Proyecto Final de Carrera - Título

Desarrollo de Videojuegos sobre la plataforma Android.

Autor: *David García Povedano*

Fecha: *10 de Enero de 2012*

Director: *Lluís Solano Albajes*

Departamento: *LSI*

Titulación: *Ingeniería Técnica en Informática de Gestión*

Centro: *Facultad de Informática de Barcelona (FIB)*

Universidad: *Universidad Politécnica de Cataluña (UPC)*

BarcelonaTech

Agradecimientos

Me gustaría agradecer especialmente el apoyo, la paciencia y la ayuda constante que he recibido por parte de mis padres y de mi hermana a lo largo de la carrera y, en especial, durante la realización del proyecto.

Por supuesto, también quería agradecer la ayuda que me han prestado mis amigos haciendo de *beta-testers* de los dos videojuegos desarrollados durante el proyecto. Así como la ayuda prestada por mi director de proyecto, el cual me ha dado libertad durante el transcurso de este para hacer lo que realmente quería y me ha aportado unos consejos muy útiles.

Por último, no me olvido de mi fiel amigo Scotty, el cual estuvo a mi lado durante toda la carrera y durante prácticamente la totalidad del proyecto.

Capítulo 1 – Introducción

En este primer capítulo de la memoria se explica la motivación que me hizo llevar a cabo este proyecto, así como la descripción detallada de dicho proyecto. Sin olvidar los objetivos que el proyecto persigue. Y, por último, una guía acerca de los diferentes capítulos de la memoria, por tal de facilitar la lectura de esta.

1. Motivación

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.

Son varios los sistemas operativos que existen para teléfonos inteligentes, sin embargo, Android está pegando fuerte en el mercado actual, al tratarse de software libre que además da 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, junto con la curiosidad por el S.O. de Google en concreto, es lo que me ha llevado a tratar este tema en mi proyecto de final de carrera.

Por otro lado, tenemos al videojuego, cada vez más presenten en nuestra sociedad. Una forma relativamente nueva de expresarnos y enviar nuestros conocimientos o emociones a los jugadores, que los recibirán y percibirán de una forma diferente a como lo hacen cuando ven una película o leen un libro.

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

Desde siempre me ha fascinado la idea de poder crear un mecanismo de juego, unos escenarios, unos personajes, etc. y que las personas puedan llegar a divertirse, disfrutar o aprender con ello, así que desde el principio pensé en combinar ambas cosas y de ahí salió la idea del proyecto.

2. Descripción del proyecto

El proyecto consiste en el estudio, análisis y puesta en práctica del desarrollo de videojuegos **en dos dimensiones** sobre la plataforma Android.

En concreto se ha estudiado el entorno de programación disponible y las herramientas y conceptos que se deben utilizar para desarrollar aplicaciones que funcionen en el sistema operativo Android. Así como el funcionamiento interno de dicho S.O.

Se ha continuado con el análisis, selección y estudio de las herramientas apropiadas para la construcción de un videojuego. Desde la selección de la librería gráfica hasta la búsqueda de motores de físicas que nos permitan facilitar toda la gestión de la física del juego, pasando por otros procesos de análisis, todos ellos explicados en capítulos posteriores.

En esta selección se ha premiado que, aparte de potente, eficiente y fácil de usar, la herramienta sea multiplataforma, facilitando la portabilidad a otros S.O. móviles así como a ordenadores de sobremesa u otras plataformas.

En último lugar se han desarrollado dos videojuegos:

El primero es una aplicación simple para poner a prueba los conocimientos adquiridos durante la fase de estudio y análisis del proyecto, así como los conocimientos adquiridos durante la carrera, algunos de los cuales deberán ser adaptados al S.O. móvil Android.

El segundo proyecto consiste en un videojuego más complejo que tiene unos objetivos más amplios. Para su desarrollo se sigue el ciclo de vida clásico como metodología de desarrollo y se intenta conseguir la máxima extensibilidad, portabilidad y reutilización; sin dejar atrás la eficiencia, muy importante cuando estamos tratando con unos dispositivos que en comparación con los ordenadores de sobremesa tienen una potencia baja.

El fomentar la extensibilidad nos va a permitir, una vez terminado el juego, añadir pantallas, elementos móviles, etc. con relativa facilidad. Es vital este punto ya que, como veremos con detalle en capítulos posteriores, el negocio de muchos de los videojuegos publicados para Android radica en la publicación de unos primeros niveles de juego gratuitos y el posterior lanzamiento de expansiones a un precio muy asequible, las cuales incluyen nuevos niveles y elementos con los que interactuar.

3. Objetivos del proyecto

3.1. Objetivos principales

- Analizar y estudiar el entorno de desarrollo y las herramientas internas o externas que el sistema operativo Android tiene para el desarrollo de videojuegos.
- Estudiar el funcionamiento interno de Android. Así como la estructura de un videojuego.
- Desarrollar dos videojuegos en dos dimensiones utilizando las herramientas seleccionadas en el análisis.

3.2. Objetivos concretos

Los objetivos concretos a alcanzar en lo que respecta a los dos productos que se desarrollarán son:

1. Realizar un primer producto simple pero funcional en el que se integren los conocimientos adquiridos. Teniendo en cuenta parte de los resultados del análisis de antecedentes.
2. Realizar un segundo producto más complejo donde se tengan en cuenta la totalidad de las conclusiones extraídas del análisis de antecedentes, de forma que desarrollemos un producto que pueda llegar a funcionar en el mercado.
3. Conseguir dotar al segundo producto de las propiedades de extensibilidad, reutilización y portabilidad, por tal de facilitar su ampliación y posibilitar la creación de segundas partes aprovechando la misma base. Siempre sin olvidar la eficiencia, que deberá ser la suficiente para que el producto funciona en dispositivos móviles de gama media/baja.
4. Obtener la capacidad de acotar un coste en tiempo y dinero para la realización de un videojuego sobre la plataforma Android a través de la experiencia adquirida con los dos productos construidos.

4. Estructura de la memoria

En este apartado se describe la estructura que presenta la memoria, por tal de facilitar la lectura de esta.

En primer lugar, el capítulo 1 como vemos, presenta la introducción, la motivación del proyecto, la descripción de este y los objetivos. El capítulo 2, en cambio, trata acerca del S.O. Android y, entre otras cosas, se describe su estructura y el funcionamiento de sus aplicaciones. Además, trata el tema desde el punto de vista del desarrollo del videojuego, con lo que también se describe la arquitectura de un videojuego en Android. Como no, también encontraremos los importantísimos análisis de antecedentes y el análisis de herramientas para el desarrollo de videojuegos.

Una vez que ya conocemos los conceptos básicos acerca de Android, en el capítulo 3, se habla sobre el primer videojuego desarrollado, con nombre "Ping Pang Pung". Podemos ver una descripción general de este, así como las partes más importantes de su especificación y de su diseño.

En el capítulo 4, se hace lo propio con el segundo videojuego "Terrestrial Blast!", un producto mucho más trabajado que el primero. En este capítulo, bastante más extenso que el anterior, se hace especial hincapié en el diseño de la estructura del videojuego, pues se ha querido desarrollar un videojuego extensible, portable y reutilizable. Aunque no se deja de prestar atención a los apartados de descripción y especificación del proyecto. También se habla de la fase de implementación, una fase muy interesante puesto que se describen las mejoras de eficiencia llevadas a cabo durante el desarrollo del proyecto y los beneficios que estas han reportado.

A continuación, tenemos el capítulo 5, en él se presenta la planificación del proyecto y se compara con las horas invertidas realmente. También se presenta el orden en que se han realizado las diferentes tareas. Y no puede faltar el apartado donde se detalla el coste económico del proyecto.

Finalmente, un último capítulo, el capítulo 6, presenta las conclusiones del proyecto.

Sin embargo, no se acaba aquí la memoria, pues también hay dos anexos. En el primero, se encuentra el manual de usuario de "Terrestrial Blast!" donde se explica, a nivel de usuario, todo lo necesario para poder jugar al videojuego. El segundo anexo tiene una gran extensión, pues explica cómo crear niveles y elementos con el motor del segundo videojuego mediante el lenguaje XML. Y es que el motor desarrollado para "Terrestrial Blast!" está pensado para utilizarse desde un nivel de abstracción mayor al nivel de abstracción que ofrece el lenguaje Java.

Capítulo 2 – Estudio y Análisis

Cuando nos disponemos a desarrollar una aplicación para un nuevo sistema operativo con el que no se hemos trabajado con anterioridad, lo primero que hay que hacer es estudiar el funcionamiento de este, así como el funcionamiento de sus aplicaciones. Especialmente si se trata de un S.O. móvil, pues estos son bastante cerrados y las aplicaciones que desarrollemos para ellos se estructurará de forma diferente a como se estructuran en otros sistemas operativos.

Igual o más importante es analizar las diferentes herramientas con las que podemos trabajar para desarrollar una aplicación que funcione en el S.O. objetivo, por tal de poder escoger las herramientas que más se ajustan a nuestros requisitos. Y, por último, también es útil llevar a cabo un análisis de las aplicaciones del mismo ámbito desarrolladas para dicho S.O., pues estas nos pueden ayudar a decidir los objetivos concretos de nuestra aplicación y a evitar errores durante el desarrollo.

En resumen, el estudio y el análisis del sistema operativo Android ha sido un primer paso muy importante antes de embarcarnos en el desarrollo de los videojuegos. En este capítulo se explican los conceptos más importantes estudiados y se exponen los análisis llevados a cabo para poder desarrollar videojuegos para Android cometiendo el mínimo número de errores posibles en todos los ámbitos.

Empezaremos por una explicación de qué es el sistema operativo Android. Seguiremos con la estructura de las aplicaciones para este sistema operativo. Continuaremos con el entorno de desarrollo que Google ha preparado para el desarrollador de Android y explicaremos cómo se adapta la estructura de un videojuego a este S.O.

Por último, se describirán los dos importantes análisis llevados a cabo: un análisis de los videojuegos más exitosos de Android, con el fin de extraer las claves para que nuestras aplicaciones tengan éxito y para que no cometamos errores en su desarrollo; y un análisis de las herramientas disponibles para llevar a cabo el desarrollo de videojuegos, con las ventajas e inconvenientes de las diferentes alternativas.

1. ¿Qué es Android?

Android es un conjunto de software que constituye un ecosistema para las aplicaciones móviles. Dentro de este conjunto se incluye un sistema operativo móvil, lo que significa que Android está dirigido principalmente a teléfonos inteligentes (o *smartphone*) y a *tablets*.

Y detrás de este software se encuentra la empresa Google Inc., archiconocida multinacional dedicada a ofrecer servicios y productos basados generalmente en Internet.

1.1. Estructura de Android

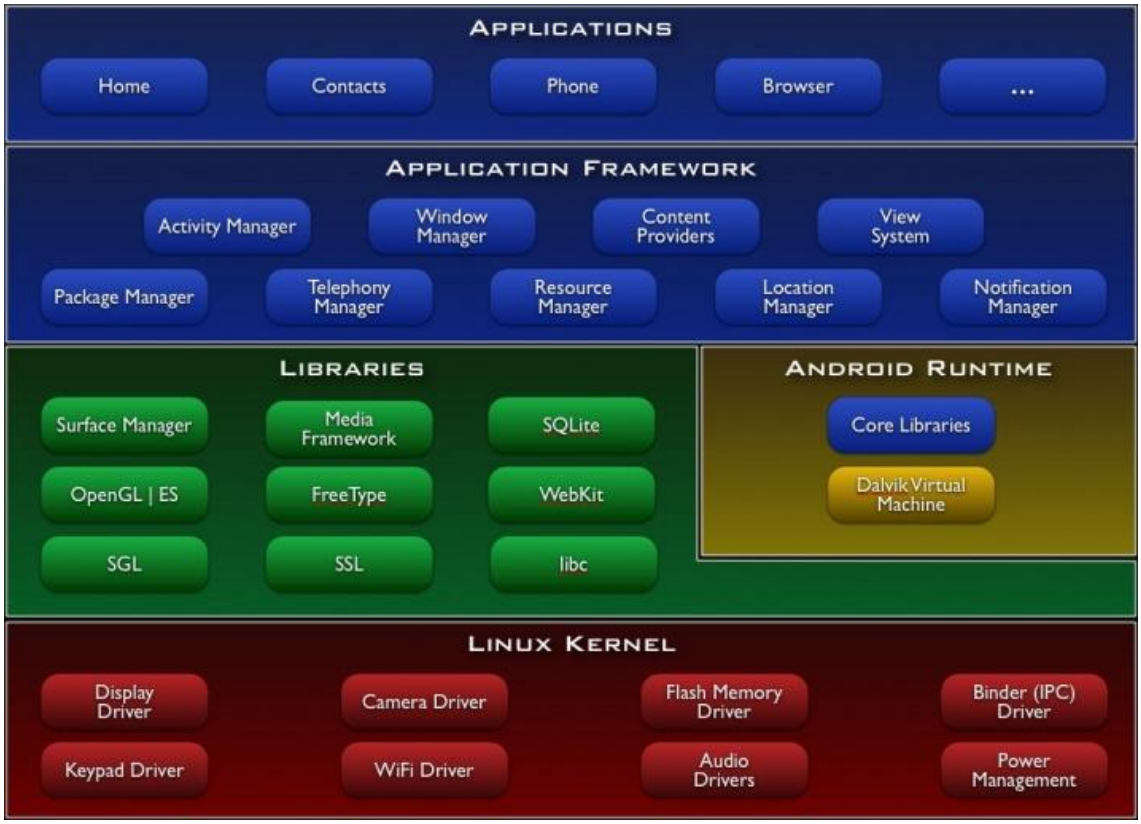


Figura 2.1: Estructura interna del conjunto de software Android.

