTitle que almacena el actor que representa la textura del titulo del juego.
  - **StartGameTitle startGameScoreMax.** Atributo privado de la clase StartGameTitle que almacena el actor que representa la textura del titulo de la máxima puntuación.
  - **StartGameTitle startGameScoreYour.** Atributo privado de la clase StartGameTitle que almacena el actor que representa la textura del titulo de nuestra puntuación.
  - **Score scoreYour.** Atributo privado de la clase Score que almacena el actor que representa la textura de la cantidad de la máxima puntuación.
  - **Score scoreMax.** Atributo privado de la clase Score que almacena el actor que representa la textura de la cantidad de nuestra puntuación.
  - **Button buttonRestart.** Atributo privado de la clase Button que almacena el botón para reiniciar la partida.
  - **Button buttonExit.** Atributo privado de de la clase Button que almacena el botón para salir del juego.
- **Métodos públicos:**
  - **EndStage(FeedYourBrain fyb,ActorGestureListener actorGestureListener, cadena nameChildClass).** Constructor parametrizado de la clase.
  - **loadActors().** Método para cargar los actores en pantalla.
  - **remove().** Método para borrar los actores.
  - **sendScore(entero score).** Método para asignar la puntuación.
  - **setAchievements(cadena code).** Método para desbloquear los logros.
- **Métodos privados:**
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.14. MathScreen

La clase MathScreen extiende de la clase GameScreen y es la encargada de representar el juego matemáticas.

- **Atributos:**
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **ArrayList<CharacterCalc>charactersCalc.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase CharacterCalc.
  - **Formulate formulate.** Atributo privado de la clase Formulate que almacena el sistema de generación de formulas.
  - **entero nHit.** Atributo privado de tipo entero que almacena el numero de aciertos.
  - **entero streak.** Atributo privado de tipo entero que almacena la racha.
- **Métodos públicos:**
  - **MathScreen(FeedYourBrain fyb, entero type).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **start().** Método sobreescrito de la clase GameScreen, este método es llamado cuando empezamos el juego.
  - **pause().** Método sobreescrito de la clase GameScreen, este método es llamado cuando pausamos el juego.
  - **resume().** Método sobreescrito de la clase GameScreen, este método es llamado cuando reanudamos el juego desde una pausa.
  - **restart().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reiniciamos el juego.
  - **exit().** Método sobrescrito de la clase GameScreen, este método es llamado cuando salimos el juego.
  - **.**
- **Métodos privados:**
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.
  - **setDifficulty().** Método para asignar la dificultad, la dificultad es asignada según en el modo de juego que nos encontremos así como el numero de aciertos que llevemos realizado.

9.1.15. Formulate

La clase Formulate es la clase encargada de generar las formulas para el juego matemáticas.

- **Atributos:**
  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **ArrayList<CharacterCalc>characters.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase CharacterCalc.
  - **cadena formulate.** Atributo privado de la clase cadena que almacena la formula en formato String.
  - **entero iResult.** Atributo privado de tipo entero que almacena el resultado de la formula.
  - **cadena sResult.** Atributo privado de la clase cadena que almacena el resultado en formato String.
  - **entero difficulty.** Atributo privado de tipo entero que almacena la dificultad.
- **Métodos públicos:**
  - **Formulate(FeedYourBrain feedyourbrain, Stage stage).** Constructor parametrizado de la clase.
  - **generateFormulate().** Método para generar una formula.
  - **entero checkResult().** Método para comprobar el resultado. Si es 0 aun no tenemos resultado, si es 1 es correcto y si es 2 es incorrecto.

9.1.16. MemoryScreen

La clase MemoryScreen extiende de la clase GameScreen y es la encargada de representar el juego memoria.

- **Atributos:**
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **MemoryStructure memoryStructure.** Atributo privado de la clase MemoryStructure que almacena el sistema de generación del juego de memoria.
  - **entero nHit.** Atributo privado de tipo entero que almacena el numero de aciertos.

- **entero streak.** Atributo privado de tipo entero que almacena la racha.

- **Métodos públicos:**

  - **MemoryScreen(FeedYourBrain fyb, entero type).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **start().** Método sobrescrito de la clase GameScreen, este método es llamado cuando empezamos el juego.
  - **pause().** Método sobrescrito de la clase GameScreen, este método es llamado cuando pausamos el juego.
  - **resume().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reanudamos el juego desde una pausa.
  - **restart().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reiniciamos el juego.
  - **exit().** Método sobrescrito de la clase GameScreen, este método es llamado cuando salimos el juego.

- **Métodos privados:**

  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadCustomView().** Método que se encarga de cargas las vistas propias de la clase.
  - **checkCard(Card card).** Método que se encarga de comprobar cada carta que pulsamos. Si la carta es correcta el turno continua y si es incorrecta se reinicia el turno.
  - **setDifficulty().** Método para asignar la dificultad, la dificultad es asignada según en el modo de juego que nos encontremos así como el numero de aciertos que llevemos realizado.

9.1.17. MemoryStructure

La clase MemoryStructure es la encargada de gestionar toda la lógica y estructura del juego memoria.

- **Atributos:**

  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **ActorGestureListener actorGestureListener.** Atributo privado de la clase ActorGestureListener que almacena un Listener para los eventos de los actores.

- **ArrayList<Card>cardsUp.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Card. Este listado representa las cartas que mostramos en la fila de arriba.
- **ArrayList<Card>cardsDown.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Card. Este listado representa las cartas que mostramos en la fila de abajo.
- **ArrayList<Card>cardsToFind.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Card. Este listado representa las cartas a buscar.
- **ArrayList<Integer>types.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Integer. Este listado representa los distintos tipos de cartas.
- **entero difficulty.** Atributo privado de tipo entero que almacena la dificultad.
- **flotante time.** Atributo privado de tipo flotante que almacena el tiempo del juego.
- **Boolean heads.** Atributo privado de la clase Boolean que almacena si las cartas estan mostrando la cara o la cruz.
- **Boolean tumble.** Atributo privado de la clase Boolean que almacena si las cartas estan volteandose.
- **Boolean changeStructure.** Atributo privado de la clase Boolean que almacena si ha habido cambios en la estructura.
- **flotante timeStructure.** Atributo privado de tipo flotante que almacena el tiempo en tardar a otra estructura.

▪ **Métodos públicos:**

- **MemoryStructure(FeedYourBrain fyb, Stage stage, ActorGestureListener actorGestureListener).** Constructor parametrizado de la clase.
- **generate().** Método para generar una nueva estructura.
- **entero getRandomType().** Método que devuelve un tipo de carta de forma aleatoria.
- **loadActors().** Método para cargar los actores en pantalla.
- **Boolean isHead().** Método que devuelve si estamos viendo las caras o las cruces de las cartas.
- **updateTime(flotante delta).** Método para actualizar el tiempo.
- **loadActorsDown().** Método para cargar los actores de la fila de abajo.
- **boolean isTumble().** Método que devuelve un booleano que representa si las cartas estan volteandose.
- **updateTimeForTumble(flotante delta).** Método para actualizar el tiempo de volteado de las cartas.
- **entero checkCard(Card cardToFind).** Método para comprobar la carta que hemos pulsado. Si devuelve 0 es que la carta no ha sido encontrada, si devuelve 1 es que la carta ha sido encontrada y si devuelve 2 es que es la ultima carta encontrada.

- **clear().** Método para limpiar la estructura.
- **updateTimeForChange(flotante delta).** Método para actualizar el tiempo de cambios en la estructura.
- **tailToHeadAllCards().** Método que voltea hacia arriba todas las cartas,
- **boolean isChange().** Método que devuelve si se asignan cambios a la estructura.

- **Métodos privados:**

  - **entero getNCards().** Método que nos devuelve el numero de cartas en base a la dificultad.
  - **Texture getTextureBy(entero type).** Método que nos devuelve la textura en base al parametro pasado.
  - **tumbleCard().** Método para asignar las cartas que se voltean.
  - **generateDown.** Método para generar las cartas de la fila de abajo.
  - **entero getNCardsHide().** Método que devuelve el numero de cartas que se ocultan en base a la dificultad.

9.1.18. VisualScreen

La clase VisualScreen extiende de la clase GameScreen y es la encargada de representar el juego visual.

- **Atributos:**

  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **World world**. Atributo privado de la clase World que almacena el mundo virtual donde mostraremos nuestras texturas.
  - **Box2DDebugRenderer debugRenderer**. Atributo privado de la clase Box2DDebugRenderer que almacena el mundo virtual donde mostraremos nuestras texturas para realizar pruebas.
  - **OrthographicCamera cam**. Atributo privado de la clase OrthographicCamera que almacena la cámara desde donde veremos el mundo virtual.
  - **boolean debugMode**. Atributo privado de tipo boolean que almacena si estamos modo prueba.
  - **Operation operation.** Atributo privado de la clase Formulate que almacena el sistema de generación de operaciones.
  - **entero nHit.** Atributo privado de tipo entero que almacena el numero de aciertos.
  - **entero streak.** Atributo privado de tipo entero que almacena la racha.

- **Métodos públicos:**
  - **VisualScreen(FeedYourBrain fyb, entero type).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **start().** Método sobrescrito de la clase GameScreen, este método es llamado cuando empezamos el juego.
  - **pause().** Método sobrescrito de la clase GameScreen, este método es llamado cuando pausamos el juego.
  - **resume().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reanudamos el juego desde una pausa.
  - **restart().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reiniciamos el juego.
  - **exit().** Método sobrescrito de la clase GameScreen, este método es llamado cuando salimos el juego.
- **Métodos privados:**
  - **loadCustomView().** Método que se encarga de cargar las vistas personalizadas del juego.
  - **checkCircle(Circle circle).** Método que se encarga de comprobar cada vez que pulsamos un circulo si es correcto o es incorrecto.
  - **setDifficulty().** Método para asignar la dificultad, la dificultad es asignada según en el modo de juego que nos encontremos así como el numero de aciertos que llevemos realizado.

9.1.19. Operation

La clase Operation es la clase encargada de realizar las operaciones para el juego visual.

- **Atributos:**
  - **ArrayList<Circle>circles.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Circle. Estos atributos representar las bolas dibujadas en el juego visual.
  - **ArrayList<Integer>numbers.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Interger. Estos atributos representan los números de las bolas dibujadas en el juego visual.
  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **ActorGestureListener actorGestureListener.** Atributo privado de la clase ActorGestureListener que almacena un Listener para los eventos de los actores.

- **entero difficulty.** Atributo privado de tipo entero que almacena la dificultad.

- **Métodos públicos:**

  - **Operation(FeedYourBrain feedyourbrain, ActorGestureListener actorGestureListener).** Constructor parametrizado de la clase.
  - **setOperation(World world).** Método para generar una nueva operación, recibe por parámetro el mundo virtual donde se dibujaran los elementos.
  - **ArrayList<Circle>getCircles().** Método para devolver el atributo circles.
  - **entero check(Circle circle).** Método que comprueba si el circulo pulsado es correcto en la operación, recibe por parámetro el circulo pulsado. Devuelve 0 si el circulo pulsado es incorrecto, 1 si es correcto y 2 si es correcto y es el ultimo para terminar la operación.
  - **clearBodies(World world).** Método para limpiar todos los cuerpos físicos en el mundo recibido por parámetro.
  - **ArrayList<Integer>getNumbers().** Método que devuelve el ArrayList de numeros.
  - **ArrayList<Vector2>generatePoints(entero nPoints).** Método para generar de forma aleatoria el numero de puntos recibidos por parámetros. Estos puntos hacen referencia a donde se dibujaran las pelotas al empezar el turno.
  - **double getDistance(Vector2 point1, Vector2 point2).** Método para obtener la distancia entre dos puntos, los puntos son pasados por parámetro.

9.1.20. WeightScreen

La clase WeightScreen extiende de la clase GameScreen y es la encargada de representar el juego de pesos.