El conjunto de software denominado Android incluye: un sistema operativo, software intermedio que trabaja al servicio de las aplicaciones que se encuentran por encima y algunas aplicaciones claves que vienen incluidas desde el principio con el sistema operativo. Un ejemplo de aplicación por defecto es el Android Market, la tienda desde donde podemos comprar o descargar gratuitamente las aplicaciones que los desarrolladores ofrecen a través de ella. Para ver la estructura de todo el paquete Android tenemos la Figura 2.1.

Como podemos ver en la figura, el núcleo del sistema operativo es una modificación del núcleo de Linux. En él se encuentran los drivers que permiten comunicarse con el hardware específico de cada dispositivo que implementa Android. Por encima tenemos una capa con todas las librerías, accesibles a la hora de programar aplicaciones, las cuales hacen uso de los drivers implementados. Entre estas librerías encontramos OpenGL ES, SQLite, SSL, etc.

Dentro del sistema operativo también encontramos la famosa maquina Dalvik. Y es que, en Android, las aplicaciones se ejecutan en una instancia de la máquina Dalvik. Cada instancia es independiente y, por tanto, ejecuta una aplicación de forma cerrada. Este es un buen mecanismo de seguridad, pues nadie puede llegar a entrometerse en la ejecución de una aplicación. De igual forma, los recursos de cada aplicación se encuentran en un fragmento de memoria privada e inaccesible desde fuera de la aplicación.

Y hasta aquí el sistema operativo, ya que en la siguiente capa, encontramos una serie de componentes utilizados por las aplicaciones para realizar funciones determinadas. Entre estos componentes, por ejemplo, se encuentra el Notification Manager, el cual recibe notificaciones de las aplicaciones y las presenta al usuario a través de la barra de notificaciones. Otro ejemplo es el Activity Manger, el cual se encarga de la gestión de las actividades de cada aplicación (veremos que es una actividad en el segundo apartado de este mismo capítulo).

Como estos componentes que conforman el *framework* manejan una información sensible para el usuario, pues en un dispositivo móvil normalmente el usuario introduce bastante información personal, se ha desarrollado un mecanismo de permisos por tal de mantener la información de forma segura.

De esta forma, cuando la aplicación se instala, esta solicita al usuario permisos para acceder a los componentes del *framework* que sean necesarios por tal de llevar a cabo las acciones pertinentes. El usuario deberá aceptar estos permisos si quiere instalar la aplicación, ya que, en caso contrario, esta aplicación no podrá utilizar los poderes que el permiso solicitado otorga.

Si seguimos con la estructura de Android, un nivel por encima tenemos las aplicaciones, que pueden estar creadas por desarrolladores externos o por la propia Google. De entre las creadas por la empresa Google destacan unas cuantas que, como hemos dicho antes, son clave y vienen incluidas en el paquete.

Aunque, como hemos visto Android es más que un S.O., cuando se habla del sistema operativo Android, se suele incluir todo el paquete de software. Así que durante el resto de la memoria nos dirigiremos de esta forma a la totalidad del software que se engloba dentro del nombre Android.

1.2. Historia y éxito del S.O.

Android (ver Figura 2.2) es un software desarrollado por Google posteriormente a la compra de la empresa Android Inc., de donde saco material y profesionales para crear el sistema. Por tanto, uno de los principales reclamos del S.O. es la total integración con los servicios de la empresa Google Inc.

Figura 2.2: Logo del paquete de software Android.

El objetivo de Google a la hora de crear y mantener Android es crear un ecosistema móvil estándar y abierto, por tal de satisfacer las necesidades de fabricantes de dispositivos móviles y *tablets*.

Desde su salida al mercado en el año 2008, Android ha ido escalando puestos en el ranking de ventas, hasta llegar a ser el sistema operativo que llevan el 52.5% de los teléfonos inteligentes vendidos en el tercer cuarto de 2011. Seguido de lejos por Symbian, sistema operativo de la empresa Nokia, el cual solo representa el 16.9%. Todo esto según un estudio realizado por la empresa Gartner [1].

Es más, si comparamos estos datos con los obtenidos en el estudio de la empresa Gartner en el mismo cuarto del año 2010, donde el porcentaje de móviles con S.O. Android era tan solo del 25.3%, concluimos que el crecimiento ha sido del 27.2%.

El secreto de este crecimiento tan rápido se debe a las características de las que presume el ecosistema Android. En primer lugar, como ya se ha explicado, Android está pensado para satisfacer los requerimientos de algunas de las empresas fabricantes de *smartphones* y *tablets*, aliadas bajo el nombre de Open Handle Alliance.

Estas empresas necesitan un ecosistema estándar, que sea ampliamente utilizado por todas las empresas fabricantes de dispositivos móviles que lo deseen. Y es que esto les permite obtener dos beneficios importantes:

- Por un lado, las empresas fabricantes del hardware, no necesitan embarcarse en el proceso de desarrollo de un sistema operativo propio, el cual no es un proceso sencillo ni barato. Además, estas empresas tienen disponible un S.O. creado y mantenido por la reconocida empresa Google, la cual dota de calidad a todas sus creaciones.

- Por otro lado, en un ecosistema móvil es muy importante el número de aplicaciones que estén disponibles, pues son estas aplicaciones las que dan utilidad al dispositivo.

  Teniendo un sistema operativo común para diferentes empresas, se consigue la compatibilidad con las mismas aplicaciones por parte de todos los dispositivos. Y, de esta forma, empresas sin tanto renombre puedan competir contra grandes empresas como son Nokia o Apple, las cuales por si solas y con sus propios S.O. móviles consiguen ventas muy importantes y mueven a un gran número de desarrolladores.

  En este sentido, Android es además un sistema muy abierto, lo que se traduce en muchas facilidades para los desarrolladores, factor que incrementa aún más el número de aplicaciones que podemos encontrar disponibles para el ecosistema.

Pero empresas importantes como Samsung, LG o HTC no se conforman con eso, pues estas quieren tener la opción de, de forma fácil y barata, personalizar la parte visual de Android, por tal de darle un toque único que se adapte a las características que estos quieren ofrecer a sus usuarios. Android, al ser un sistema muy abierto, permite con facilidad esta personalización, tanto por parte de fabricantes de dispositivos móviles, como por parte de otros desarrolladores que quieran crear su propia interface.

Hay que decir, además, que el sistema operativo Android es de código abierto lo que aun facilita más la labor de las empresas que quieran implementar este S.O. en sus terminales, puesto que estos pueden acceder al código. Eso sí, para obtener la compatibilidad con Android, el fabricante no puede modificar el código completamente a su antojo, hay una serie de requisitos que deberá cumplir. En caso de no cumplirlos, no se pasará el test de compatibilidad y no se estará implementando el ecosistema Android.

Por si fuera poco, además Android es un sistema operativo gratuito, así que el fabricante no tiene que pagar ninguna licencia. Google recibe beneficios económicos a través de la tienda de aplicaciones y de sus servicios, los cuales se financian principalmente mediante publicidad.

En resumen, el único trabajo que deberá llevar a cabo el fabricante que quiera implementar el ecosistema, es el desarrollo de los drivers necesarios para que Android sepa comunicarse con el hardware del dispositivo, el resto queda en manos de si el fabricante quiere personalizar más o menos el sistema. Por tanto, la inversión en software de estos fabricantes puede reducirse mucho gracias a Android.

Esta versatilidad de Android y la idea de tener un ecosistema común es la que ha hecho que un gran número de fabricantes añadan el ecosistema a sus dispositivos móviles y que, por tanto, este S.O. goce de gran popularidad.

El segundo punto importante que justifica el crecimiento tan rápido que ha experimentado el S.O. tiene que ver con el desarrollador de aplicaciones. Y es que este desarrollador tiene muchas ventajas a la hora de desarrollar para el sistema operativo Android.

Para empezar, publicar una aplicación en el Android Market (la tienda de aplicaciones de Google) es muy barato, únicamente necesitamos una licencia que tiene un coste de 25$ (unos 20€) para poder publicar aplicaciones durante un tiempo ilimitado. A parte de este pago, Google se lleva el 30% de las ganancias obtenidas con la aplicación, si es que esta es de pago. En caso de ser gratuita, no será necesario pagar nada más. Además, el entorno de desarrollo se puede montar tanto en Windows como en Linux e incluso Mac O.S.

Todas estas facilidades hacen que desarrollar para Android sea bastante más barato que hacerlo para otros ecosistemas móviles.

Las aplicaciones además, no sufren ningún tipo de control a la hora de publicarse. Desde el punto de vista de la libertad del desarrollador y de los usuarios esto es una ventaja, pues en el Android Market encontramos aplicaciones que en otras tiendas de aplicaciones de la competencia son censuradas, simplemente por ir en contra de la política de la empresa o por tener contenidos para adultos. Pero también tiene su parte negativa, pues cualquier desarrollador puede publicar una aplicación malintencionada.

Para evitar que el usuario instale una aplicación con fines maliciosos sin darse cuenta, se ha desarrollado el sistema de permisos explicado con anterioridad y que recordamos a continuación. Cualquier aplicación que se instale en el dispositivo y quiera llevar a cabo una serie de accesos a la información externa a la propia aplicación, deberá solicitar al usuario los diferentes permisos necesarios para llevar a cabo estas acciones.

Así que, si por ejemplo, el usuario ve que un videojuego está pidiendo permiso para acceder a la información de los contactos o de las llamadas realizadas, podrá decidir no instalarlo, pues es sospechoso que este tipo de aplicación requiera tales permisos. En cambio, si instala el videojuego estará aceptando estos permisos y, por tanto, la aplicación podrá acceder a dicha información.

Como vemos se trata de un sistema que confía en la prudencia de los usuarios y, al mismo tiempo, en la buena fe de los desarrolladores. Por el momento, este sistema tan abierto está provocando algunas quejas por parte de algunos usuarios que lo consideran poco seguro.

Pero es que el termino abierto, en el sistema operativo Android, se extiende más allá de lo citado anteriormente, pues tampoco hay ninguna restricción que impida instalar aplicaciones externas al Android Market. Por tanto, las aplicaciones, se pueden instalar mediante ejecutables introducidos directamente dentro de la memoria del dispositivo o mediante otras tiendas de aplicaciones gestionadas libremente por otras empresas. Eso sí, la política de permisos se extiende más allá del Android Market, pues afecta a cualquier aplicación, que en caso de no obtener los permisos necesarios no podrá realizar las acciones deseada.

Podemos concluir, por tanto, que el S.O. Android ha cosechado éxito gracias a haberse adaptado a lo que los fabricantes demandaban, así como gracias a la facilidad blindada a los desarrolladores por tal de que puedan realizar sus aplicaciones para el ecosistema de forma barata y libre. Ecosistema libre, abierto y gratuito son, por tanto, los principales adjetivos que definen al ecosistema Android.

2. Las aplicaciones en Android

Ya hemos visto qué es Android y qué características tiene el ecosistema. En este apartado nos centraremos en las aplicaciones compatibles. En él se explica cómo se lleva a cabo la ejecución de cada aplicación y cómo se estructuran estas.