- **Atributos:**

  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **WeightStructure weightStructure.** Atributo privado de la clase WeightStructure que almacena el sistema de generación del juego de pesos.
  - **entero nHit.** Atributo privado de tipo entero que almacena el numero de aciertos.
  - **entero streak.** Atributo privado de tipo entero que almacena la racha.

- **Métodos públicos:**

  - **WeightScreen(FeedYourBrain fyb, entero type).** Constructor parametrizado de la clase.

- **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
- **start().** Método sobrescrito de la clase GameScreen, este método es llamado cuando empezamos el juego.
- **pause().** Método sobrescrito de la clase GameScreen, este método es llamado cuando pausamos el juego.
- **resume().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reanudamos el juego desde una pausa.
- **restart().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reiniciamos el juego.
- **exit().** Método sobrescrito de la clase GameScreen, este método es llamado cuando salimos el juego.

- **Métodos privados:**
  - **loadCustomView().** Método que se encarga de cargar las vistas personalizadas del juego.
  - **result().** Método que se encarga de comprobar cada vez que pulsamos un peso si es correcto o es incorrecto.
  - **setDifficulty().** Método para asignar la dificultad, la dificultad es asignada según en el modo de juego que nos encontremos así como el numero de aciertos que llevemos realizado.

9.1.21. WeightStructure

La clase WeightStructure es la encargada de gestionar toda la lógica y estructura del juego de pesos.

- **Atributos:**
  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **ArrayList<Balance>balances.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los balanzas del juego.
  - **ArrayList<Weight>weights.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los pesos de cada balanza.
  - **ArrayList<Weight>weightsDown.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los pesos a seleccionar.
  - **ActorGestureListener actorGestureListener.** Atributo privado de la clase ActorGestureListener que almacena un Listener para los eventos de los actores.

- **entero difficulty.** Atributo privado de tipo entero que almacena la dificultad.
- **flotante time.** Atributo privado de tipo flotante que almacena el tiempo del juego.

- **Métodos públicos:**

  - **WeightStructure(FeedYourBrain fyb, Stage stage, ActorGestureListener actorGestureListener).** Constructor parametrizado de la clase.
  - **generate().** Método para generar una nueva secuencia.
  - **load().** Método para cargar los actores.
  - **loadActors().** Método para cargar los actores en pantalla.
  - **shuffleWeights().** Método para reordenar los pesos de forma aleatoria.
  - **updateTime(flotante delta).** Método para actualizar el tiempo.
  - **boolean checkWeight(Weight weightToFind).** Método para comprobar el resultado al seleccionar un peso.

- **Métodos privados:**

  - **Texture getTextureBy(entero type).** Método que nos devuelve la textura en base al parametro pasado.
  - **Weight findWeightByLetter(cadena letter).** Método que nos devuelve el peso en base al parametro pasado.

9.1.22. AssociationScreen

La clase AssociationScreen extiende de la clase GameScreen y es la encargada de representar el juego asociación.

- **Atributos:**

  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
  - **ArrayList<Card>cards.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Cards.
  - **ArrayList<Shape>shapes.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Shape.
  - **ArrayList<Integer>numbersRepeats.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Integer.
  - **Card cardSelected.** Atributo privado de la clase Card que almacena la carta que tenemos seleccionada en cada momento.
  - **entero nCoupleHit.** Atributo privado de tipo entero que almacena el numero de parejas acertadas.

- **entero nHit.** Atributo privado de tipo entero que almacena el numero de aciertos.
- **entero streak.** Atributo privado de tipo entero que almacena la racha.
- **cadena shapesNames[].** Array de atributos privados de la clase cadena que almacenan los nombres de las texturas de los dibujos de las cartas.

- **Métodos públicos:**
  - **AssociationScreen(FeedYourBrain fyb, entero type).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **generateCards(entero difficulty).** Método que genera las cartas en base a la dificultad pasada por parámetro.
  - **checkCard(Card card).** Método que es llamado cada vez que seleccionamos una carta, este método es el encargado de comprobar si hemos realizado una pareja.
  - **start().** Método sobrescrito de la clase GameScreen, este método es llamado cuando empezamos el juego.
  - **pause().** Método sobrescrito de la clase GameScreen, este método es llamado cuando pausamos el juego.
  - **resume().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reanudamos el juego desde una pausa.
  - **restart().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reiniciamos el juego.
  - **exit().** Método sobrescrito de la clase GameScreen, este método es llamado cuando salimos el juego.

- **Métodos privados:**
  - **loadCustomView().** Método que se encarga de cargar las vistas personalizadas del juego.
  - **setDifficulty().** Método para asignar la dificultad, la dificultad es asignada según en el modo de juego que nos encontremos así como el numero de aciertos que llevemos realizado.

9.1.23. SequenceScreen

La clase SequenceScreen extiende de la clase GameScreen y es la encargada de representar el juego de secuencias.

- **Atributos:**
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.

- **ArrayList<Actor>actors.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Actors, estos actores representar los objetos del juego.
- **ArrayList<CharacterCalc>charactersCalc.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase CharacterCalc.
- **Sequence sequence.** Atributo privado de la clase Sequence que almacena el sistema de generación de secuencias.
- **entero nHit.** Atributo privado de tipo entero que almacena el numero de aciertos.
- **entero streak.** Atributo privado de tipo entero que almacena la racha.

- **Métodos públicos:**

  - **Sequencecreen(FeedYourBrain fyb, entero type).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **start().** Método sobrescrito de la clase GameScreen, este método es llamado cuando empezamos el juego.
  - **pause().** Método sobrescrito de la clase GameScreen, este método es llamado cuando pausamos el juego.
  - **resume().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reanudamos el juego desde una pausa.
  - **restart().** Método sobrescrito de la clase GameScreen, este método es llamado cuando reiniciamos el juego.
  - **exit().** Método sobrescrito de la clase GameScreen, este método es llamado cuando salimos el juego.

- **Métodos privados:**

  - **loadCustomView().** Método que se encarga de cargar las vistas personalizadas del juego.
  - **setDifficulty().** Método para asignar la dificultad, la dificultad es asignada según en el modo de juego que nos encontremos así como el numero de aciertos que llevemos realizado.

9.1.24. Sequence

La clase Sequence es la clase encargada de generar las formulas para el juego de secuencias.

- **Atributos:**

  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.

- **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
- **ArrayList<CharacterCalc>charactersSequence.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase CharacterCalc.
- **ArrayList<CharacterCalc>charactersInput.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase CharacterCalc.
- **cadena formulate.** Atributo privado de la clase cadena que almacena la secuencia en formato String.
- **entero iResult.** Atributo privado de tipo entero que almacena el resultado de la secuencia en formato entero.
- **cadena sResult.** Atributo privado de la clase cadena que almacena el resultado de la secuencia en formato String.
- **entero difficulty.** Atributo privado de tipo entero que almacena la dificultad.

- **Métodos públicos:**

  - **Sequence(FeedYourBrain feedyourbrain, Stage stage).** Constructor parametrizado de la clase.
  - **generateSequence().** Método para generar una secuencia.
  - **entero checkResult().** Método para comprobar el resultado. Si es 0 aun no tenemos resultado, si es 1 es correcto y si es 2 es incorrecto.

9.1.25. AbstractScreen

La clase AbstractScreen. utiliza la interfaz de la clase Screen. Es la clase encargada de dibujar un fondo con una serie de animaciones para distintas pantallas de la aplicación.

- **Atributos:**

  - **FeedYourBrain feedYourBrain.** Atributo de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
  - **World world**. Atributo privado de la clase World que almacena el mundo virtual donde mostraremos nuestras texturas.
  - **Box2DDebugRenderer debugRenderer**. Atributo privado de la clase Box2DDebugRenderer que almacena el mundo virtual donde mostraremos nuestras texturas para realizar pruebas.
  - **OrthographicCamera cam**. Atributo privado de la clase OrthographicCamera que almacena la cámara desde donde veremos el mundo virtual.
  - **boolean debugMode**. Atributo privado de tipo boolean que almacena si estamos modo prueba.

- **Métodos públicos:**

- **AbstractScreen(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
- **createBugs().** Método para crear las animaciones.
- **dispose().** Método de la interfaz Screen que es llamado al liberar los recursos.
- **hide().** Método de la interfaz Screen que es llamado cuando esta pantalla no es la actual.
- **pause().** Método de la interfaz Screen que es llamado cuando la aplicación entra en pausa.
- **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
- **resize(entero arg0, entero arg1).** Método de la interfaz Screen que se encarga de la redimensión de la pantalla, recibe por parámetro el ancho y el alto al que queremos redimensionar.
- **resume()..** Método de la interfaz Screen que es llamado cuando la aplicación se reanuda desde un estado de pausa.
- **show().** Método de la interfaz Screen que es llamado cuando esta pantalla pasa a ser la actual.

- **Métodos privados:**
  - **loadInputProcessor().** Método para asignar los procesos de entrada.

9.1.26. CreateWorld

La clase CreateWorld es la clase encargada de crear los mundos con leyes físicas para los distintos juegos.

- **Métodos públicos:**
  - **World createWorld().** Método estático para crear un mundo.

9.1.27. Log

La clase Log es la clase encargada de gestionar los mensajes del sistema para realizar pruebas.

- **Atributos:**
  - **cadena tag.** Atributo privado de la clase cadena que almacena el identificador para identificar los mensajes del sistema.
- **Métodos públicos:**
  - **log(cadena content).** Método para imprimir como mensaje de sistema.

9.1.28. EndScreen

La clase EndScreen extiende de la clase AbstractScreen y es la encargada de representar la ultima pantalla del modo jugar.

- **Atributos:**
  - **entero globalScore.** Atributo privado de tipo entero que almacena la puntuación global obtenida en la partida.
  - **YourScore yourScore.** Atributo privado de la clase YourScore que almacena el actor que representa la textura del titulo de nuestra puntuación.
  - **MaxScore maxScore.** Atributo privado de la clase MaxScore que almacena el actor que reprensenta la textura del titulo de la máxima puntuación
  - **Button buttonRestart.** Atributo privado de la clase Button que almacena el boton para reiniciar la partida.
  - **Button buttonExit.** Atributo privado de de la clase Button que almacena el boton para salir del modo de juego.
  - **Title title.** Atributo privado de la clase Title que almacena el actor que representa la textura del titulo.
  - **ButtonClickListener buttonClickListener.** Atributo privado de la clase ButtonClickListener que almacena un Listener para los eventos de los botones.
  - **Score scoreLeft.** Atributo privado de la clase Score que almacena el actor que representa la textura de la cantidad de la maxima puntuación.
  - **Score scoreRight.** Atributo privado de la clase Score que almacena el actor que representa la textura de la cantidad de nuestra puntuación.
  - **Stage stage.** Atributo privado de la clase Stage que almacena el escenario virtual donde pintaremos nuestras texturas.
- **Métodos públicos:**
  - **EndScreen(FeedYourBrain feedyourbrain, entero globalScore).** Constructor parametrizado de la clase.
  - **render(flotante arg0).** Método de la interfaz Screen que es llamado cada vez que la aplicación tiene que renderizarse. El argumento representa el tiempo que tardo en llamarse desde la ultima vez.
  - **exit().** Método sobrescrito de la clase GameScreen, este método es llamado cuando salimos el juego.
  - **boolean keyDown(entero keycode).** Método de la interfaz InputProcessor que es llamado cuando una tecla es pulsada.
  - **boolean keyUp(entero keycode) .** Método de la interfaz InputProcessor que es llamado cuando una tecla es liberada.
  - **boolean keyTyped(char character).** Método de la interfaz InputProcessor que es llamado cuando una tecla es presionada.
  - **boolean touchDown(entero screenX, entero screenY, entero pointer, entero button).** Método de la interfaz InputProcessor que es llamado cuando la pantalla es pulsada.

- **boolean touchUp(entero screenX, entero screenY, entero pointer, entero button) .** Método de la interfaz InputProcessor que es llamado cuando la pantalla es liberada.
- **boolean touchDragged(entero screenX, entero screenY, entero pointer).** Método de la interfaz InputProcessor que es llamado cuando arrastramos una pulsación en la pantalla.
- **boolean mouseMoved(entero screenX, entero screenY).** Método de la interfaz InputProcessor que es llamado cuando movemos un cursor con un ratón.
- **boolean scrolled(entero amount) .** Método de la interfaz InputProcessor que es llamado cuando realizamos un scroll.

- **Métodos privados:**
  - **loadInputProcessor().** Método para asignar los procesos de entrada.
  - **loadView().** Método que se encarga de cargas las vistas propias de la clase.

9.1.29. Button

La clase Button extiende de la clase Actor e implementa los botones en la aplicación.

- **Atributos:**
  - **BitmapFont font.** Atributo privado de la clase BitmapFont que almacena el controlador de fuente.
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **entero type.** Atributo privado de tipo entero que almacena el tipo de botón.
- **Métodos públicos:**
  - **Button(Texture texture, BitmapFont font, cadena content).** Constructor parametrizado de la clase.
  - **Button(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.30. Character

La clase Character. extiende de la clase Actor y es la encargada de dibujar caracteres sueltos.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.

  - **Boolean completed.** Atributo privado de la clase Boolean que representa si el carácter ha realizado una animación.

- **Métodos públicos:**

  - **Character(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **Boolean hasCompleted().** Método que nos devuelve si el carácter ha finalizado una animación.

9.1.31. Level

La clase Level extiende de la clase actor y representa el dibujo del nivel.

- **Atributos:**

  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **cadena textures[].** Array de atributos privados de la clase cadena que almacena el nombre de las distintas texturas que usa la clase.

- **Métodos públicos:**

  - **Level(FeedYourBrain feedYourBrain).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

- **Métodos privados:**

  - **configTexture().** Método que se encarga de asignar la textura en base a la dificultad almacenada en las preferencias.

9.1.32. Message

La clase Message extiende de la clase Actor y es la encargada de dibujar mensajes por pantalla.

- **Atributos:**

  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.

- **BitmapFont font.** Atributo privado de la clase BitmapFont que almacena el controlador de fuente.
- **cadena message.** Atributo privado de la clase cadena que almacena el mensaje que queremos reprensentar.
- **flotante time.** Atributo privado de tipo flotante que almacena el tiempo que tarda en desaparecer el mensaje en pantalla.

- **Métodos públicos:**

  - **Message(Texture texture, BitmapFont font).** Constructor parametrizado de la clase.
  - **setContent(cadena message, flotante time).** Método para asignar el contenido que queremos mostrar el mensaje y el tiempo de duración de este.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.33. Ok

La clase Ok extiende de la clase Actor y representa la imagen de un acierto dentro de un juego.

- **Atributos:**

  - **flotante time.** Atributo privado de tipo flotante que almacena el tiempo que tarda en desaparecer la imagen de error en pantalla.
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.

- **Métodos públicos:**

  - **Ok(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **act(flotante delta).** Método que se encarga de actualizar el tiempo de renderizado de la textura.
  - **reload().** Método que se encarga de reiniciar el tiempo de renderizado de la textura.

9.1.34. No

La clase No extiende de la clase Actor y representa la imagen de un error dentro de un juego.

- **Atributos:**

- **flotante time.** Atributo privado de tipo flotante que almacena el tiempo que tarda en desaparecer la imagen de error en pantalla.
- **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.

- **Métodos públicos:**
  - **No(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **act(flotante delta).** Método que se encarga de actualizar el tiempo de renderizado de la textura.
  - **reload().** Método que se encarga de reiniciar el tiempo de renderizado de la textura.

9.1.35. Pause

La clase Pause extiende de la clase Actor y representa la textura del elemento pausa dentro de los juegos.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **entero type.** Atributo privado de tipo entero que almacena el tipo de elemento.
- **Métodos públicos:**
  - **Pause(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.36. Shape

La clase Shape extiende de la clase Actor y representa las texturas que se renderizan dentro de las cartas en el juego asociación.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **Shape(Texture texture, Vector2 position).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.37. Sound

La clase Sound extiende de la clase Actor y es la encargada de renderizar el elemento sonido dentro de los juegos.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **FeedYourBrain feedYourBrain.** Atributo de la clase FeedYourBrain que almacena la clase general de la aplicación.
- **Métodos públicos:**
  - **Sound(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **configTexture().** Método que se encarga de asignar la textura en base a las preferencias.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.38. Streak

La clase Streak extiende de la clase Actor y es la encargada de renderizar los distintos elementos para las rachas.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **FeedYourBrain feedYourBrain.** Atributo de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **entero streak.** Atributo privado de tipo entero para almacenar el numero de la racha.
- **Métodos públicos:**
  - **Streak(FeedYourBrain feedyourbrain).** Constructor parametrizado de la clase.
  - **configTexture().** Método que se encarga de asignar la textura en base al numero de la racha.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **update(entero update).** Método para actualizar la racha.

9.1.39. Time

La clase Time extiende de la clase Actor y es la encargada de renderizar el tiempo en cada juego.

- **Atributos:**
  - **BitmapFont font.** Atributo privado de la clase BitmapFont que almacena el controlador de fuente.
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **cadena content.** Atributo privado de la clase cadena que almacena el tiempo a mostrar por pantalla.
- **Métodos públicos:**
  - **Time(Texture texture, BitmapFont font).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **setContent(flotante seconds).** Método para actualizar el tiempo que mostramos por pantalla.

9.1.40. Bug

La clase Bug extiende de la clase Actor y es la encargada del renderizado de las animaciones en las pantallas AbstractScreen.

- **Atributos:**
  - **Body body.** Atributo privado de la clase Body que almacena la representación del cuerpo físico.
  - **BodyDef bodyDef.** Atributo privado de la clase BodyDef que almacena la definición del cuerpo físico.
  - **CircleShape circle.** Atributo privado de la clase CircleShape que almacena la representación de un circulo.
  - **FixtureDef fixtureDef.** Atributo privado de la clase FixtureDef que almacena la definición del cuerpo del circulo.
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **Bug(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **setButtonPosition(Vector2 position).** Método para asignar nuestra posición en el eje X y en el eje Y.

9.1.41. Description

La clase Description extiende de la clase Actor y es la encargada de renderizar las descripciones de los juegos.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **Description(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.42. Score

La clase Score extiende de la clase Actor y es la encargada de representar la puntuación en los juegos.

- **Atributos:**
  - **Texture[] textureNumbers.** Array de atributos privados de la clase Texture que almacena las distintas texturas para representar la puntuación.
  - **FeedYourBrain feedYourBrain.** Atributo privado de la clase FeedYourBrain que almacena la clase general de la aplicación.
  - **cadena nameChilds.** Atributo privado de la clase cadena que almacena el nombre de la clase que lo ha instanciado.
  - **cadena score.** Atributo privado de la clase cadena que almacena el valor de la puntuacion en formato String.
  - **boolean side.** Atributo privado de tipo boolean que almacena el lado donde mostrar la puntuación. Si es true se mostraría a la derecha y si es false a la izquierda.
- **Métodos públicos:**
  - **Score(FeedYourBrain feedYourBrain, Texture[] textureNumbers, cadena nameChilds).** Constructor parametrizado de la clase.
  - **Score(FeedYourBrain feedYourBrain, Texture[] textureNumbers, entero score).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
- **Métodos privados:**
  - **cadena getMaxScore().** Método que devuelve la máxima puntuación.

9.1.43. MaxScore

La clase MaxScore extiende de la clase Actor y es la encargada de representar el titulo de máxima puntuación.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **MaxScore(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
- **Métodos privados:**
  - **loadPosition().** Método para asignar una posición especifica a este actor.

9.1.44. StartGameDescription

La clase StartGameDescription extiende de la clase Actor y es la encargada de representar la descripción de un juego.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **StartGameDescription(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.45. StartGameScoreMax

La clase StartGameScoreMax extiende de la clase Actor y es la encargada de representar el titulo de máxima puntuación.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.

- **Métodos públicos:**
  - **StartGameScoreMax(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.46. StartGameTitle

La clase StartGameScoreMax extiende de la clase Actor y es la encargada de representar el titulo en el modo jugar.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **StartGameTitle(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.47. Title

La clase Title extiende de la clase Actor y es la encargada de representar el titulo de cada juego.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **Title(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
- **Métodos privados:**
  - **loadPosition().** Método para asignar una posición especifica a este actor.

9.1.48. YourScore

La clase YourScore extiende de la clase Actor y es la encargada de representar el titulo de nuestra puntuación.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **YourScore(Texture texture).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
- **Métodos privados:**
  - **loadPosition().** Método para asignar una posición especifica a este actor.

9.1.49. Circle

La clase Circle extiende de la clase Actor y es la encargada del manejo de las bolas del juego visual.

- **Atributos:**
  - **Body body.** Atributo privado de la clase Body que almacena la representación del cuerpo físico.
  - **BodyDef bodyDef.** Atributo privado de la clase BodyDef que almacena la definición del cuerpo físico.
  - **CircleShape circle.** Atributo privado de la clase CircleShape que almacena la representación de un circulo.
  - **FixtureDef fixtureDef.** Atributo privado de la clase FixtureDef que almacena la definición del cuerpo del circulo.
  - **BitmapFont font.** Atributo privado de la clase BitmapFont que almacena el controlador de fuente.
  - **entero number.** Atributo privado de tipo entero que almacena el numero del circulo.
  - **cadena sNumber.** Atributo privado de la clase cadena que almacena el numero del circulo.
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**

- **(Texture texture, BitmapFont font, entero number).** Constructor parametrizado de la clase.
- **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
- **setButtonPosition(Vector2 position).** Método para asignar nuestra posición en el eje X y en el eje Y.

9.1.50. Card

La clase Card extiende de la clase Actor y es la encargada de renderizar las cartas en el juego memoria.

- **Atributos:**
  - **Texture textureHead.** Atributo privado de la clase Texture que almacena la textura del actor en el estado cara.
  - **Texture textureTail.** Atributo privado de la clase Texture que almacena la textura del actor en el estado cruz.
  - **entero type.** Atributo privado de tipo entero que almacena el tipo de carta.
  - **Boolean head.** Atributo privado de la clase Boolean que almacena si tenemos que pintar la cara o la cruz de una carta
  - **Boolean tumble.** Atributo privado de la clase Boolean que almacena si tenemos que voltear una carta o no.
  - **flotante TIME.** Atributo privado de tipo flotante que almacena el tiempo que tarda una carta en realizar la animación de voltearse.
  - **flotante time.** Atributo privado de tipo flotante que almacena el tiempo que lleva una carta volteandose.
- **Métodos públicos:**
  - **Card(Texture textureHead, Texture textureTail, entero type).** Constructor parametrizado de la clase.
  - **Card(Card card).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **tailToHead()**. Método para voltear una carta de abajo hacia arriba.
  - **updateTime(flotante delta).** Método para actualizar el tiempo de volteado de una carta.
  - **boolean isTumble().** Método que nos devuelve si una carta se tiene que voltear.
  - **setTumble(boolean tumble).** Método para asignar si tenemos que voltear una carta
  - **setHead(Boolean head).** Método para asignar si queremos que se muestre la cara de una carta.

9.1.51. CharacterCalc

La clase CharacterCalc extiende de la clase Actor y es la encargada del manejo de los caracteres de la calculadora en el juego matemáticas.