2.1. Estructura de una aplicación

Las aplicaciones de Android se estructuran en componentes, cada componente de una aplicación juega un papel específico dentro de esta y existe por sí mismo. Eso sí, puede haber componentes que dependan unos de otros.

Es decir, a la hora de desarrollar una aplicación para Android no tendremos una sola función principal (o función *main*) la cual lanzará las diferentes pantallas de la aplicación. Sino que tendremos varios componentes independientes cuya comunicación será llevada a través del S.O.

Además, estos componentes, lejos de tener cada uno una sola función *main* la cual nos permitiría llevar a cabo una ejecución secuencial de la aplicación, implementan una serie de funciones que serán llamadas por el S.O. cuando se cumpla la condición necesaria para llamar a cada función. Por tanto, las aplicaciones en Android funcionan de forma asíncrona y es el S.O. el que ante las peticiones del usuario va llamando a una y otra función de los diferentes componentes de la aplicación, según convenga.

A través de la API de Android podemos declarar cuatro tipos de componentes: Actividades (Activities), Servicios (Services), Proveedores de Contenido (Content Providers) o Receptores de Transmisión (Broadcast Receivers).

Las **Actividades** son los componentes más importantes, de hecho los videojuegos desarrollados se componen exclusivamente de este tipo de componentes. Una actividad es una pantalla de la aplicación, con su interface de usuario, es decir, lo que en el desarrollo de aplicaciones se conoce habitualmente como *vista*.

Por ejemplo, en una aplicación que implementa un cliente de correo electrónico, la pantalla que nos permite rellenar un mensaje de texto y nos presenta el botón de envió, junto con el mecanismo para dar respuesta a las diferentes acciones que el usuario pueda llevar a cabo en esta pantalla, sería una actividad. Otra actividad de esta aplicación sería la pantalla que nos presenta una lista con todos los correos que nos han llegado, nuevamente junto con la lógica necesaria para atender las peticiones del usuario.

Un **Servicio** [2], en cambio, es un componente que no tiene interface propia, sino que se ejecuta en segundo plano y lleva a cabo tareas largas que consuman muchos recursos.

Por ejemplo, en caso de querer implementar un reproductor de música, este tendrá un servicio que reproduce esta música. Al mismo tiempo, la aplicación tendrá una actividad que permite que el usuario escoja la canción deseada. La actividad se comunicará con este servicio para pausar, reanudar o cambiar de canción.

En cuando al **Proveedor de Contenido** [3], este es un tipo de componente que permite almacenar una información para que después pueda ser accedida desde varias aplicaciones. Según cómo configuremos al Proveedor limitaremos dicho contenido a unas aplicaciones muy concretas o no. Se trata del único mecanismo para compartir información entre diferentes aplicaciones que existe en Android.

Por último, se encuentra el **Receptor de Transmisión**, este crea una notificación que se presentará en la Barra de Notificaciones del sistema. Normalmente, una notificación contiene una información acerca de lo que se pretende informar y un apuntador a una actividad o servicio que será lanzado cuando se presione sobre la notificación.

Hay que tener en cuenta que estos componentes son representados por clases Java, lo que significa que, cuando la aplicación se ejecuta, se crean instancias de estos, pudiendo crear más de una instancia del mismo componente. Dichas instancias además son independientes.

2.1.1. Las Actividades

Seguidamente se detallará el funcionamiento de un componente de tipo Actividad [4], pues como ya se ha dicho, es el tipo de componente más importante que encontramos en una aplicación y el único utilizado para implementar los videojuegos del proyecto.

Toda aplicación debe tener, al menos, una actividad. Se trata de la actividad Main, la cual se lanzará cuando el usuario indique que quiere ejecutar la aplicación en cuestión.

Para crear un componente del tipo actividad, deberemos crear una clase en Java que herede de la clase *Activity*. Y esta clase deberá definir una interface que la actividad presentará al usuario. Android da una serie de herramientas para definir estas interfaces utilizando el lenguaje XML, lo que nos permite separar, en la medida de lo posible, la parte visual de la actividad de la parte lógica. Entendiendo como parte lógica la parte que, ante una entrada del usuario, hace una serie de operaciones para darle una respuesta. Esta lógica se implementa en lenguaje Java.

La definición de una interface para una actividad en Android se lleva a cabo mediante la construcción de un árbol de elementos gráficos llamados *views* (ver Figura 2.3). Las *views* pueden estar prefabricadas o desarrolladas por el propio desarrollador de la aplicación, según las necesidades de este.

Una *view* es una estructura de datos que mantiene parámetros y contenidos para gestionar la interacción del usuario con un rectángulo específico. Este rectángulo es un fragmento del área de la pantalla ocupado por la actividad cuya interface está compuesta por la *view* en cuestión.

Es decir, a la hora de visualizar una actividad, esta se mostrará en un área específica de la pantalla, más grande o más pequeño según si la estamos visualizando a pantalla completa o no. La construcción de una interface consiste en dividir dicha área en rectángulos, cada uno gestionado a través de una *view* diferente.

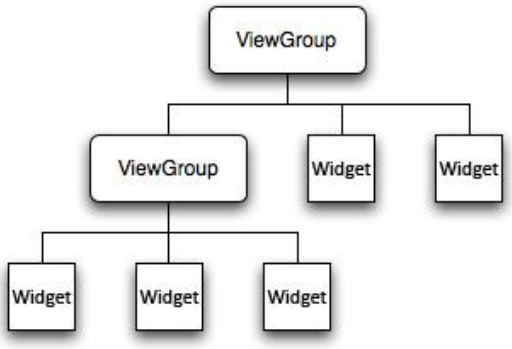


Figura 2.3: Esquema de un árbol de *views*. Son este tipo de estructuras las que definen la interface de una actividad.

Existen dos tipos de *views*: los *ViewGroup*, que son *views* que dentro pueden tener otras *views* de cualquier tipo y dotan a estas *views* interiores de una serie de propiedades concretas; y los *witgets*, que son *views* raíz, las cuales llevan a cabo algún tipo de interacción con el usuario, independiente del resto de *views*, dentro del rectángulo que se les asigna. [5]

Ejemplos de *ViewGroup* son las *layouts*, unas *views* que condicionan la forma en que el área ocupada por el *ViewGroup* se reparte entre sus *views* internas. La repartición puede ser dividiendo el área horizontalmente, verticalmente, etc. dependiendo del tipo de *layout* escogida.

En cambio, ejemplos de *widgets* son: el botón, la caja para rellenar con texto, la lista de texto, entre otras. La mayoría de *widgets* necesarios ya vienen prefabricados en la API de Android y ya traen implementada la interacción con el usuario. Por tanto, en caso de utilizarse un *widget* prefabricado, el desarrollador únicamente deberá leer los datos que el *widget* recoge del usuario (pulsado de un botón, texto insertado, etc.) y llevar a cabo con ellos el propósito que desee.

Eso sí, para leer estos datos, en lugar de llevar a cabo una comprobación constante, normalmente se deben definir una serie de operaciones que, correctamente configuradas, son llamadas por el S.O. cuando una de las *views* tiene nueva información que presentar. Nuevamente vemos como el funcionamiento de los componentes de una aplicación en Android es asíncrono y es el S.O. el que nos avisa de los sucesos cuando ocurren. En concreto, en una actividad, con el mecanismo asíncrono evitando bloquear el *thread* donde se está ejecutando dicha actividad con bucles largos, lo cual provocaría que no se pudiera llevar a cabo una interacción con el usuario.

Ciclo de vida de una actividad

Como ya hemos dicho, una actividad en Android funciona de forma asíncrona y el *thread* donde se ejecuta esta actividad debe quedar libre por tal de que los eventos que activa el usuario se reciban con rapidez.

Por este motivo, en la clase *Activity* se declaran una serie de operaciones, las cuales deberemos sobre-escribir en la actividad que creemos, pues a través de ellas llevaremos a cabo las acciones que queremos que dicha actividad haga. Estos métodos son llamadas por el S.O. cuando se dan una serie de sucesos concretos, son las operaciones a través de las cuales gestionamos el ciclo de vida de la actividad (Ver Figura 2.4).

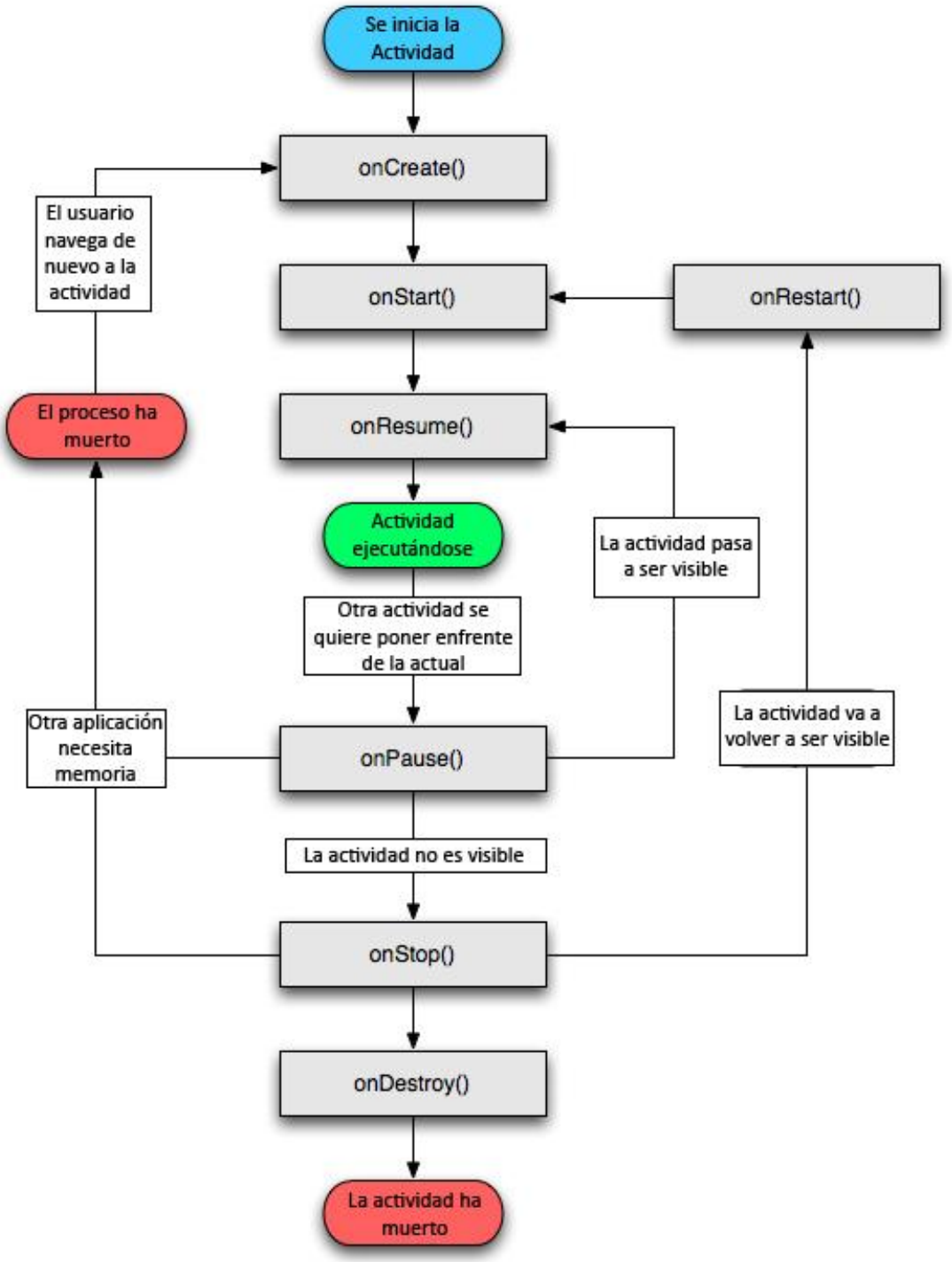


Figura 2.4: Esquema del ciclo de vida de un componente del tipo *Activity*, de una aplicación del sistema operativo Android.

A continuación se describen las operaciones a través de las cuales se gestiona el ciclo de vida de la actividad:

- *onCreate()*: Operación que llama el sistema operativo a la hora de crear la instancia de la actividad.

  Esta operación es muy importante, pues es donde tenemos que introducir la jerarquía de *views* que define la interface de la actividad. Esto se hace mediante la llamada a *setContentView()*, independientemente de si cargamos dicha jerarquía de un fichero XML (lo más aconsejable) o de si la definimos en el propio código Java. Además, también deberemos inicializar los datos y llevar a cabo las operaciones cuyos resultados se mantengan estáticos durante toda la ejecución de la actividad.

- *onStart()*: Esta operación será llamada por el S.O. cuando la actividad vaya a pasar a ser visible para el usuario, antes de hacerlo.

  En ella deberemos llevar a cabo los preparativos para una visualización concreta de la actividad. Preparativos que se deberán hacer cada vez que se visualice la actividad de nuevo, pues en caso contrario se habrían llevado a cabo en *onCreate()*.

- *onResume()*: Esta operación la llama el S.O. justo después de que la actividad pase a ser visible para el usuario y hasta que no se acabe su ejecución la interacción con el usuario no podrá llevarse a cabo. Se suele utilizar para activar animaciones o efectos que deben empezar justo cuando el usuario ya está viendo la actividad.

- *onPause()*: Otra de las operaciones más importantes en el ciclo de vida de una actividad. Esta operación es llamada por el S.O. cuando la actividad va a pausarse, pues otra actividad va a pasar a ejecutarse en primer plano.

  Se trata de la última operación que el S.O. Android nos asegura que será llamada. Y es que, una vez ejecutada esta operación, por falta de memoria principal el S.O. puede matar la actividad antes o después de ejecutar *onStop()*. Por este motivo, en *onPause()* deberemos hacer persistentes los datos que aún no hayamos guardado.

  Además, en esta operación también se suelen pausar animaciones u operaciones que se estén ejecutando en otro *thread* y consuman mucha CPU, pues cuando la actividad se pausa probablemente estas operaciones ya no es necesario que se estén ejecutando.

  En cualquier caso, las tareas de *onPause()* se deben llevar a cabo rápidamente, pues hasta que no acabe su ejecución la actividad no podrá dejar de ser visible y, por tanto, la nueva actividad que vaya a ser presentada no podrá visualizarse.

- *onStop()*: El S.O. llama a esta operación cuando la actividad acaba de ser retirada de la vista del usuario. Aquí se suelen llevar a cabo acciones que liberan los recursos del dispositivo y que mientras la actividad estaba visible no se podían llevar a cabo, ya sea porque habrían tardado mucho en realizarse o porque el usuario las habría visto.

  El S.O. puede matar la actividad antes, durante o después de esta operación si necesita liberar memoria principal.

- *onRestart()*: Esta operación se llama cuando la actividad, que permanece invisible, va a volver a visualizarse. En ella se pueden llevar a cabo preparativos que únicamente sea necesario realizar cuando la actividad pasa a ser visible después de una pausa anterior. Ver Figura 2.4 para más detalles acerca de las operaciones llamadas posteriormente a esta.

  Una vez llamada la operación *onRestart()*, la actividad nuevamente no podrá ser eliminada por el S.O. hasta después de que se vuelva a llamar a *onPause()*, pues el S.O. da la prioridad más alta a la actividad visible en cada momento.

- *onDestroy()*: Esta operación será llamada por el S.O. cuando la instancia de la actividad en cuestión vaya a ser eliminada, antes de llevar a cabo la eliminación. En ella se deben realizar las acciones de liberación de recursos necesarias.

  Esta operación solo será llamada si la actividad va a ser eliminada por indicación explicita del desarrollador de la aplicación, a causa de que el usuario haya presionado el botón Back o debido a una necesidad de recursos leve por parte del S.O. Pero si, por el contrario, el S.O. necesita memoria rápidamente, este eliminará el proceso entero donde se esté ejecutando la actividad sin llamar a *onDestroy()*, aunque tampoco hará falta, pues al eliminar el proceso se liberarán todos los recursos automáticamente.

Y hasta aquí las operaciones a implementar para gestionar el ciclo de vida de una actividad. Nótese que no todas tienen por qué redefinirse en la subclase de *Activity* que creemos, solo las operaciones en las que nos sea necesario llevar a cabo algunas acciones. Para el resto de operaciones dejaremos el comportamiento por defecto implementado en la superclase *Activity*.

2.2. La ejecución de una aplicación

En Android, las aplicaciones se ejecutan cada una en su propia instancia de la máquina Dalvik. La máquina Dalvik es una máquina virtual y su trabajo es interpretar, en tiempo real, el código en el que están escritas las aplicaciones para Android, transformándolo en instrucciones que el procesador del dispositivo entienda. La decisión de utilizar una máquina virtual que en tiempo real interpreta el código se debe a la necesidad de que Android sea un ecosistema que se pueda utilizar en múltiples dispositivos diferentes, con arquitectura diferente.

Para conseguir un código multiplataforma, en el desarrollo de aplicaciones para Android se utiliza el lenguaje Java [6], que también es un lenguaje interpretado durante la ejecución. Gracias a la utilización de este lenguaje el hecho de que una aplicación funcione en un determinado dispositivo dependerá únicamente de que la máquina virtual, que interpreta el código en tiempo real y que viene incluida con el paquete de software Android, este adaptada a la arquitectura del dispositivo. Evitamos así tener que compilar el código para cada arquitectura diferente y tan solo debemos compilarlo una única vez para que este se transforme en un código compatible con la máquina Dalvik.

Esta máquina virtual esta especialmente optimizada para que se puedan ejecutar varias instancias de ella al mismo tiempo. Y es que Android es un S.O. multitarea y varias aplicaciones pueden estar ejecutándose a la vez. En este caso, cada una de ellas utilizará una instancia diferente de la máquina Dalvik. Además, entre diferentes instancias de la máquina Dalvik no puede haber una comunicación directa, pues así se consigue mayor seguridad en el ecosistema.

En la Figura 2.5 podemos ver un ejemplo de las múltiples instancias de la máquina Dalvik que pueden estar activas en un momento dado de la ejecución del ecosistema Android.

En el ejemplo, podemos apreciar como una instancia está ocupada por una aplicación que hace la función de cliente de correo electrónico. Esta aplicación tiene, entre otras, dos instancias de componentes en ejecución: un servicio que va comprobando si se han recibido nuevos mensajes para alertar al usuario y una actividad que presenta al usuario la pantalla de redacción de un correo electrónico. Esta última actividad, la ha lanzado el navegador de Internet que se está ejecutando en la segunda instancia de la máquina Dalvik.

En esta segunda instancia de la máquina virtual podemos ver dos instancias de la actividad Pestaña, las cuales representan a dos pestañas independientes del navegador. Una de ellas debe haber accedido a una web con un correo electrónico al

que el usuario se ha decidido a enviar un mensaje, de ahí que haya lanzado la actividad de la aplicación de correo electrónico correspondiente a la redacción de un correo.

Por tanto, podemos concluir que dentro de cada instancia de la máquina Dalvik se encuentran las diferentes instancias de los componentes de una aplicación determinada que se han creado en un momento dado.

Figura 2.5: Ejemplo de las instancias de la máquina Dalvik activas durante la ejecución del sistema Android.

Por defecto y a menos que queramos lo contrario, todos los componentes de una instancia de la máquina Dalvik se ejecutan en el mismo proceso.

Y en lo que respecta a los *threads* o hilos de ejecución, en principio, todas las instancias de componente se ejecutan también en un único *thread*, llamado UI Thread. El UI Thread tiene prioridad ante el resto de *threads* que pueden crearse en un proceso de una instancia de la máquina Dalvik, pues se trata del *thread* que recibe las peticiones del usuario y del S.O., así que contra más rápido se reaccione a estas peticiones más positiva será la experiencia de usuario.

Por este motivo debemos evitar que el UI Thread realice demasiado trabajo y, para conseguirlo, cuando una petición del S.O. requiera una serie de operaciones costosas deberemos crear otro *thread* (también llamado Worker Thread) donde llevarlas a cabo. De igual forma, si creamos un servicio, lo aconsejable es que este se ejecute en su propio *thread*, en lugar de en el UI Thread.

2.3. Una aplicación desde el punto de vista del usuario

Ya hemos visto que el S.O. Android crea una instancia de la máquina Dalvik para cada una de las aplicaciones instaladas, pero esta solo es una visión interna de lo que ocurre en el S.O. De cara al usuario existe otra visión, donde no tratamos con aplicaciones diferenciadas, sino con **tareas** diferentes.

Y es que, aunque entre dos instancias de la máquina Dalvik no puede haber comunicación directa, lo que sí que se puede hacer es pedir al S.O. que cree una instancia de un componente de otra aplicación. A esta instancia de componente le podremos pasar los datos de entrada que solicite, pero se estará ejecutando en otra instancia de la máquina Dalvik, así que la comunicación se reducirá a esos datos de entrada. En el ejemplo anterior (Figura 2.5) teníamos un caso en el que una aplicación solicita la creación de una instancia de actividad de otra aplicación.

En caso de que los componentes de otras aplicaciones utilizados sean servicios, el usuario puede ni enterarse de que estos se están ejecutando. Pero, en caso de que sean actividades, estas se le mostrarán al usuario en pantalla y, por tanto, el usuario tendrá la sensación de que estas pantallas son de la misma aplicación que está ejecutando, cuando resulta que son de otra. El hecho de que todas las aplicaciones sean una y sus componentes se compartan entre diferentes aplicaciones es una de las características que Android intenta potenciar.

De aquí surge el concepto de tarea, que viene a referirse a la secuencia de actividades, de una o muchas aplicaciones, que una detrás de otra se han ido ejecutando hasta llegar a parar a la última actividad mostrada en pantalla. Siempre una tarea parte de la actividad *main* de la aplicación que el usuario está ejecutando. La actividad *main* es una actividad que tienen todas las aplicaciones y que es la primera que se ejecuta cuando el usuario presiona sobre el icono de la aplicación en cuestión. La última actividad o pantalla ejecutada de una tarea puede estar visible o no, dependiendo de si en un momento dado la tarea ha sido pausada por tal de retomar otras tareas anteriores o no.

Por tanto, una tarea sería algo así como una ejecución puntual de una aplicación, pero refiriéndonos con aplicación al concepto que tiene el usuario, que puede implicar componentes de diferentes aplicaciones.

Las actividades que preceden a otras en una tarea se van almacenando en una pila de actividades (ver Figura 2.6), de forma que cada tarea tiene su propia pila independiente. La necesidad de apilar actividades se debe a que cuando se navega a través de una aplicación, el usuario puede querer volver hacia atrás en la navegación para recuperar una actividad anterior, lo que requiere que se guarden las actividades anteriores.

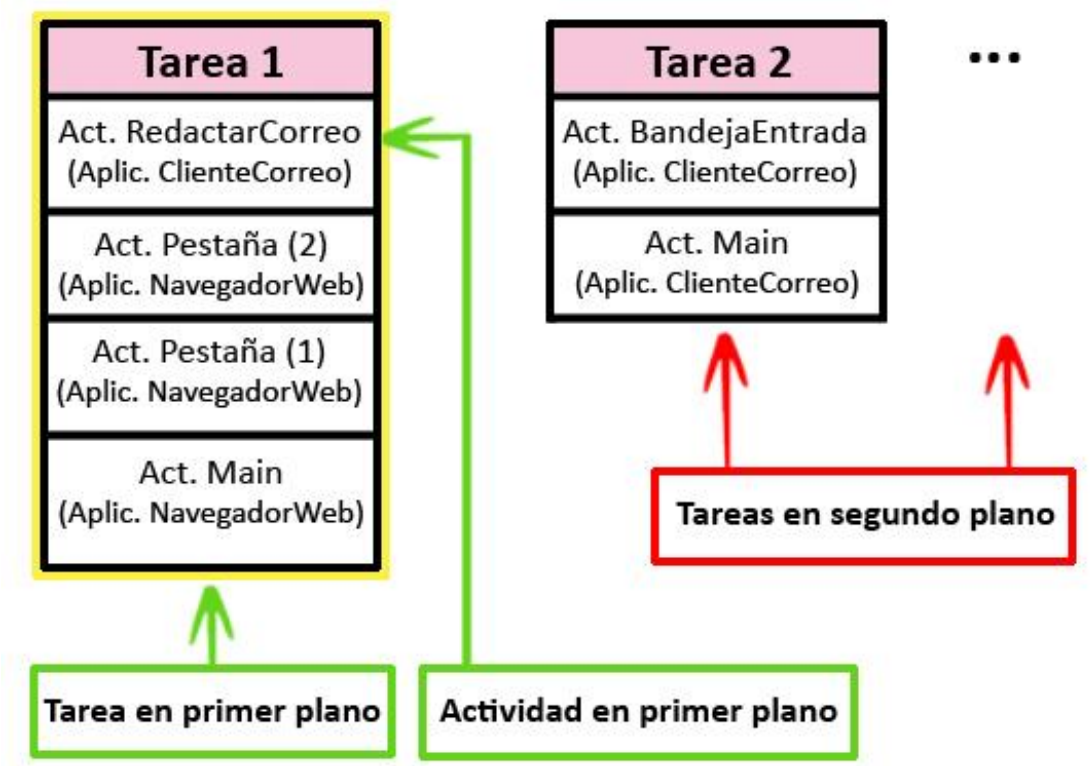


Figura 2.6: Ejemplo de tareas, con sus pilas de actividades correspondientes, que se encuentran en memoria principal durante una ejecución dada del sistema Android.