- **Atributos:**
  - **char character.** Atributo privado de tipo char que almacena el carácter interno que manejamos.
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
- **Métodos públicos:**
  - **CharacterCalc(Texture texture, Vector2 position, char character).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

9.1.52. Balance

La clase Balance extiende de la clase Actor y es la encargada del manejo de las cartas para el juego memoria.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **ArrayList<Weight>left.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Weight, estos actores representar los pesos de las balanzas izquierdas.
  - **ArrayList<Weight>left.** Atributo privado de la clase ArrayList que almacena una lista de atributos de la clase Weight, estos actores representar los pesos de las balanzas derechas.
- **Métodos públicos:**
  - **public Balance(Texture texture, entero type).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.
  - **addtoLeft(Weight weight).** Método para añadir un peso a las balanzas izquierdas.
  - **addtoLeft(Weight weight).** Método para añadir un peso a las balanzas derechas.

9.1.53. Weight

La clase Weight extiende de la clase Actor y es la encargada del manejo de las cartas para el juego pesos.

- **Atributos:**
  - **Texture texture.** Atributo privado de la clase Texture que almacena la textura del actor.
  - **entero amount.** Atributo privado de tipo entero para manejar la cantidad del peso.
  - **cadena letter.** Atributo privado de tipo cadena para manejar un identificador interno del peso.
- **Métodos públicos:**
  - **public Weight(Texture texture, entero amount, cadena letter).** Constructor parametrizado de la clase.
  - **draw(Batch batch, flotante alpha).** Método que se encarga de dibujar la textura.

Capítulo 10

Diseño del juego

En este capítulo definiremos el *gameplay* de la aplicación, abordando los distintos juegos, las mecánicas, los logros y retos. Si bien el aspecto visual de un juego es muy importante con el **gameplay** (la totalidad de las acciones que un jugador puede realizar durante la interacción con un videojuego) es con lo que se fideliza a un usuario.

10.1. Juego: Matemáticas

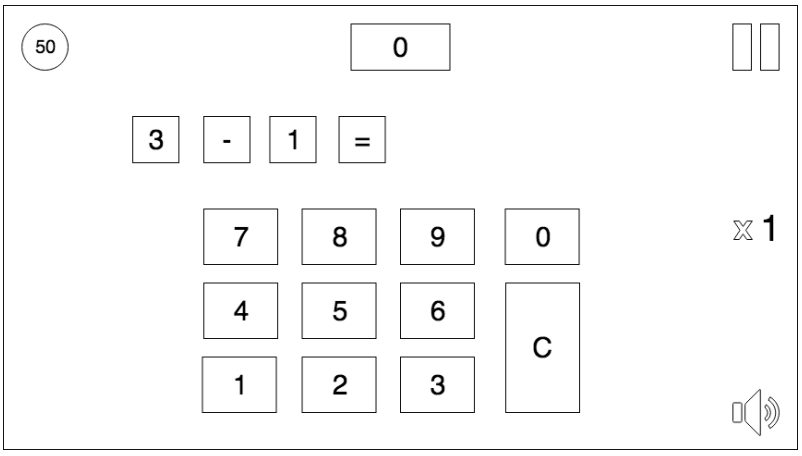


Figura 10.1: Matemáticas

10.1.1. Objetivo

El objetivo del juego es realizar correctamente las operaciones matemáticas presentadas en la pantalla. Para introducir el resultado se utilizará el teclado numérico dibujado en la pantalla.

Las operaciones tendrán los símbolos de suma (+), resta (-), multiplicación (x) y división (/).

10.1.2. ¿Qué implica cada nivel de dificultad?

Cada nivel de dificultad implica un aumento de la complejidad de las operaciones, todos los niveles tienen operaciones con suma, resta, multiplicación y división.

Nivel fácil:

- Suma de dos números, cada número estará dentro del rango 0-9.
- Resta de dos números, cada número estará dentro del rango 0-9.
- Multiplicación de dos números, cada número estará dentro del rango 0-9.
- División de dos números, el dividendo estará dentro del rango 1-81 y el divisor dentro del rango 1-9.

Nivel normal:

- Suma de dos números, cada número estará dentro del rango 0-19.
- Resta de dos números, cada número estará dentro del rango 0-19.
- Multiplicación de dos números, cada número estará dentro del rango 0-9, a esta multiplicación se le añadirá una suma o una resta de un número dentro del rango 0-9.
- División de dos números, el dividendo estará dentro del rango 1-81 y el divisor dentro del rango 1-9, a esta división se le añadirá una suma o una resta de un número dentro del rango 0-9.

Nivel difícil:

- Suma de dos números, cada número estará dentro del rango 0-29.
- Resta de dos números, cada número estará dentro del rango 0-29.
- Multiplicación de dos números, cada número estará dentro del rango 0-10, a esta multiplicación se le añadirá una suma o una resta de un número dentro del rango 0-19.
- División de dos números, el dividendo estará dentro del rango 1-81 y el divisor dentro del rango 1-9, a esta división se le añadirá una suma o una resta de un número dentro del rango 0-19.

10.1.3. Sistema de puntuación:

¿Cuánto suma cada acierto en cada modo de dificultad?

- En nivel fácil un acierto suma 10 puntos.

- En nivel normal un acierto suma 100 puntos.
- En nivel difícil un acierto suma 1000 puntos.

¿Qué implica un fallo?

Reinicio de rachas.

10.1.4. Rachas:

- Racha de 5 aciertos, la puntuación se multiplicará por el doble de su valor, en nivel fácil serán 20 puntos, en nivel normal 200 puntos y en nivel difícil 2000 puntos.
- Racha de 10 aciertos, la puntuación se multiplicará por el triple de su valor, en nivel fácil serán 30 puntos, en nivel normal 300 puntos y en nivel difícil 3000 puntos.

10.1.5. Logros:

- Conseguir 1000 puntos en Matemáticas fácil.
- Conseguir 10000 puntos en Matemáticas normal.
- Conseguir 100000 puntos en Matemáticas difícil.

10.2. Juego: Memoria

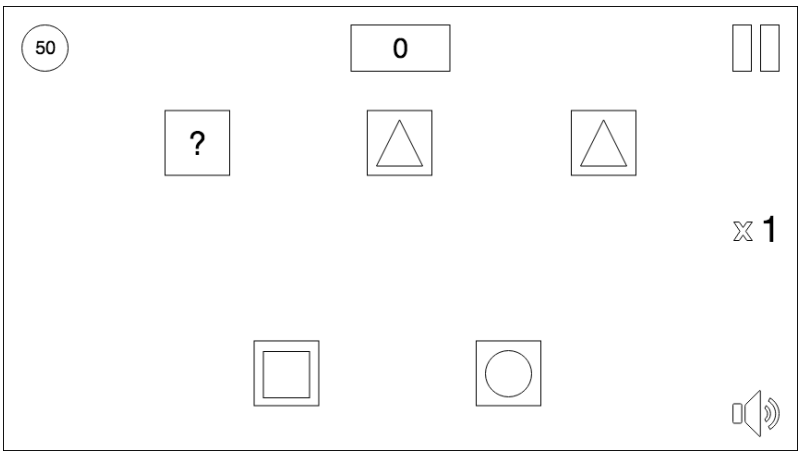


Figura 10.2: Memoria

10.2.1. Objetivo

El objetivo del juego es seleccionar las cartas correctas, para ello debemos memorizar las cartas que salen en la parte de arriba, algunas de estas cartas se voltearán y aparecerán unas nuevas en la parte de abajo. Una vez que aparezcan estas cartas de abajo deberemos de seleccionar las que sean iguales que las que se voltearon arriba.

La fila de arriba se genera carta a carta, cada vez que generamos una carta se ejecuta un código donde se extrae uno de los tipos posibles de carta.

El número de tipos de cartas es 6, cada carta es distinta en forma y en color

10.2.2. ¿Qué implica cada nivel de dificultad?

Cada nivel de dificultad implica un aumento tanto en el número de cartas que aparecen arriba como en el número de cartas que se voltearán.

Nivel fácil:

- El número de cartas en la fila de arriba es 3.
- El número de cartas que se voltean es 1.
- El número de cartas que tendremos de muestra sera 2.

Nivel normal:

- El número de cartas en la fila de arriba es 4.
- El número de cartas que se voltean es 2.
- El número de cartas que tendremos de muestra sera 2 o 3.

Nivel difícil:

- El número de cartas en la fila de arriba es 5
- El número de cartas que se voltean es 3.
- El número de cartas que tendremos de muestra sera 3 o 4.

10.2.3. Sistema de puntuación:

¿Cuánto suma cada acierto en cada modo de dificultad?

- En nivel fácil un acierto suma 20 puntos.
- En nivel normal un acierto suma 200 puntos.
- En nivel difícil un acierto suma 2000 puntos.

¿Qué implica un fallo?

Reinicio de rachas.

10.2.4. Rachas:

- Racha de 5 aciertos, la puntuación se multiplicará por el doble de su valor, en nivel fácil serán 40 puntos, en nivel normal 400 puntos y en nivel difícil 4000 puntos.
- Racha de 10 aciertos, la puntuación se multiplicará por el triple de su valor, en nivel fácil serán 60 puntos, en nivel normal 600 puntos y en nivel difícil 6000 puntos.

10.2.5. Logros:

- Conseguir 1000 puntos en Memoria fácil.
- Conseguir 10000 puntos en Memoria normal.
- Conseguir 100000 puntos en Memoria difícil.

10.3. Juego: Visual

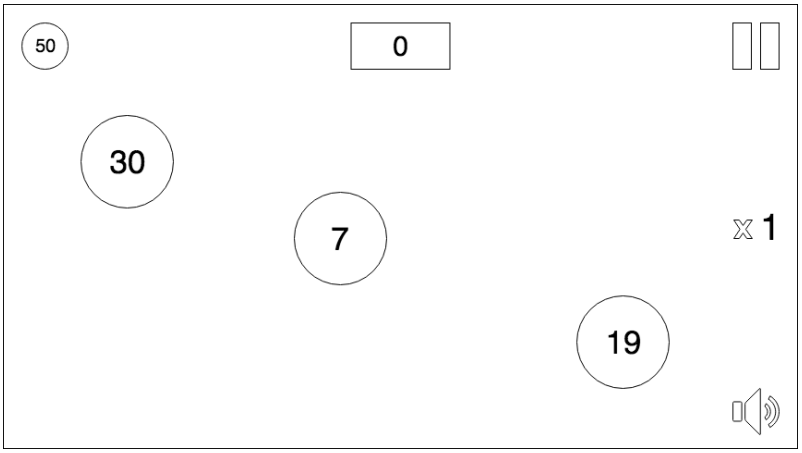


Figura 10.3: Visual

10.3.1. Objetivo

El objetivo del juego es seleccionar las bolas. Para que esta selección sea correcta hay que hacerla de menor a mayor basándonos en el número que contiene la bola.

10.3.2. ¿Qué implica cada nivel de dificultad?

Cada nivel de dificultad implica un mayor número de bolas, también implica números mayores en valor absoluto dentro de las bolas.

Nivel fácil:

- El número de bolas es 3.

- El número que contiene esta dentro del rango [1, 40].

Nivel normal:

- El número de bolas es 4.
- El número que contiene esta dentro del rango [-50, 50].

Nivel difícil:

- El número de bolas es 4.
- El número que contiene esta dentro del rango [-99, 99].

10.3.3. Sistema de puntuación:

¿Cuánto suma cada acierto en cada modo de dificultad?

- En nivel fácil un acierto suma 10 puntos.
- En nivel normal un acierto suma 100 puntos.
- En nivel difícil un acierto suma 1000 puntos.

¿Qué implica un fallo?

Reinicio de rachas.

10.3.4. Rachas:

- Racha de 5 aciertos, la puntuación se multiplicará por el doble de su valor, en nivel fácil serán 20 puntos, en nivel normal 200 puntos y en nivel difícil 2000 puntos.
- Racha de 10 aciertos, la puntuación se multiplicará por el triple de su valor, en nivel fácil serán 30 puntos, en nivel normal 300 puntos y en nivel difícil 3000 puntos.

10.3.5. Logros:

- Conseguir 1000 puntos en Visual fácil.
- Conseguir 10000 puntos en Visual normal.
- Conseguir 100000 puntos en Visual difícil.

10.4. Juego: Asociación

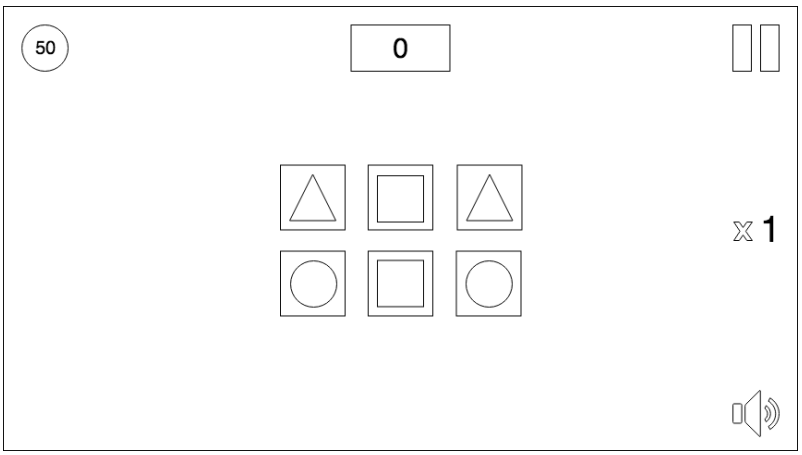


Figura 10.4: Asociación

10.4.1. Objetivo

El objetivo del juego es encontrar las parejas, para ello se presentará una matriz de imágenes donde algunas de ellas estarán duplicadas.

10.4.2. ¿Qué implica cada nivel de dificultad?

Cada nivel de dificultad implica un mayor número de imágenes y un mayor número de parejas.

Nivel fácil:

- El número de parejas a buscar es 1.
- El número de imágenes de la matriz es 6.

Nivel normal:

- El número de parejas a buscar es 2.
- El número de imágenes de la matriz es 6.

Nivel difícil:

- El número de parejas a buscar es 4
- El número de imágenes de la matriz es 15.

10.4.3. Sistema de puntuación:

¿Cuánto suma cada acierto en cada modo de dificultad?

- En nivel fácil un acierto suma 10 puntos.
- En nivel normal un acierto suma 100 puntos.
- En nivel difícil un acierto suma 1000 puntos.

¿Qué implica un fallo?

Reinicio de rachas.

10.4.4. Rachas:

- Racha de 10 aciertos, la puntuación se multiplicará por el triple de su valor, en nivel fácil serán 30 puntos, en nivel normal 300 puntos y en nivel difícil 3000 puntos.
- Racha de 5 aciertos, la puntuación se multiplicará por el doble de su valor, en nivel fácil serán 20 puntos, en nivel normal 200 puntos y en nivel difícil 2000 puntos.

10.4.5. Logros:

- Conseguir 1000 puntos en Asociación fácil.
- Conseguir 10000 puntos en Asociación normal.
- Conseguir 100000 puntos en Asociación difícil.

10.5. Juego: Secuencias

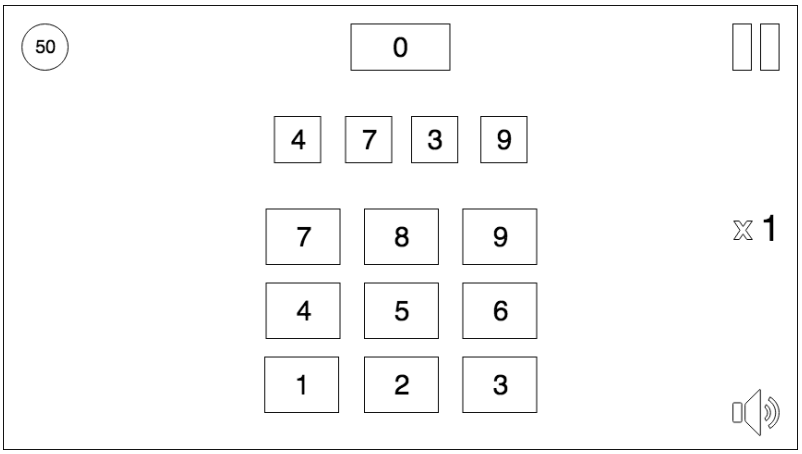


Figura 10.5: Secuencias

10.5.1. Objetivo

El objetivo del juego es memorizar la secuencia de objetos que se muestra por pantalla para después replicarla, la secuencia desaparece pasado un tiempo.

Para replicar la secuencia se mostrará un panel de selección de objetos.

10.5.2. ¿Qué implica cada nivel de dificultad?

Cada nivel de dificultad implica un aumento en el número de objetos de la secuencia.

Nivel fácil:

- Cada secuencia tendrá un número de objetos dentro del rango 2-3.

Nivel normal:

- Cada secuencia tendrá un número de objetos dentro del rango 3-5.

Nivel difícil:

- Cada secuencia tendrá un número de objetos dentro del rango 5-7.

10.5.3. Sistema de puntuación:

¿Cuánto suma cada acierto en cada modo de dificultad?

- En nivel fácil un acierto suma 10 puntos.
- En nivel normal un acierto suma 100 puntos.
- En nivel difícil un acierto suma 1000 puntos.

¿Qué implica un fallo?

Reinicio de rachas.

10.5.4. Rachas:

- Racha de 5 aciertos, la puntuación se multiplicará por el doble de su valor, en nivel fácil serán 20 puntos, en nivel normal 200 puntos y en nivel difícil 2000 puntos.
- Racha de 10 aciertos, la puntuación se multiplicará por el triple de su valor, en nivel fácil serán 30 puntos, en nivel normal 300 puntos y en nivel difícil 3000 puntos.

10.5.5. Logros:

- Conseguir 1000 puntos en Secuencias fácil.
- Conseguir 10000 puntos en Secuencias normal.
- Conseguir 100000 puntos en Secuencias difícil.

10.6. Juego: Pesos

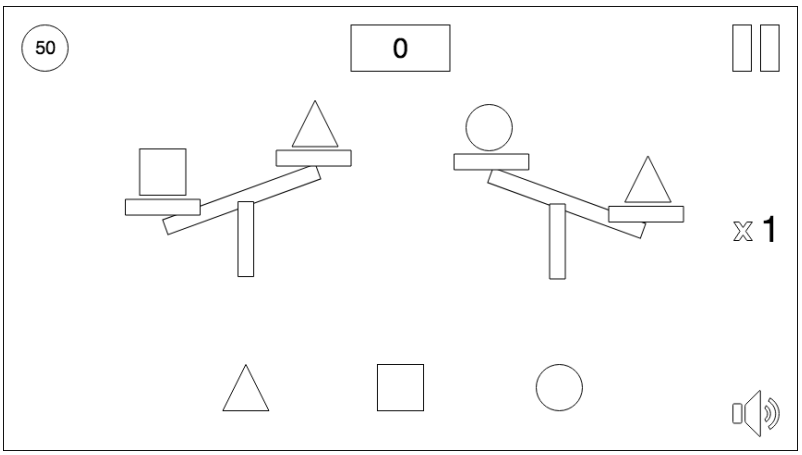


Figura 10.6: Pesos

10.6.1. Objetivo

El objetivo del juego es seleccionar el peso correcto, para ello debemos comparar todos los pesos que se encuentran en las distintas balanzas que se mostrarán por pantalla.

Conforme vaya avanzando el juego se irá cambiando entre seleccionar el peso más pesado o el peso menos pesado.

10.6.2. ¿Qué implica cada nivel de dificultad?

Cada nivel de dificultad implica un aumento en el número de balanzas y un aumento en el número de pesos por balanza.

Nivel fácil:

- Juego con dos o tres balanzas, cada balanza tendrá un número de pesos dentro del rango 2-3.

Nivel normal:

- Juego con tres balanzas, cada balanza tendrá un número de pesos dentro del rango 2-4.

Nivel difícil:

- Juego con tres balanzas, cada balanza tendrá un número de pesos dentro del rango 2-4.

10.6.3. Sistema de puntuación:

¿Cuánto suma cada acierto en cada modo de dificultad?

- En nivel fácil un acierto suma 10 puntos.
- En nivel normal un acierto suma 100 puntos.
- En nivel difícil un acierto suma 1000 puntos.

¿Qué implica un fallo?

Reinicio de rachas.

10.6.4. Rachas:

- Racha de 5 aciertos, la puntuación se multiplicará por el doble de su valor, en nivel fácil serán 20 puntos, en nivel normal 200 puntos y en nivel difícil 2000 puntos.
- Racha de 10 aciertos, la puntuación se multiplicará por el triple de su valor, en nivel fácil serán 30 puntos, en nivel normal 300 puntos y en nivel difícil 3000 puntos.

10.6.5. Logros:

- Conseguir 1000 puntos en Pesos fácil.
- Conseguir 10000 puntos en Pesos normal.
- Conseguir 100000 puntos en Pesos difícil.

Capítulo 11

Diseño de la interfaz gráfica

Dado que la aplicación dispone de una interfaz gráfica de usuario, en este capítulo expondremos los bocetos de las distintas pantallas que contendrá la aplicación, así como donde irán colocados los elementos específicos de cada pantalla. Los elementos de cada pantalla serán imágenes, textos y botones.

11.1. Diseño de la pantalla de carga

El diseño de la *pantalla de carga* es sencillo. Como muestra la Fig. 11.1 contiene el nombre de la aplicación y una barra que representa la carga de los recursos. La barra de carga se ira rellenando conforme se vayan cargando los recursos.

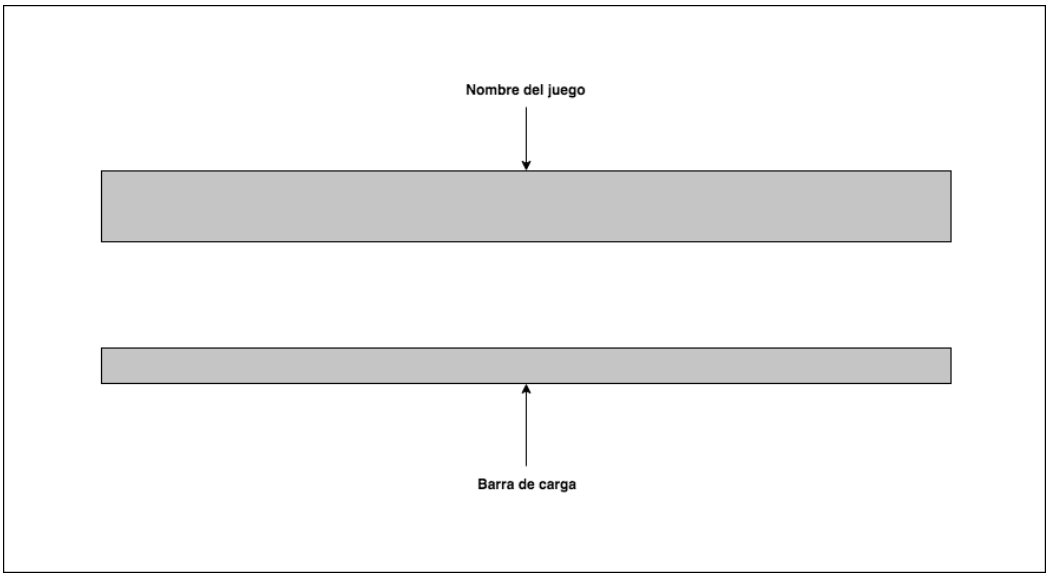


Figura 11.1: Diseño de la pantalla de carga

11.2. Diseño de la pantalla principal

La *pantalla principal*, como muestra la Fig. 11.2, contiene los botones principales de la aplicación. Estos botones darán acceso a las opciones, Jugar, Juegos, Puntuación y Ayuda. También contiene un botón para salir de la aplicación y otro para activar o desactivar el sonido.