En un momento dado de la ejecución del sistema Android solo puede haber una actividad en primer plano interactuando con el usuario. Lo que significa que también habrá solo una tarea en primer plano, será la que tiene una pila cuya primera actividad es la actividad en primer plano, como podemos ver en la Figura 2.6.

Cuando una actividad se pausa, se inserta en la pila que le corresponde y pasa a no consumir prácticamente recursos, pues el UI Thread donde se estaba ejecutando pasa a tener una prioridad de ejecución muy pequeña o a ocuparse de la ejecución de otra actividad, según el caso. En este último caso, si la actividad anterior sigue solicitando gran cantidad de recursos para realizar operaciones provocaría problemas, pues la interacción con el usuario se vería ralentizada.

También es importante tener en cuenta que en esta pila de actividades puede haber varias instancias de un mismo componente si el diseñador así lo ha preparado.

Además, nótese que estas pilas conviven con el concepto de instancias de la máquina Dalvik explicado en el apartado anterior. La pila de actividades simplemente es otra visión implementada por otro mecanismo. Visión que coincide con el concepto de aplicación que tiene el usuario con conocimientos básicos, nosotros en cambio a esto le llamamos tarea.

Para vaciar la pila tenemos la tecla Back, que lo que hace es matar la actividad que está en ejecución en un momento dado. Cuando esto ocurre el S.O. rescata la actividad anterior, que se encuentra en la pila de la tarea que se está reproduciendo, y la presenta al usuario.

Además, el desarrollador puede matar actividades cuando lo desee, pues a veces, una actividad debe morir independientemente de que el usuario presione la tecla Back o no.

Por ejemplo, a la hora de finalizar un nivel de un videojuego la actividad que lo reproduce debe morir después de crear la actividad que presenta el mensaje de "Nivel Completado", pues a ojos del usuario el nivel ha acabado y, por tanto, la actividad en cuestión no pinta nada en la pila. En esta pila deberán encontrarse las actividades que implementan los menús previos visualizados por el usuario, en el orden inverso a como se han mostrado por pantalla.

Por último, una tarea o, lo que es lo mismo, la ejecución puntual de una aplicación se puede pausar en cualquier momento si el usuario presiona la tecla Home. En este caso, la actividad visible se guardará también en la pila de actividades de la tarea correspondiente y permanecerá allí hasta que volvamos a acudir a la aplicación cuya actividad *Main* conforma la raíz de esta tarea; o hasta que el S.O. requiera memoria principal, lo que provocará que la tarea sea eliminada.

Si el S.O. elimina una tarea, en la mayoría de los casos, a la hora de querer recuperarla este será capaz de hacerlo, siempre y cuando, el desarrollador haya preparado las actividades para ello [4].

3. Interacción entre el usuario y las aplicaciones

Cuando el usuario de un dispositivo con Android se encuentra ejecutando una aplicación, este lleva a cabo la interacción con la aplicación que el desarrollador le haya preparado.

No obstante hay una serie de botones que todo dispositivo con el S.O. tiene y que llevan a cabo unas acciones concretas dentro de todas las aplicaciones. Su papel dentro de estas aplicaciones puede alterarse, pero normalmente no se hace, pues la idea es que dichos botones tengan siempre la misma función, para mejorar así la usabilidad del sistema.

Figura 2.7: Botones de navegación comunes a todos los dispositivos que implementan el ecosistema Android.

Salvo el botón Search que es prescindible y algunos dispositivos no lo implementan.

En concreto son cuatro los botones (ver Figura 2.7) y en los dos productos desarrollados en este proyecto se han respetado sus funciones.

En primer lugar, tenemos el botón Menu. Este botón lo que hace es activar el llamado menú principal de la aplicación. Las aplicaciones que necesiten un menú de configuración harán bien en adaptarse e implementar un menú que se muestre al presionar este botón. En nuestro caso, las aplicaciones no tienen menú, por tanto el botón no tiene efecto alguno.

En segundo lugar, tenemos la tecla Home. Al presionar esta tecla durante la ejecución de una tarea pausa la actividad actual (mediante la llamada a *onPause()* vista con anterioridad) y la introduce a la cabeza de la pila de actividades que representa a la tarea en ejecución. Seguidamente, nos envía al escritorio de Android. En caso de querer retomar la tarea anterior, la cual ahora permanece pausada y en segundo plano, deberemos presionar el botón de la aplicación cuya actividad Main constituye la primera actividad de la tarea dada. De esta forma volveremos a recuperar la tarea por el punto donde la habíamos dejado.

En tercer lugar, tenemos el botón Back, este nos permite matar la actividad actual y dar paso a la actividad anterior, la cual se encuentra en la pila de actividades que conforman la tarea que estamos ejecutando en el momento de presionar el botón. La actividad que muere lo hace llamando a la operación *onDestroy()* como hemos visto en el apartado anterior. En resumen, mediante esta tecla, estamos retrocediendo en la pila de actividades.

Por último, algunos terminales tienen un botón con forma de lupa, pero este simplemente supone un acceso directo a la búsqueda del dispositivo, así que no forma parte del mecanismo de interacción común a todas las aplicaciones.

4. Entorno de desarrollo de Android

A continuación se va a describir el entorno de desarrollo que Android pone a disposición de los desarrolladores. Hay que destacar que una de las ventajas de este entorno es que se puede instalar en Windows, en Linux y en Mac O.S. X.

Lo único que necesitamos para desarrollar una aplicación es el SDK (Software Development Kit) de Android. Un kit que incluye una serie de herramientas como son: el Traceview, herramienta que nos permite extraer estadísticas de rendimiento de la aplicación; el LogCat, que nos presenta los mensajes que se imprimen desde el código durante la ejecución de una aplicación; y herramientas para generar los instaladores de las aplicaciones que desarrollemos; entre otras herramientas de utilidad.

Además, el SDK incluye la API (Application Programming Interface) de Android, unas librerías que contienen todas las clases y operaciones que debemos utilizar para poder comunicarnos con el S.O. y de esta forma poder, por ejemplo, definir una actividad.

En cuanto al IDE (Integrated Development Environment) o entorno que integra las herramientas necesarias para desarrollar aplicaciones, podemos utilizar cualquiera. Incluso podríamos utilizar un editor de textos normal para programar la aplicación y no utilizar un IDE. Aun así, se recomiendo encarecidamente el uso de Eclipse. Y es que existe un *plugin* oficial llamado ADT (Android Development Tools) que una vez instalado en Eclipse nos permite acceder a la API y a las diferentes herramientas que el SDK nos blinda directamente desde la interface de Eclipse, facilitando enormemente el desarrollo de las aplicaciones.

Si utilizamos Eclipse, por tanto, podemos visualizar el LogCat, controlar el Traceview o generar el fichero .apk (el instalador) resultante de la aplicación, desde la propia interface de Eclipse.

Por último, en cuanto al lenguaje de programación, la API se encuentra en Java y el lenguaje recomendado para programar es este. Java es un lenguaje menos eficiente que C y sus variantes, por lo tanto no se suele utilizar para desarrollar videojuegos. Pero también tiene una gran ventaja y es que, como se ha comentado con anterioridad, es un lenguaje interpretado, lo que significa que se interpreta a través de una máquina virtual en tiempo real.

De esta forma podemos evitar tener que compilar nuestra aplicación para diferentes arquitecturas, pues con una sola compilación, la aplicación funcionará en todos los dispositivos sin problemas.

Sin embargo, el uso de C sigue siendo imprescindible para desarrollar con éxito algunas partes de un videojuego complejo, como la gestión de físicas. Así que a través de un paquete llamado NDK (Native Development Kit) podemos utilizar funciones programadas en C, que se ejecutarán de forma nativa durante la ejecución de la aplicación. Claro está que, al ser C un lenguaje no interpretado, deberemos compilar este código para cada una de las arquitecturas donde la aplicación de Android vaya a ejecutarse. Para facilitar las cosas, Android nos permite introducir en el mismo instalador de la aplicación varias compilaciones diferentes de las operaciones en C, por tal de que el mismo instalador funcione en varias arquitecturas al mismo tiempo.

Nótese que cada nueva versión de Android que sale a la luz amplía el número de acciones que podemos llevar a cabo utilizando únicamente el lenguaje C. Se espera que en un futuro se puedan llegar a desarrollar aplicaciones únicamente haciendo uso de este lenguaje.

Pero en este proyecto se ha trabajado con versiones del S.O. que no permiten realizar demasiadas acciones en C, así que se han desarrollado las aplicaciones haciendo uso únicamente del lenguaje Java. Aunque, para obtener la eficiencia que aporta una gestión de las físicas del videojuego en lenguaje C, se ha utilizado una librería de físicas desarrollada en este lenguaje. Eso sí, dicha librería esta oculta por un mapeo entre sus operaciones y el lenguaje Java, con lo cual, aunque realmente haya una parte del videojuego funcionando en C, desde el punto de vista del desarrollador solo se ha trabajado en Java.

5. El problema de la fragmentación

Ya hemos comentado algunas de las ventajas que tiene desarrollar para un sistema operativo abierto como es Android. Pero realmente, el hecho de que sea abierto no genera solo ventajas, también conlleva una desventaja muy importante. El problema radica en que Android se va actualizando periódicamente, pero como este ecosistema se implementa en muchos otros dispositivos, la actualización de estos a la última versión del sistema, depende de los fabricantes.

La cadena es la siguiente. Cuando Google libera una versión nueva de Android, los fabricantes de cada dispositivo, tienen que adaptar sus *drivers* a esta versión y, si lo desean, desarrollar una interface visual para la nueva versión del ecosistema, cosa que suelen hacer. Seguidamente y antes de mandar la actualización a los terminales, si estos móviles no son libres, las actualizaciones pasan por manos de las operadoras de telefonía móvil, las cuales se encargan de introducir una serie de aplicaciones que después el usuario no podrá borrar.

Todo este proceso provoca que las actualizaciones lleguen a los terminales móviles muy tarde, a veces pasan muchos meses hasta su llegada. Pero el problema va más allá, porque la mayoría de los terminales no tienen la suerte de actualizarse a la última versión. Como el mercado de teléfonos móviles inteligentes evoluciona tan rápidamente, los fabricantes se olvidan muy fácilmente de los modelos ya puestos a la venta y muchos se quedan sin estas actualizaciones.

Esto provoca que, de cara al desarrollador, de aplicaciones se presente una fragmentación entre versiones bastante compleja. En Android normalmente una aplicación desarrollada para una versión va a ser compatible con todas las futuras versiones, pero no es así con las versiones más antiguas. Y es que la API va evolucionando y va incluyendo nuevas operaciones que facilitan las cosas y que no están soportadas en anteriores versiones de la misma.

Por tanto, a la hora de desarrollar, a menos que queramos hacer una aplicación muy a largo plazo, no podremos ni mucho menos utilizar la API más moderna y cómoda, sino que tendremos que conformarnos con una antigua. Pues en caso de no hacerlo nuestra aplicación llegaría a muy pocos usuarios.

Para hacernos una idea, la versión 2.3 de Android, salió a la luz en Diciembre de 2010. En mayo de 2011, fue cuando se llevó a cabo el análisis de antecedentes que veremos en el apartado 7 de este mismo capítulo, y la versión 2.3 tan solo suponía el 4% de usuarios de Android, según la propia Google. A Enero de 2012 la versión 2.3 de Android es la que utilizan el 55.5% de los usuarios de Android, como vemos son la mayoría pero no una mayoría demasiado grande.

Así que podemos concluir que la falta de exigencia de Google sobre las empresas que implementan su ecosistema provoca que el ecosistema evolucione más lentamente de cómo debería hacerlo. Cabe decir que aunque hay una serie de desarrolladores que se dedican a sacar actualizaciones no oficiales para diferentes dispositivos, al tratarse de personas que lo hacen como hobby, no suelen ser perfectas y pueden provocar la pérdida de algunas funcionalidades del dispositivo; por tanto, esta no es una solución al problema.

A continuación se presenta una tabla con las versiones principales del S.O. publicadas hasta la fecha y las mejoras más importantes que estás han aportado de cara al desarrollador de videojuegos:

Versión de Android	Mejoras añadidas	Fecha de Salida
Android 1.0 y 1.1	-	27/09/2008
Android 1.5 (Cupcake)	Se añade la *view GLSurfaceView* para facilitar la creación de aplicaciones que utilicen OpenGL ES. (Veremos su utilidad en el apartado 6 de este mismo capítulo)	30/04/2009
Android 1.6 (Donut)	No hay añadidos importantes que afecten al desarrollo de videojuegos de forma especial.	15/09/2009
Android 2.0 y 2.1 (Eclair)	Se añade soporte para pantallas *multitouch*. Que son las pantallas táctiles en las cuales podemos tener más de un punto de presión al mismo tiempo.	26/10/2009
Android 2.2 (Froyo)	Se añade soporte para OpenGL ES 2.0, versión de OpenGL ES que destaca por permitir el uso de *shaders*. Hasta ahora solo las versiones de OpenGL 1.0 y 1.1 estaban soportadas.	20/05/2010
Android 2.3, 2.3.3, … (Gingerbread)	Esta versión ofrece múltiples ventajas para el desarrollo de videojuegos, algunas de ellas son: - Mejora en la forma en que el S.O. trata los eventos de entrada, por tal de ganar eficiencia en videojuegos. - Mejora en el recolector de basura. - Mejora del NDK, haciéndolo más robusto. - Y la mejora más importante, ahora se puede gestionar directamente a través del lenguaje nativo C: el audio, los gráficos, el ciclo de vida de las actividades, el acceso a los recursos, etc.	06/12/2010
Android 3.0 y 3.1 (Honeycomb)	Esta versión está dirigida a *tablets*. Su principal añadido es la aceleración por hardware de los elementos de la interface de usuario, lo que provoca que esta vaya más fluida.	22/02/2011
Android 4.0 (Ice Cream Sandwich)	La versión más reciente de Android, une la última versión para *tablets* (3.0) y la última versión para *smartphones* (2.3) en una sola. Conserva la aceleración por hardware en las interfaces de usuario. Y en cuanto al desarrollo de videojuegos, añade una nueva versión de *GLSurfaceView* más potente, llamada *TextureView*.	19/10/2011

Figura 2.8: Tabla con las diferentes versiones de Android publicadas hasta la fecha. Se detallan las mejoras más importantes en el desarrollo de videojuegos que cada versión ha aportado, así como la fecha de publicación del SDK de cada una de las versiones.

Un dato curioso que podemos ver en la tabla es que, desde Android 1.5, cada versión ha sido bautizada con el nombre de un dulce.

Por último, decir que, en la Figura 2.8, hay algunas versiones que aparecen en un mismo recuadro, esto es debido a que en realidad no son diferentes versiones, sino revisiones de la misma versión que solucionan algunos errores. En estos casos la fecha de salida indicada corresponde a la fecha de publicación del SDK de la versión en sí (no de las revisiones).

6. Estructura de un videojuego

En este apartado vamos a detallar cómo es la estructura que todo videojuego debe implementar, para después explicar cómo se implementa esta en Android, puesto que hay una serie de cambios provocados por las peculiaridades de este S.O.

Cuando estamos ejecutando un nivel de un videojuego lo que realmente estamos haciendo es simular un universo, el universo formado por los elementos que conforman dicho nivel: personaje principal, enemigos, escenario, etc.

Debemos hacer creer al usuario que ese universo tiene vida propia, que sus elementos se mueven e interaccionan. Para llevar a cabo esta simulación, lo que realmente hacemos es discretizar el tiempo de este universo en diferentes *frames*, nombre por el cual se conoce a las diferentes imágenes que componen una animación. Es decir, cada cierto tiempo hacemos una foto del universo del videojuego y se la presentamos al usuario. Si hacemos suficientes fotos por segundo y las vamos mostrando rápidamente, la vista humana tendrá la sensación de movimiento y, por tanto, el usuario tendrá la sensación de que ese universo tiene vida.

Como el universo en cuestión no existe, no podemos hacer fotos, así que lo que se hace es calcular, para cada *frame*, la posición y el estado de cada elemento en el instante de tiempo que queremos representar. Una vez calculados estos datos, se pintan las texturas que representan a los diferentes elementos del escenario sobre la geometría que conformará la imagen resultante, que al tratarse de un juego en 2D es simplemente un plano. Fruto de este proceso sale lo que llamamos *render*, que no es más que otra forma de llamar a esta imagen resultante, cuando es calculada mediante un proceso de *renderizado* [7], como el que lleva a cabo OpenGL.

Para que una sucesión de imágenes sea vista por el ojo humano como una animación y no se aprecien ralentizaciones o trompicones, se deben presentar al menos 30 *frames* cada segundo. A pesar de ello, hasta los 60 *frames* por segundo el ojo humano notará la diferencia. Por tanto, el usuario verá más fluida y real una animación que está presentando 60 *frames* por segundo, que una que este presentando tan solo 30.

En cambio, si superamos la tasa de 60 *frames* ya es más difícil notarlo, quizás en objetos que se mueven muy rápido se pueda llegar a notar, pero realmente no es necesario ofrecer más de 60 *frames* por segundo.

En los videojuegos desarrollados a lo largo de este proyecto se ha decidió marcar como requisito el que los videojuegos funcionen a 30 *frames* por segundo, pues al estar trabajando con dispositivos móviles, el hardware de estos es limitado, con lo cual puede llegar a ser difícil conseguir 60 *frames* por segundo estables. Y hay que decir que es preferible una tasa de *frames* más baja y estable, que una más alta pero inestable, pues el hecho de que la tasa de *frames* vaya cambiando durante la ejecución del videojuego acaba siendo mucho más molesto.

6.1. Sincronización del videojuego

Ya hemos visto cómo vamos a simular un universo con vida durante la ejecución de cada nivel del videojuego. Pero, a la hora de generar los diferentes *frames* que componen esta simulación, es preciso llevar a cabo una sincronización entre ellos, de forma que sepamos en qué momento de la simulación nos encontramos y podamos calcular la posición y el estado de los diferentes elementos que componen el nivel en dicho momento.

Existen, al menos, dos formas de llevar a cabo esta sincronización. El tipo de sincronización que se ha utilizado en los videojuegos implementados es la **sincronización por *framerate***. Con este tipo de sincronización cada *frame* tiene la misma duración. De forma que si queremos conseguir una tasa de 30 *frames* por segundo, deberemos presentarle al usuario un *frame* nuevo cada 33.3 milisegundos.

Al utilizar este método, sabemos que, para cada *frame* nuevo que creemos, debemos simular que en el videojuego han pasado 33.3 milisegundo. Así que partiendo de las últimas posiciones y estados de cada elemento, deberemos calcular su posición y el estado de cada elemento 33.3 milisegundos después.

Claro está, si utilizamos este tipo de sincronización debemos intentar que cada *frame* se pueda calcular en tan solo 33.3 milisegundos. Ya que, en caso de que un *frame* tarde más, el usuario tendrá la sensación de que los elementos se mueven más lentamente de los normal, pues estaremos presentando una foto del universo 33.3 milisegundos después cuando habrá pasado más tiempo.

Si, por el contrario, tardamos menos en calcular el *frame*, deberemos esperar a que transcurra el tiempo completo antes de presentarlo al usuario, pues en caso contrario, el usuario tendría la sensación de que los elementos se mueven más rápidamente de la cuenta.

El segundo método de sincronización, el cual se ha descartado, es la **sincronización por tiempo**. Con este tipo de sincronización, cada *frame* tiene la duración que sea necesaria, pues sincronizamos los elementos del juego con respecto al tiempo real transcurrido desde el inicio de la ejecución del videojuego.

La desventaja principal de este tipo de sincronización es que ante un procesado lento de los frames, el juego no se ralentiza, sino que los elementos del videojuego dan saltos espaciales a las posiciones que deban ocupar en el momento de generar el *frame*. Lo que puede llevar a que un procesado lento de los *frames* sea la causa de que el personaje pierda una vida, por ejemplo, ya que el juego puede dar un salto y el personaje principal puede ir a parar a un lugar donde haya un enemigo sin que el usuario tenga tiempo a reaccionar y esquivarlo.

Por tanto, este segundo método es útil para videojuegos online, pues en este caso los elementos del escenario están siendo compartidos entre varios usuarios y la lentitud en el procesado de uno de los computadores implicados no debe afectar a todos los usuarios. Así, el usuario que sufra el problema de procesamiento verá que los elementos van dando saltos espaciales de un lugar al otro, pero el resto de usuarios no tendrá problema en ver todo el recorrido que hagan dichos elementos.

En cambio para un juego offline lo mejor es que un mal procesamiento no afecte negativamente al usuario (restándole vida o de cualquier otra forma), así que por este motivo empleamos la sincronización por *framerate*.

6.2. El bucle del videojuego

En este apartado se van a detallar los pasos que debemos llevar acaba para generar cada uno de los *frames* de un nivel del videojuego. Y es que, una vez superada la fase de **inicialización**, en la que se inicializan los diferentes elementos del nivel, entramos en un bucle que durará hasta que la simulación del nivel finalice (ver Figura 2.9).

Cada vez que vamos a generar un *frame*, lo primero que tenemos que hacer es leer la **entrada** del usuario, por si este ha llevado a cabo alguna acción a la que debamos dar respuesta.

Seguidamente entramos en la fase en la que se **procesa** el *frame*. Dentro de esta fase, en primer lugar, calcularemos la posición física de todos los elementos del escenario en el instante de tiempo que queramos representar a través del *frame*. Para ello tendremos que aplicar la entrada del usuario sobre el personaje principal, la Inteligencia Artificial sobre los elementos que estén dotados con ella y las leyes de la física sobre todos los elementos que estén sometidos a ellas.

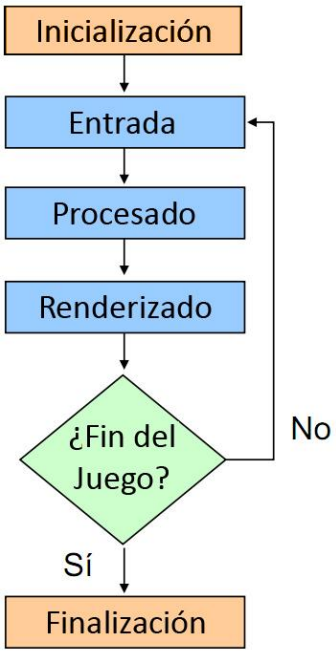


Figura 2.9: Estructura de un videojuego.

Seguidamente llega el momento de procesar las lógicas de los diferentes elementos. La lógica viene a ser el estado de cada elemento, e incluye: el número de vidas, su estado de ánimo, etc. De forma que, si por ejemplo, después de procesar las físicas vemos que hay dos elementos que están colisionando, durante el procesado de las lógicas seguramente deberemos restar una vida a alguno de estos dos elementos.

Es a través de la lógica, es decir, del estado de los diferentes elementos del videojuegos, que sabemos si los objetivos del nivel en ejecución se han cumplido o no y, por tanto, si el nivel se ha completado o si el usuario ha fracasado.

Una vez acabada la fase de procesamiento, llega el ***renderizado***. Durante esta fase, se cogen las posiciones y el estado de los diferentes elementos que componen el nivel, calculados en la fase anterior, y se genera el *render* resultante de este *frame*. Es decir, se genera la foto del universo del videojuego, con todos los elementos en la posición que les corresponde y con el estado en el que se encuentra cada uno (el cual puede afectar a las texturas con las que se representa el elemento).

Nótese que después de esta última fase, si llevamos a cabo una sincronización por *framerate*, es cuando deberemos esperar a que pase el tiempo que debe durar cada *frame* antes de presentarle el resultado al usuario. Si, por el contrario, el tiempo pasado desde que se comenzó la fase de "Entrada", en esta iteración del bucle, es mayor al tiempo que debe durar el *frame*, habremos llegado tarde y, por tanto, el usuario puede llegar a apreciar alguna ralentización.