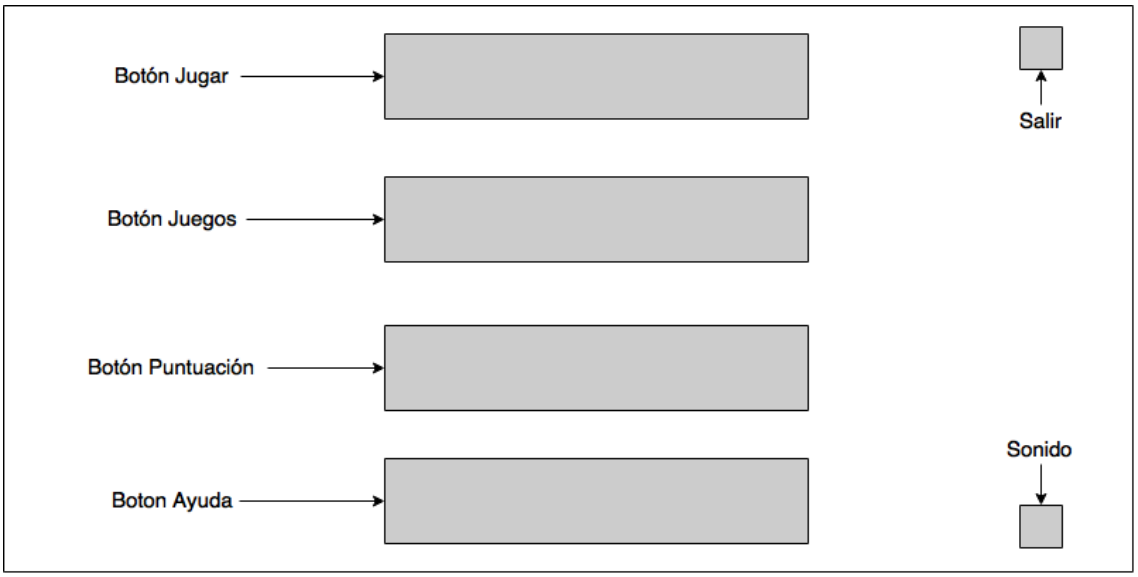


Figura 11.2: Diseño de la pantalla principal

11.3. Diseño de la pantalla principal de jugar o de juegos

La *pantalla principal de jugar o de juegos* es la pantalla a la que accederemos antes de empezar el modo Jugar o cualquier juego de manera individual. Como muestra la Fig. 11.3 está formada por un título que informa en qué modo nos encontramos, una descripción del modo que vamos a jugar, la máxima puntuación obtenida en este modo, y dos botones, uno para empezar el modo y otro para volver atrás.

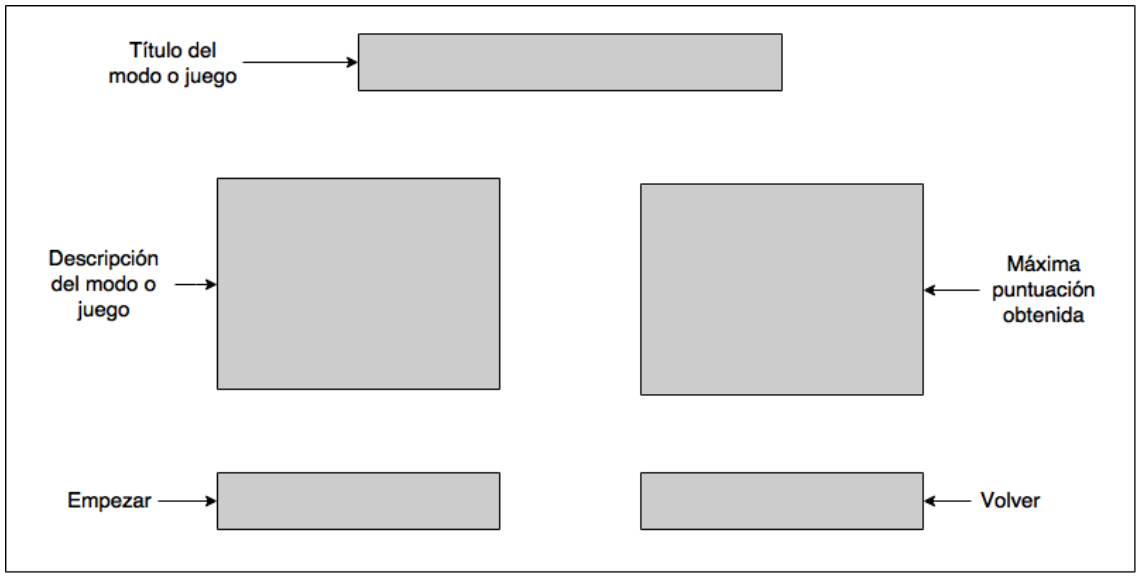


Figura 11.3: Diseño de la pantalla principal de jugar o de juegos

11.4. Diseño de la pantalla de juego

La *pantalla de juego* está compuesta por distintos elementos. Como muestra la Fig. 11.4 la barra superior de esta pantalla está compuesta por el tiempo, la puntuación y el botón de pausa, a la derecha tendremos un indicador de rachas, en la esquina inferior derecha un botón para activar o desactivar el sonido y finalmente en el centro tendremos todas las vistas para el juego.

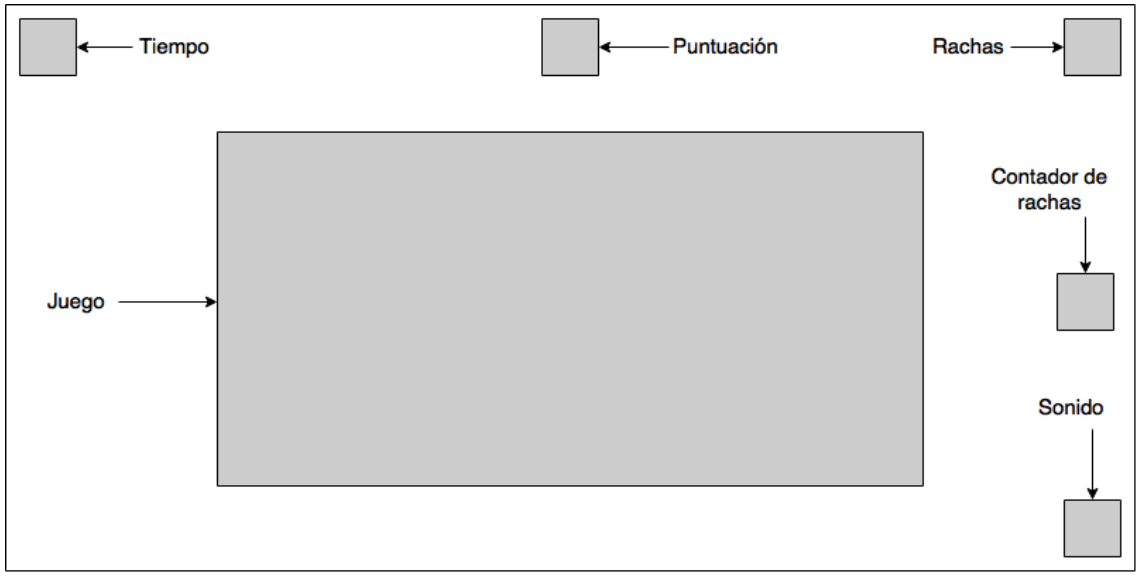


Figura 11.4: Diseño de la pantalla de juego

11.5. Diseño de la pantalla selección de nivel

El diseño de la *pantalla de selección de nivel* es sencillo. Como muestra la Fig. 11.5 está principalmente compuesta por tres botones que representan el acceso a cada nivel de dificultad: fácil, normal y difícil. La barra superior esta compuesta por un botón para volver atrás.

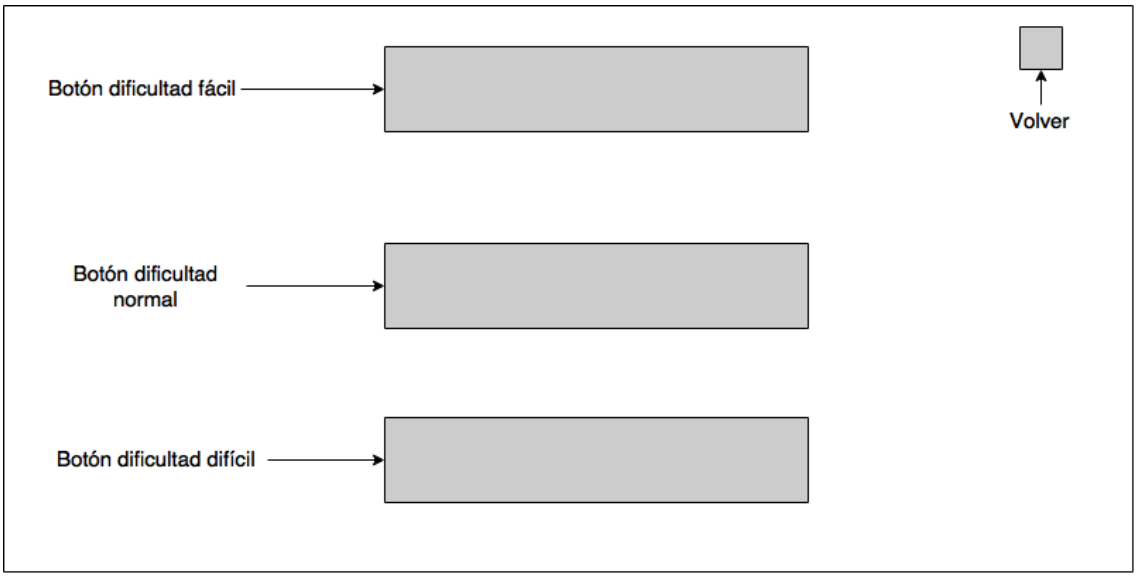


Figura 11.5: Diseño de la pantalla selección de nivel

11.6. Diseño de la pantalla de selección de juego

El diseño de la *pantalla de selección de juego*, como muestra la Fig. 11.6, está compuesta principalmente por 6 botones que representar los accesos a los seis distintos juegos que tiene la aplicación que son matemáticas, visual, asociación, memoria, secuencias y pesos. La parte inferior de esta pantalla tiene un indicador de dificultad seleccionado. La barra superior está compuesta por un botón para volver atrás.

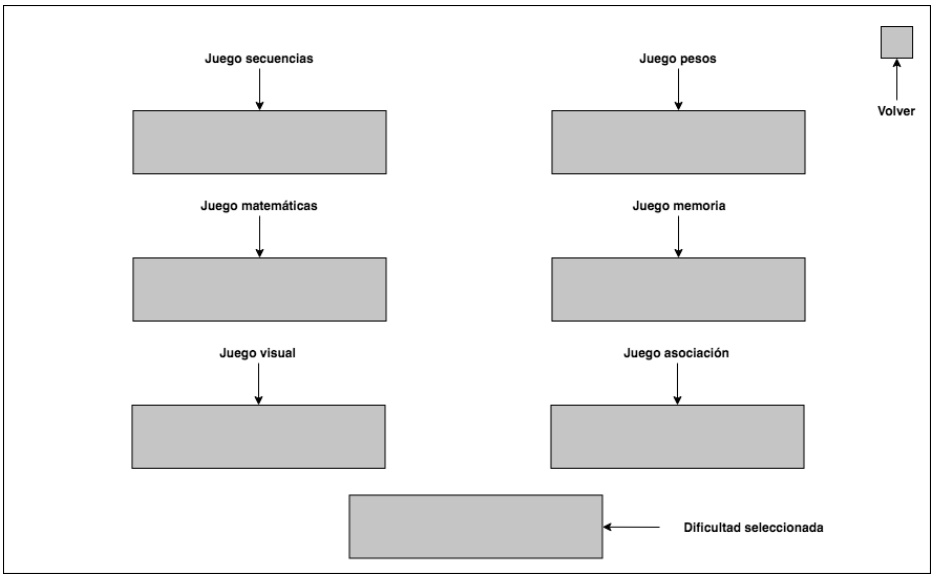


Figura 11.6: Diseño de la pantalla de selección de juego

11.7. Diseño de la pantalla final de jugar o de juegos

La *pantalla final de jugar o juegos*, como muestra la Fig. 11.7, es la pantalla a la que accederemos cuando finalicemos el modo Jugar o cualquier juego de manera individual. Está formada por un titulo que informa en que modo nos encontramos, la puntuación obtenida, la máxima puntuación obtenida en este modo y dos botones, uno para reiniciar el modo y otro para volver atrás.

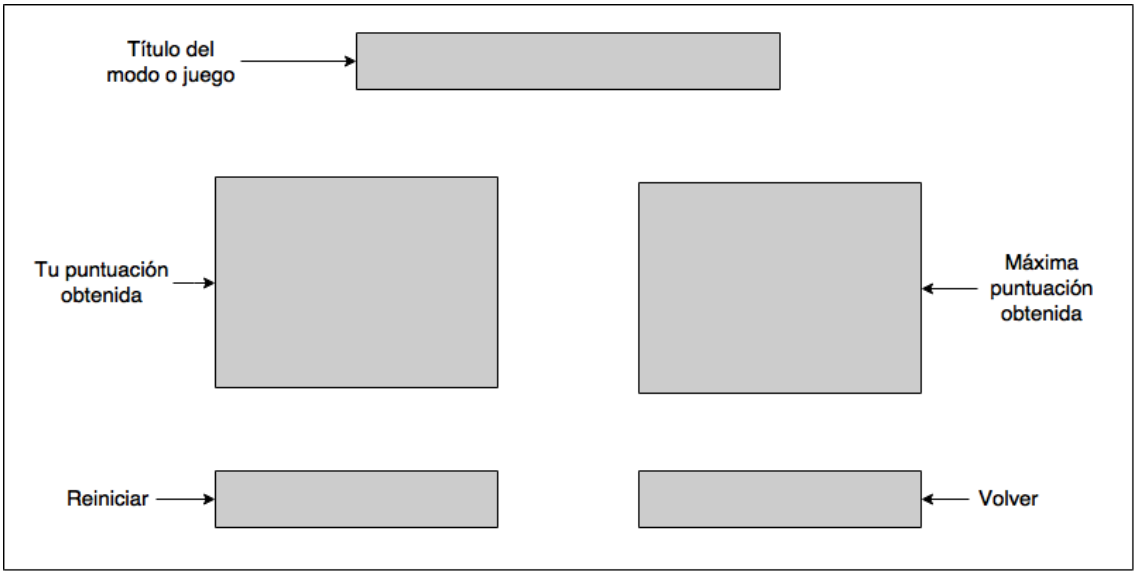


Figura 11.7: Diseño de la pantalla final de jugar o de juegos

11.8. Diseño de la pantalla de selección de puntuación

La *pantalla de selección de puntuación*, como muestra la Fig. 11.8, está compuesta por tres botones, estos botones nos permiten consultar las puntuaciones así como los logros que hemos obtenido. La barra superior está compuesta por un botón para volver atrás.

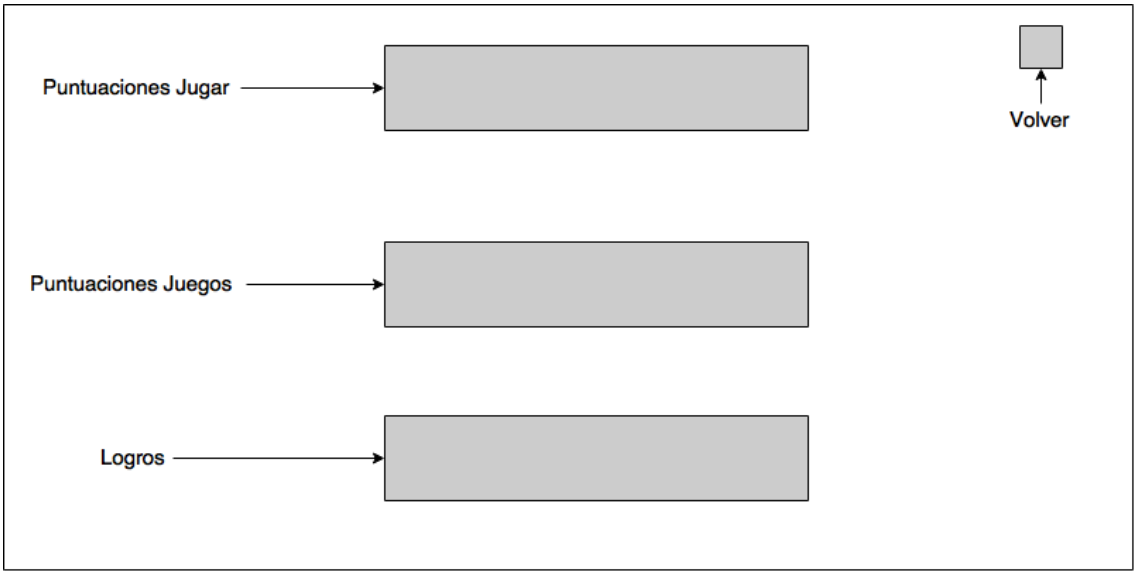


Figura 11.8: Diseño de la pantalla de selección de puntuación

11.9. Diseño de la pantalla de ayuda

El diseño de la *pantalla de ayuda*, como muestra la Fig. 11.9, está compuesta por una descripción y un botón para volver atrás en la barra superior.

Figura 11.9: Diseño de la pantalla de ayuda

Capítulo 12

Pruebas

En este apartado se detallan las pruebas realizadas al proyecto. Las pruebas se realizan con el objetivo de mejorar la calidad y fiabilidad del software, reparar posibles errores no encontrados hasta entonces y comprobar su robustez ante fallos. Un buen caso de prueba es aquel que tiene una alta probabilidad de descubrir un error no encontrado hasta entonces.

El objetivo de la etapa de pruebas, expresado de forma sencilla, es encontrar el mayor número de errores posibles con una cantidad razonable de esfuerzo, aplicado sobre un lapso de tiempo realista [17].

La ejecución de las pruebas se caracteriza por la imposibilidad de realizar pruebas exhaustivas. Esto es, probar todas las posibles combinaciones de distintos caminos por los que puede ejecutarse el software. Ante esto, se definen dos enfoques de pruebas para evaluar las distintas características de un software:

- **Pruebas de caja blanca o enfoque estructural.** Estas pruebas se centran en la estructura interna del programa para elegir los casos de prueba. La prueba exhaustiva consistiría en probar todos los posibles caminos de ejecución del programa, algo que, como se ha indicado antes, conllevaría un enorme número de caminos lógicos y, en muchos casos, la imposibilidad de su realización práctica. Por ello se eligen los más importantes o aquellos que sean más susceptibles a fallar.
- **Pruebas de caja negra o enfoque funcional.** Estas pruebas estudian la especificación de las funciones. La prueba exhaustiva consistiría en comprobar las salidas de todas y cada una de las funciones para distintos conjuntos de datos de entrada. En cuanto a las entradas, es importante tener en cuenta cuestiones como el tipo de dato, el volumen de éstos y las posibles combinaciones de entradas proporcionadas a la función. También es importante en este tipo de pruebas estudiar las funciones interfaz, esto es, la comunicación entre distintas funciones y módulos.

12.1. Pruebas de caja blanca

Las pruebas de caja blanca son pruebas estructurales. Conociendo el código y siguiendo su estructura lógica, se pueden diseñar pruebas destinadas a comprobar que el código hace correctamente lo que el diseño de bajo nivel indica y otras que demuestren que no se comporta adecuadamente ante determinadas situaciones. Ejemplos típicos de ellos son las pruebas unitarias. Se centran en lo que hay codificado o diseñado a bajo nivel por lo que no es necesario conocer la especificación de requisitos, que por otra parte será difícil de relacionar con partes diseñadas a muy bajo nivel.

Los elementos de mayor interés, y también los más susceptibles de presentar fallos, son los siguientes:

- Bucles simples o anidados. Se estudia el número de veces que se ejecuta el bloque de código que se encuentra dentro del bucle. Pueden ser especialmente problemáticos los casos en los que el bucle se pase por alto (0 ejecuciones), o se recorra un número mayor de veces del previsto.
- Sentencias condicionales. Puede ocurrir que el flujo de ejecución no supere ninguna de las condiciones propuestas y no ejecute ninguno de los casos, que entre en más de una condición, etcétera. Es importante estudiar el solapamiento de casos.

Mediante las pruebas de caja blanca en nuestro proyecto hemos obtenido que se garantizan los siguientes casos:

- Ejecución, por lo menos una vez, de los caminos independientes de cada módulo.
- Comprobación de todas las decisiones lógicas en sus vertientes verdadera y falsa.
- Comprobación de todos los limites operacionales en bucles.
- Validación de los datos de las estructuras internas.

12.2. Pruebas de caja negra

Son pruebas funcionales. Se parte de los requisitos funcionales, a muy alto nivel, para diseñar pruebas que se aplican sobre el sistema sin necesidad de conocer como está construido por dentro(Caja negra). Las pruebas se aplican sobre el sistema empleando un determinado conjunto de datos de entrada y observando las salidas que se producen para determinar si la función se está desempeñando correctamente por el sistema bajo prueba. Las herramientas básicas son observar la funcionalidad y contrastar con la especificación.

Las pruebas de caja negra se basan en estudiar los tipos de entradas de las funciones y comprobar que las salidas proporcionadas son las esperadas. Este proceso se debe llevar a cabo tanto durante la implementación del software como a la finalización del mismo. En la fase de implementación, para comprobar que las entradas de una función que sean las salidas de otra función, son válidas y están controladas. Y a la finalización del software para

probar nuevos conjuntos de entradas que se usen para valorar casos de error, y permitir al software recuperarse ante situaciones inesperadas y manejar los errores.

Ejemplo típicos de pruebas de caja negra son la comprobación de valores límite, pruebas de integridad de la base de datos, pruebas de situaciones de excepción, o pruebas de rendimiento del sistema. Presentan una limitación en cuanto a que es prácticamente imposible reproducir todo el espectro por la innumerable cantidad de combinaciones de entrada posibles, agravada por el desconocimiento de la lógica interna.

A continuación se expondrán las pruebas de caja negra aplicadas a la aplicación. Principalmente hay que comprobar la carga y acceso de todos los recursos del sistema así como los procesos de lógica de cada juego.

- **Acceso a recursos en distintos idiomas.** La aplicación esta continuamente accediendo a los recursos (imágenes y sonidos) precargados en el sistema. Un caso a comprobar es que el acceso al recurso exista siempre y no produzca fallo. Si solicitamos un recurso en un idioma predeterminado que no tengamos en nuestra aplicación este devolverá uno configurado por defecto.
- **Uso de memoria.** La aplicación trata de utilizar la mínima cantidad de memoria posible pero puede darse el caso en el que el dispositivo se quede sin memoria por lo que la aplicación a su vez no tendría memoria para seguir trabajando. En esta situación, la aplicación captura la excepción que lanza el sistema operativo y libera la memoria consumida por la aplicación. Si aún así el sistema operativo no le proporciona más memoria a la aplicación, ésta termina su ejecución de forma controlada y avisa al usuario de que no dispone de memoria suficiente para poder ejecutarse, y le recomienda que cierre programas que no esté usando para liberar memoria.

Además de estas pruebas genéricas, se han realizado otras pruebas más específicas que se detallan a continuación.

12.2.1. Caso de prueba 1 - Liberar recursos y salir de la aplicación

Esta prueba consistirá en salir de la aplicación, previa liberación de los recursos.

Para realizar esta prueba hemos pulsado el botón de la pantalla principal para salir de la aplicación y después hemos comprobado desde el sistema Android y desde el sistema iOS el consumo de esta aplicación confirmando que hemos salido y liberado los recursos de forma correcta.