El último paso de la iteración consiste en comprobar si se han satisfecho los objetivos que se deben cumplir para completar el nivel o para fracasar el nivel. Sí el nivel ni ha fracasado ni se ha completado, volveremos a realizar otra iteración del bucle. En caso contrario, pasaremos a ejecutar la última fase de la estructura, la fase de **Finalización**. En esta fase se liberan los recursos que han sido necesarios durante la simulación del nivel y se guarda el avance del usuario si es necesario.

6.3. El bucle del videojuego en Android

Ya hemos visto cómo es la estructura utilizada para ejecutar un nivel de un videojuego. Pero, en Android, cada pantalla de una aplicación es una actividad y las interfaces de estas son un árbol de *views*.

En este apartado vamos a ver cómo integrar la estructura de una aplicación para Android y la estructura de un videojuego. Es decir, cual es la estructura de un videojuego en Android.

Para presentar el *render* resultante del procesado del *frame* y, en concreto de la fase de *renderización*, existe una *view* especial llamada *GLSurfaceView*. Esta *view*, como todas las otras, gestiona un rectángulo del área ocupada por la actividad independientemente. Por tanto, cuando queramos implementar una actividad que vaya a presentar los resultados de la ejecución de un nivel del videojuego, la jerarquía que contiene la interface de esta actividad estará únicamente formada por una instancia de la *view GLSurfaceView*.

En concreto, lo que normalmente se hace es heredar de *GLSurfaceView*, para crear una subclase propia que personalice la forma de presentar el *render*. El funcionamiento de *GLSurfaceView* requiere que le introduzcamos a la instancia de esta clase una instancia de una subclase de la clase *Renderer* implementada para la ocasión. [8]

Esta subclase de *Renderer* debe implementar tres operaciones ya declaradas en la superclase. Se trata de operaciones que el S.O. llamará cuando se den una serie de condiciones, entre las que se incluye querer crear un *render*. Además, cuando el S.O. llame a estas operaciones lo hará a través de un thread especial llamado GL Thread, pues como ya hemos visto, si queremos que la interacción con el usuario sea cómoda para este, el UI Thread donde se ejecuta la actividad debe permanecer libre.

En concreto las tres operaciones de *Renderer* son:

- *onSurfaceCreated()*: Esta operación es llamada por el S.O. cada vez que se obtiene un nuevo contexto OpenGL. EL contexto OpenGL es el objeto a través del cual se realizan todas las llamadas a OpenGL.

  Cuando se obtiene un nuevo contexto hay que realizar de nuevo todas las acciones llevadas a cabo para poder construir el *render*. Como, por ejemplo, registrar las texturas para obtener los identificadores a través de los que poder utilizarlas.

  En ella se deberían llevar a cabo todos los preparativos necesarios para utilizar el contexto OpenGL. Preparativos que afecten a todos los *renders*. Pues gracias a este método, evitaremos repetir estos preparativos para cada *render* que se cree.

  Normalmente el contexto OpenGL se pierde cuando pausamos o eliminamos la actividad que mantiene como interface la jerarquía con la instancia de *GLSurfaceView* que tiene la instancia de *Renderer* en su interior.

- *onSurfaceChaged()*: Esta operación será llamada cada vez que cambie la orientación del dispositivo. Es decir, cada vez que cambie el tamaño de la pantalla. Esto incluye una primera llamada cuando se comienza a ejecutar el nivel, para adaptarnos al tamaño por defecto de la pantalla.

  En esta llamada se debería configurar el tamaño del *render* dentro del área gestionada por la *view GLSurfaceView*, así como el tamaño del área de dentro del universo del videojuego que se va a mostrar por pantalla. Y es que, normalmente no querremos que el usuario vea todo el universo simulado, sino solo una parte de este, la parte del escenario en la que se encuentre el personaje principal.

- *onDrawFrame()*: Esta es la operación más importante y es que el S.O. la llama cada vez que se quiera construir un *render*. En ella se deben implementar las acciones pertinentes para construir el *render* que represente al *frame* que estemos acabando de generar.

  Si dejamos la configuración del *Renderer* por defecto, será el S.O. el que periódicamente llame a la operación para generar el *render* que toque. Pero también podemos configurar-lo para que, en lugar de ser el S.O. el que decida cuando llevar a cabo las llamadas a esta operación, sea el desarrollador desde el código quien lo especifique. Aunque esto no quiere decir que vayamos a conseguir una generación del *render* totalmente secuencial, pues es el S.O. el que, cuando reciba la petición, llamará a la operación, llamada que puede no ser inmediata.

Por tanto, como podemos ver, la fase de *renderización* del bucle del videojuego en este caso no se realiza secuencialmente con el resto del bucle, sino que, a causa del carácter asíncrono que las aplicaciones de Android tienen, el *render* se crea de forma concurrente a la ejecución del bucle del videojuego.

Además, el hecho de que el *render* se genere en otro *thread* nos obliga a llevar a cabo algún tipo de sincronización, para que desde el thread donde se esté llevando a cabo la fase de procesado no se modifiquen los datos hasta que no hayan sido leídos para generar el *render* correspondiente a un *frame* anterior.

Pero es que además, el resto del bucle del videojuego no puede estar en el UI Thread, pues ya sabemos que este tiene que estar libre para poder recibir los eventos del usuario. Así que lo que hacemos es crear otro thread, el llamado *thread* de juego y en este se lleva a cabo el bucle del videojuego, el cual no cesará su ejecución hasta que se pause o se acabe la simulación del nivel del videojuego.

En este bucle primero de todo se comprueba si se ha recibido una entrada nueva por parte del usuario y, posteriormente, se lleva a cabo la fase de procesado. Una vez procesado el *frame*, se espera a que este se *renderice* y a que pase el tiempo asignado al *frame* en cuestión, pues recordemos que llevamos a cabo una sincronización por *framerate*.

Por tanto, en el UI Thread solo se configura el *GLSurficeView* y se reciben las entradas del usuario, las cuales se guardarán en un buffer para que el *thread* de juego las lea cuando le toque.

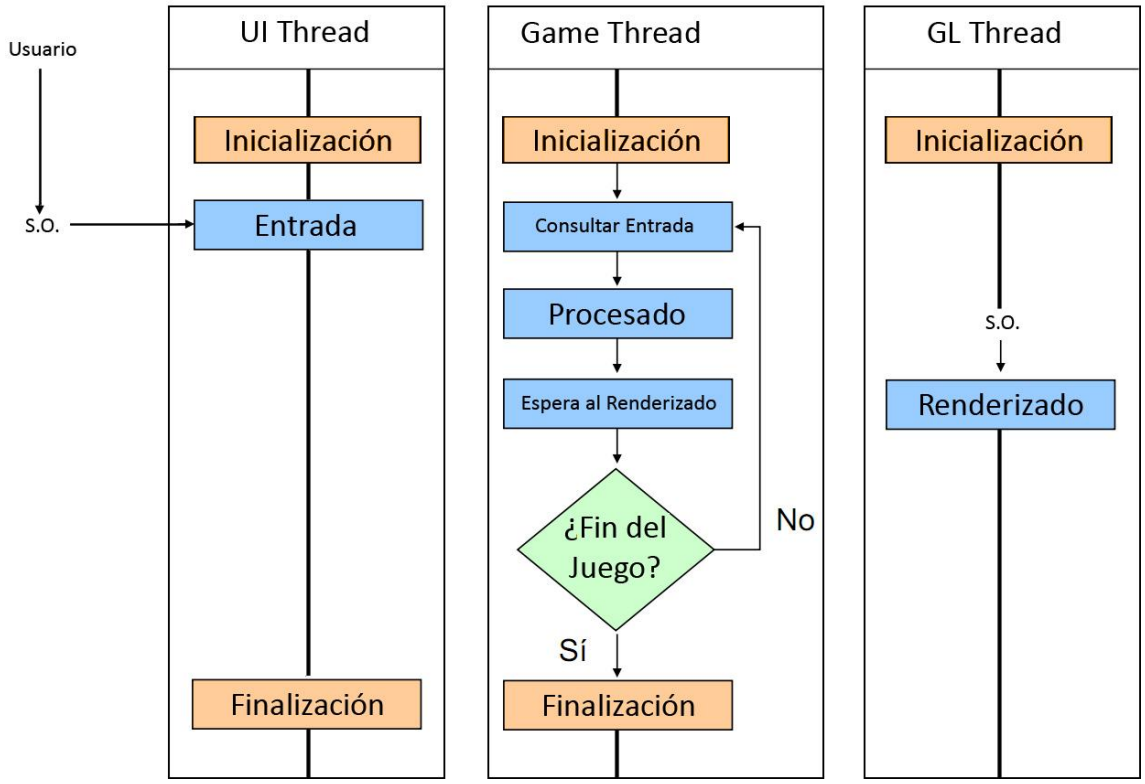


Figura 2.10: Estructura de un videojuego para Android.

En la Figura 2.10 podemos ver un esquema de la estructura que sigue un videojuego para Android. Podemos ver que hay fases que se dividen en varios *threads*, como la inicialización y la finalización. Al mismo tiempo, encontramos que el único *thread* que se ejecuta de forma secuencial, es el Game Thread, el cual lleva a cabo el bucle del videojuego visto anteriormente. Los otros dos *threads* funcionan de forma asíncrona, pues se mantienen en reposo hasta que el S.O. les llama para llevar a cabo una función determinada.

Para más información sobre esta estructura y sobre la comunicación entre los tres *threads*, los casos concretos de implementación de la estructura llevados a cabo en los dos videojuegos construidos se describen en los capítulos 3 y 4 de esta memoria.

7. Análisis de antecedentes

Cuando desarrollamos un videojuego está muy bien que este introduzca cosas nuevas que otros no tienen. Pero también es importante, sobre todo si queremos intentar ganar dinero con el proyecto, que examinemos el mercado, observando qué es lo que tiene éxito y lo que no, cuales son los mecanismos de financiación más exitosos o qué mecanismos de control son incómodos para el jugador, entre otras cosas.

Una vez analizado el mercado, podemos: tanto apostar por la formula que está teniendo más éxito actualmente y intentar mejorar los videojuegos existentes, como fijarnos en una parte del mercado que pueda estar desatendida y donde creamos que podemos tener éxito rellenando un hueco solicitado. O quizás tengamos muy claro que queremos desarrollar un tipo de juego muy concreto, pero siempre será útil ver, por ejemplo, si la compatibilidad con versiones antiguas del sistema operativo o con dispositivos antiguos tiene realmente peso en el éxito y disfrute que los usuarios puedan hacer del producto.

Con el fin de aprender del mercado de los videojuegos actual en Android y evitar así, en la medida de lo posible, cometer errores en la elaboración del proyecto que ya hayan cometido otros, se han analizado los videojuegos más exitosos disponibles para la plataforma Android.

7.1. Análisis de los videojuegos más populares para Android

Primeramente se van a describir en detalle tres juegos muy descargados en el Android Market [11], para posteriormente presentar las consideraciones extraídas del análisis de estos y otros juegos populares para la plataforma Android.

Título: Angry Birds

Figura 2.11: Captura de pantalla del juego Angry Birds.

Género: Puzles

Descripción:

Se trata del juego que corona la lista de juegos más descargados en el Market de Android a día 13 de Mayo de 2011, su éxito ha sido arrollador y su número de descargas está entre 10 y 50 millones.

El juego presenta un mecanismo de juego simple. Disponemos de unos pájaros que debemos disparar mediante un tirachinas y que deben golpear una estructura en la que se encuentran unos cerdos. El objetivo en cada pantalla es el de acabar con todos los cerdos, mediante un golpe de tirachinas directo o haciendo que la estructura se desplome encima de ellos (ver Figura 2.11). Además, cada pantalla nos puntúa según el número de pájaros que hayamos sacrificado, el número de plataformas rotas, etc.

Existe un porqué de las acciones que hacemos, que aunque simple, provee de carisma al videojuego. Los pájaros quieren vengarse de los cerdos debido a una mala acción que estos han realizado.

Por otro parte el aspecto visual está muy cuidado ofreciendo unas texturas muy coloridas.