Dispositivo	Memoria(MB)
Nexus 5	1320
iPhone 6	737
Aplicación de escritorio - Mac OS X	6963
Navegador Chrome - Mac OS X	6881

Tabla 12.1: Caso de Prueba: Aplicación en ejecución

Dispositivo	Memoria(MB)
Nexus 5	1126
iPhone 6	614
Aplicación de escritorio - Mac OS X	6799
Navegador Chrome - Mac OS X	6799

Tabla 12.2: Caso de Prueba: Recursos liberados

12.2.2. Caso de prueba 2 - Activar sonido

Esta prueba consistirá en la activación del sonido en nuestra aplicación.

Para realizar esta prueba hemos pulsado el botón sonido de la pantalla principal para activarlo. Una vez pulsado hemos jugado una partida y hemos podido comprobar que no se escuchaba ningún sonido. La prueba ha sido incorrecta porque no se guardaba de forma correcta la variable interna que representaba cuando el sonido estaba activado.

Para solucionar esta prueba hemos cambiado el identificador con el que almacenar la variable interna que representa cuando el sonido esta activo o no.

12.2.3. Caso de prueba 3 - Jugar una partida en modo Jugar

Esta prueba consistirá en jugar una partida en el modo Jugar.

Para realizar esta prueba hemos pulsado el botón del modo Jugar, este modo nos permite jugar en una misma prueba a todos los juegos de la aplicación, hemos realizado las seis pruebas de forma satisfactoria por lo que la prueba ha sido correcta.

12.2.4. Caso de prueba 4 - Reiniciar una partida

Esta prueba consistirá en reiniciar una partida.

Para realizar esta prueba hemos pulsado el botón reiniciar al finalizar una partida en el modo Jugar y se ha realizado de manera incorrecta, pues en vez de reiniciar la partida hemos salido al menú principal.

El error consistía en que el botón de reiniciar una partida no enlazaba de forma correcta a comenzar una partida sino que lo estaba haciendo al menú principal.

Para solucionar esta prueba hemos cambiado la acción del botón reiniciar.

12.2.5. Caso de prueba 5 - Seleccionar dificultad

Esta prueba consistirá en seleccionar una de las dificultades dentro de la pantalla selección de dificultad.

Para realizar esta prueba hemos navegado hasta la pantalla de selección de dificultad y hemos realizado pruebas con todas las dificultades obteniendo resultados incorrectos,

todos los resultados nos devolvían la misma dificultad.

El error consistía en que los botones de selección de dificultad no realizaban ninguna acción, simplemente se estaba seleccionando la dificultad fácil de forma predeterminada.

Para solucionar esta prueba hemos añadido una variable interna para controlar en memoria la dificultad seleccionada.

12.2.6. Caso de prueba 6 - Introducir números en juego matemáticas

Esta prueba consistirá en la introducción de números en el juego matemáticas desde el teclado virtual.

Para realizar esta prueba hemos comenzado una partida en el juego matemáticas y hemos probado a pulsar todos los caracteres posibles, algunas de las pulsaciones han causado excepciones que nos han obligado a cerrar la aplicación y otras pulsaciones han pintado un número distinto al seleccionado, la prueba ha sido incorrecta.

El error consistía en que al introducir algunos de los números saltaba una excepción y se cerraba la aplicación, el fallo estaba en que el recurso del numero a pintar no estaba cargado en memoria.

Para solucionar esta prueba hemos añadido la carga de los recursos restantes asociados a los números que daban error.

12.2.7. Caso de prueba 7 - Pulsar bolas en el juego visual

Esta prueba consistirá en la pulsación de las bolas en el juego visual.

Para realizar esta prueba hemos comenzado una partida en el juego visual y hemos probado a pulsar varias combinaciones de bolas, la aplicación ha respondido correctamente por lo que la prueba ha sido correcta.

12.2.8. Caso de prueba 8 - Seleccionar carta en el juego memoria

Esta prueba consistirá en la selección de carta en el juego memoria.

Para realizar esta prueba hemos comenzado una partida en el juego memoria y hemos probado a pulsar varias combinaciones de carta, tanto la carta a pulsar cuando fuera correcta como cuando fuera incorrecta, la prueba se ha realizado de manera correcta.

12.2.9. Caso de prueba 9 - Seleccionar carta en el juego asociación

Esta prueba consistirá en la selección de carta en el juego asociación.

Para realizar esta prueba hemos comenzado una partida en el juego memoria y hemos probado a pulsar varias combinaciones de carta, tanto carta para comenzar una pareja, como la carta para terminar una pareja y cuando la carta a seleccionar no era la

correcta, los resultados han sido correctos por lo que la prueba se ha realizado de manera correcta.

12.2.10. Caso de prueba 10 - Comprobación de secuencia en el juego secuencias

Esta prueba consistirá en la comprobación de secuencia en el juego secuencias.

Para realizar esta prueba hemos comenzamos una partida en el juego secuencias y hemos probado a realizar la secuencia de forma correcta y de forma incorrecta, los resultados eran incorrectos por lo que la prueba no ha sido correcta.

12.2.11. Caso de prueba 11 - Seleccionar de peso en el juego pesos

Esta prueba consistirá en la selección de peso en el juego pesos.

Para realizar esta prueba hemos comenzado una partida en el juego pesos, hemos pulsado un peso correcto obteniendo una salida positiva y a continuación hemos pulsado un peso incorrecto obteniendo una salida negativa, los resultados de este caso de prueba han sido satisfactorios.

12.2.12. Caso de prueba 12 - Obtener máxima puntuación

Esta prueba consistirá en obtener la máxima puntuación de cualquier modo o juego.

Para realizar esta prueba hemos finalizado una partida en cada modo y juego y hemos comprobado si la puntuación que devolvía era la correcta, en algunos de los casos devolvía una máxima puntuación que no correspondía a la pedida, la prueba ha sido incorrecta.

El fallo consistía en que al obtener la puntuación en cualquiera de los juegos siempre nos devolvía la máxima puntuación del juego matemáticas.

12.2.13. Caso de prueba 13 - Carga de recursos

Esta prueba consistirá en el tiempo consumido en la carga de recursos. Se desea comprobar si la aplicación es capaz de realizar una carga de recursos con cuotas inferiores a los 15 segundos.

Dispositivo	Tiempo(s)	Couta
Nexus 5	8.2	Inferior
Nexus 7	7.9	Inferior
iPhone 5	8.5	Inferior
iPhone 6	7.4	Inferior
Aplicación de escritorio - Mac OS X	8.9	Inferior
Navegador Chrome - Mac OS X	10.8	Inferior

Tabla 12.3: Caso de Prueba: Carga de recursos

12.2.14. Caso de prueba 14 - Detección de puntos para el juego visual

Esta prueba consistirá en probar cuánto tiempo tardamos en generar puntos para el juego visual basándonos en la distancia entre los objetos a dibujar en los puntos. Se probara el algoritmo de generación de puntos en el mapa del juego con distintas distancias entre los objetos y se medirá el tiempo que tarda en generarse. De esta manera buscaremos una características óptimas para la generación del algoritmo.

Número de objetos	Distancia(px)	Pruebas	Tiempo de ejecución(ms)
3	500	10	351
4	500	10	1781
3	400	10	351
4	400	10	494
3	300	10	182
4	300	10	286
3	200	10	156
4	200	10	195
3	100	10	143
4	100	10	143

Tabla 12.4: Caso de Prueba: Detección de puntos para el juego visual

Una vez aplicadas las distintas pruebas diseñadas en este capítulo y observando los resultados obtenidos tras su aplicación, se han llevado a cabo las modificaciones necesarias sobre el software eliminando las posibles causas de provocar fallos en la aplicación.

Capítulo 13

Conclusiones

En este apartado se realiza un análisis de las conclusiones obtenidas, abarcando los conocimientos obtenidos en el desarrollo del sistema, los objetivos cumplidos y el mantenimiento y futuras mejoras que se pueden aplicar al software.

13.1. Cumplimiento de objetivos

En relación a los objetivos planteados en el apartado 3, se van a analizar los objetivos cumplidos y en qué grado:

- Se ha creado una aplicación con una colección de videojuegos de destreza en el que el usuario puede divertirse mediante la realización de estas pruebas.
- Se ha creado una aplicación que es compatible con cualquier dispositivo cuya versión del sistema sea igual o superior a Android 4.0 *"ICE CREAM SANDWICH"*, igual o superior a iOS 9.0, que contenga un navegador web o que tenga Java integrado.
- Se ha analizado y estudiado el entorno de desarrollo y las herramientas internas y externas que ofrece el SDK de la librería libGDX para el desarrollo de videojuegos.
- Se ha desarrollado un videojuego en dos dimensiones utilizando las herramientas seleccionadas en el análisis.

13.2. Conocimientos adquiridos

A día de hoy, se han ampliado los conocimientos suficientes como para afrontar el desarrollo de un videojuego multiplataforma a través de la librería libGDX.

Por otra parte, las necesidades del proyecto y, en concreto, las limitaciones del desarrollo para plataformas móviles en cuanto a recursos se refiere, han permitido ver la importancia del uso mínimo de recursos y de la correcta gestión de la memoria vistas durante el desarrollo de las clases de la titulación y que, ahora, se han tenido que aplicar de forma exhaustiva para realizar una aplicación robusta y optimizada.

13.3. Futuras mejoras

A continuación se exponen una serie de posibles mejoras que se pueden aplicar a la aplicación para aumentar sus funcionalidades:

- Añadir más videojuegos a la colección.
- Realizar un sistema multijugador, donde sea posible competir en tiempo real.
- Añadir funcionalidades extras como un chat en línea.

Bibliografía

[1] Videojuego Who Has The Biggest Brain. hhttps://en.wikipedia.org/wiki/Playfish#List_of_games (107-11-2017).

[2] Videojuego Brain Training. http://www.nintendo.es/Juegos/Nintendo-DS/Brain-Training-del-Dr-Kawashima-Cuantos-anios-tiene-tu-cerebro--270627.html (07-11-2017).

[3] Videojuego Big Brain Academy. http://www.nintendo.es/Juegos/Nintendo-DS/Big-Brain-Academy-270143.html (07-11-2017).

[4] GitHub Librería libGDX. https://github.com/libgdx/libgdx (07-11-2017).

[5] Pagina oficial de la librería libGDX. http://libgdx.badlogicgames.com/ (07-11-2017).

[6] ADT Plugin for Eclipse Android Developers. http://developer.android.com/tools/sdk/eclipse-adt.html (07-11-2017).

[7] iOS SDK Developers. https://developer.apple.com/ios/ (07-11-2017).

[8] Android SDK Android Developers. http://developer.android.com/sdk/index.html (07-11-2017).

[9] Android Studio. https://developer.android.com/studio/ (07-11-2017).

[10] GIMP - The GNU Image Manipulation Program. http://www.gimp.org/ (07-11-2017).

[11] Product Design Specification - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Product_design_specification (07-11-2017).

[12] Oracle Technology Network for Java Developers http://www.oracle.com/technetwork/java/index.html (07-11-2017).

[13] xCode. https://developer.apple.com/xcode (07-11-2017).

[14] Diagrama de clases - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Diagrama_de_clases (07-11-2017).

[15] Kile Latex. http://kile.sourceforge.net/ (07-11-2017).

[16] ShareLatex. https://www.sharelatex.com/ (07-11-2017).

[17] R. S. Pressman. Ingeniería del Software - Un enfoque práctico. Luis Joyanes, Aguilar, 2002

[18] Grady Booch, James Rumbaugh e Ivar Jacobson. El Lenguaje Unificado de Modelado. Madrid, Pearson Addison Wesley, 2006.

[19]

[20] Diagrama de casos de uso - Wikipedia, la enciclopedia libre. `https://es.wikipedia.org/wiki/Diagrama_de_casos_de_uso` (04-06-2018).