Otro aspecto a destacar es que el juego va presentando novedades a medida que avanzan las pantallas. Ya sea presentando nuevos pájaros que al ser disparados atacan a los cerdos de forma diferentes o mediante nuevas estructuras con propiedades también diferentes.

Se trata por tanto, de un juego muy adictivo y con una historia y *jugabilidad* fácil de asimilar, ahí reside su éxito dentro de las plataformas móvil.

Controles: El juego se maneja mediante la pantalla táctil, en concreto debemos hacer un gesto de tensar el tirachinas para disparar a los pájaros con más o menos fuerza.

Beneficios: En Android se ha optado por lanzar el juego gratuito y recaudar beneficios a través de la publicidad que aparece en pantalla durante la ejecución del juego (una publicidad que no es, en absoluto, molesta para el usuario).

Pero no acaba ahí y es que el juego ya cuenta con dos continuaciones que se encuentran disponibles para descargar de forma gratuita. Estas continuaciones presentan un número de descargas de entre 10 y 50 millones por parte de una y entre 5 y 10 millones por parte de la más reciente. Parece que la estrategia tomada por parte de la compañía es la de, una vez establecida la base del juego sacar nuevas versiones añadiendo pantallas y personajes, pero conservando el mismo aspecto gráfico y el mismo motor de juego.

Versión del S.O.: La versión mínima del sistema operativo Android es la 1.6. Es decir, se ha buscado llegar a prácticamente todo el público que hoy en día tiene un móvil con Android.

Cabe decir que en un principio el juego solo funcionaba en versiones 2.2 y superiores de Android pero posteriormente fue adaptado a versiones anteriores. Por tanto, podemos concluir que en este caso compensa ajustar el juego para que funcione en versiones más antiguas de Android, por tal de conseguir que el videojuego llegue a más personas.

Por último, el videojuego ha aparecido desde en los sistemas operativos móviles más famosos, hasta en videoconsolas, pasando por S.O. de computadores personales como Microsoft Windows. Recientemente ha aparecido incluso una versión para el navegador Google Chrome.

Herramientas utilizadas: Angry Birds ha sido desarrollado utilizando el motor de físicas multiplataforma Box2D, que facilita la gestión de las físicas en juegos en dos dimensiones.

Título: Robo Defense.

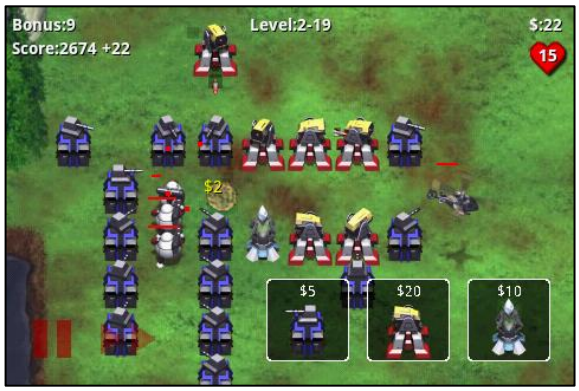


Figura 2.12: Captura de pantalla del juego Robo Defense (versión gratuita). Los personajes que tienen una barra roja en su cabeza son los enemigos y dicha barra indica la vida que les queda.

Género: Estrategia

Descripción:

Este juego copa el número uno de la lista de juegos de pago más populares del Android Market y tiene entre 500 mil y 1 millón de descargas.

Se trata de un juego de Estrategia en el que tenemos un campo de batalla a través del cual los enemigos quieren llegar a nuestra base, debemos posicionar maquinaria de guerra en sitios claves para evitar que los enemigos atraviesen el campo y lleguen a nuestra base vivos (ver Figura 2.12).

La partida se puede pausar en cualquier momento para posicionar nuevas tropas, aunque estas también pueden posicionarse sin tener el juego pausado, además, cabe decir que cuando acabamos con enemigos ganamos dinero que podremos utilizar para mejorar o ampliar nuestro armamento. Cada enemigo que atraviesa el campo nos quita una vida, con lo cual, el objetivo del juego es llevar a cabo unos objetivos dados, evitando que los enemigos que atraviesen el campo de batalla durante el tiempo que dure la misión sean igual o superiores en número a nuestro número de vidas inicial.

En este caso se trata de un videojuego que tiene una mecánica algo más difícil que el anterior, pero sin llegar a la complejidad de un juego de estrategia de los que disponemos en equipos de sobremesa. Las partidas son cortas y se pueden pausar a mitad, facilitando jugarse en cualquier sitio.

A medida que avanzamos, el juego nos ofrece niveles más complejos con nuevos enemigos que guardan peculiaridades como ser débil a un tipo concreto de armamento y más resistentes a otros.

No se intenta en ningún caso incentivar que los usuarios sigan jugando mediante una historia, puesto que el juego carece totalmente de ella a parte de la que se entrevé debido al objetivo del juego.

En cuanto al apartado gráfico, nuevamente esta cuidado ofreciendo unos gráficos oscuros, acordes con los sucesos que el juego plantea, y simulando elementos en 3D, aunque se trate realmente de un juego 2D.

En este caso el éxito reside en que se trata de un juego nuevamente muy adictivo, que además presenta un mecanismo de juego más complejo que el del juego anterior, pero que en ningún caso requiere de excesivo tiempo el hacerse con él, y unas pantallas que pueden llegar a constituir retos muy difíciles de superar y que requieren elaborar estrategias.

Controles: Nuevamente el control se lleva a cabo a través de la pantalla táctil, pero en este caso no se busca tratarla como una forma de llevar más realismo al usuario (como en el caso de estirar la cuerda del tirachinas). En este juego la pantalla táctil implementa un control que solo busca ser cómodo para el usuario a través de pulsar botones y lugares específicos de la pantalla.

Beneficios: En este caso se trata de un juego de pago. El precio es de 2€, un precio poco mayor a la media de entre los precios que vemos en los juegos de pago más descargados.

Eso sí, el juego tiene disponible una versión gratuita, con algunas limitaciones. Estas limitaciones no afectan a la *jugabilidad*, sino más bien a la variedad de pantallas, de niveles, etc. Las descargas de esta versión gratuita se encuentran entre los 5 y 10 millones. Es importante decir que ninguna de las dos versiones muestra anuncios por pantalla.

Versión S.O.: Nuevamente la versión de Android mínima es la 1.6. Se elige, por tanto, intentar llegar a contra más usuarios mejor frente a la posibilidad de utilizar las mejoras que las versiones más recientes de Android puedan ofrecer.

Herramientas: Se desconocen las herramientas empleadas para el desarrollo del juego.

Título: Doodle Jump

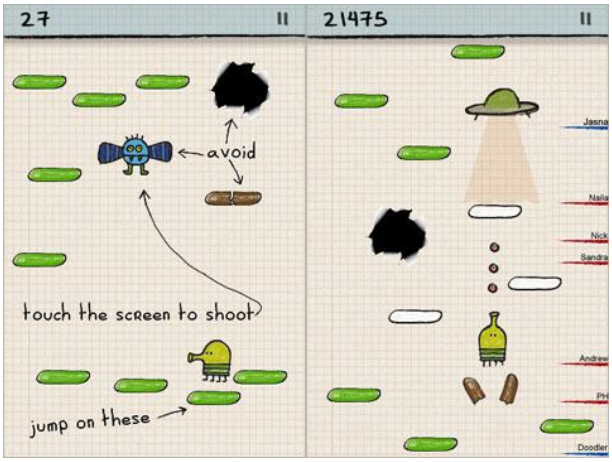


Figura 2.13: Doble captura de pantalla del videojuego Doodle Jump. En la primera captura podemos observar una explicación de los diferentes elementos del juego.

Género: Plataformas

Descripción:

Otro juego muy popular que podemos encontrar en el Android Market. Este tiene entre 100 y 500 mil descargas.

Se trata de un juego que se ha hecho muy famoso debido, como ya llegamos viendo a lo largo de los juegos examinados, a lo adictivo que resulta. En Doodle Jump tenemos una pantalla, la cual se extiende a lo largo hasta el infinito. El mecanismo es algo peculiar, puesto que simplifica la forma de jugar que nos presentan los juegos de plataformas habitualmente.

Primero de todo, en este juego, avanzamos hacia arriba siempre, puesto que si nos salimos de la pantalla por la izquierda apareceremos por la derecha y al revés. Por otro lado, nuestro personaje no para de saltar en ningún momento.

De forma que si en su salto encuentra una plataforma superior utilizará esta para dar un salto que aún lo haga subir más arriba. Nosotros lo que debemos es orientar este salto hacia la izquierda o hacia la derecha utilizando el sensor de orientación del dispositivo, haciendo que nuestro personaje alcance las plataformas que le permitan saltar más alto y teniendo cuidado con enemigos y obstáculos.

Partiendo de este sistema de juego tan simple, debemos avanzar lo máximo posible hacia arriba, teniendo en cuenta que el nivel cada vez se complica más, presentándonos plataformas con propiedades especiales o enemigos más poderosos (ver Figura 2.13). Cuando finalmente nos abatan, los metros avanzados serán nuestra puntuación en un ranking global (online).

Todo el juego está envuelto en un diseño artístico que imita al de un dibujo con lápices de colores, muy cuidado y atractivo.

Aquí tampoco se incentiva que el usuario siga jugando a través de una historia, nuevamente el juego carece de ella. En su lugar el usuario se ve incentivado a través de la simpleza y sobre todo a través de las puntuaciones obtenidas. La comparación de puntuaciones en rankings hace que el jugador tenga ganas de superarse constantemente.

Por tanto, podemos concluir que el éxito del juego reside en la simpleza del mecanismo de juego, en el diseño artístico de los elementos y sobretodo en el reto que supone la competición para conseguir más puntos que tus amigos o que el resto de jugadores de todo el mundo.

Controles: El control, como ya se ha comentado, se lleva a cabo a través del sensor de orientación, que mide la inclinación de nuestro dispositivo y, por tanto, detecta si se quiere dirigir el salto del personaje hacia la izquierda o hacia la derecha.

Por otro lado, para disparar, debemos presionar sobre la pantalla, según el ángulo formado entre el punto de presión y el personaje el disparo llevará una dirección u otra.

Por tanto, podemos decir que el juego tiene un control atractivo, donde a través del sensor de orientación se muestra al usuario un modo de control diferente, que no está acostumbrado a utilizar.

Beneficios: En este caso se trata de un juego de pago. El precio es de 70 céntimos, un precio bajo si lo comparamos con el resto de juegos con buenas ventas que encontramos en el Market. Eso sí, el juego no dispone de ningún tipo de versión de prueba, obligandonos a su compra si queremos probarlo.

Versión S.O.: Esta vez la versión mínima de Android exigida es la 1.5, una versión más baja que la de los anteriores juegos.

El juego se ha puesto a la venta también para iOS (el Sistema Operativo de Apple para iPhone) y otros S.O. para teléfonos inteligentes. Además también hay una versión para la consola de sobremesa Xbox 360.

Herramientas: El juego en primera instancia se desarrollo en el entorno de desarrollo Xcode, para sistemas operativos de Apple y, en concreto se utilizó la librería OpenGL. Y seguidamente fue adaptado al entorno Android.

7.2. Consideraciones extraídas a través del análisis

Una vez analizados los tres juegos anteriores y examinados el resto de juegos que parecen tener un mayor número de ventas y una popularidad más elevada podemos extraer las siguientes consideraciones:

Sistema de juego y género

Por un lado, concluimos que, para que el juego tenga éxito con mayor probabilidad, este debe presentar un **sistema de juego adictivo y fácil de aprender**, que se enseñe poco a poco.

El secreto para hacer un juego adictivo radica en una mecánica fácil pero que permita realizar unas pantallas difíciles de superar, para que el usuario no se aburra. Además, debemos sorprender al jugador a medida que avanza el juego con nuevos escenarios, enemigos, habilidades, etc.

Dicho esto, si observamos los juegos que tienen éxito en el Android Market, a pesar de que parece que las plataformas, los juegos de habilidad y los puzles tienen más éxito, los juegos de estrategia (de más a menos complejos), de rol, etc. también hacen acto de presencia. Por lo que parece que todo género es viable dentro del mercado de videojuegos para móviles si está bien realizado.

Eso sí, se percibe que los géneros deben intentar adaptarse para, ante todo, poderse pausar en cualquier momento. Esto también significa dotar al videojuego de un sistema de juego más simple para que el usuario no tenga que estar completamente metido en el juego ni recordar excesiva información sobre una partida después de una pausa. Debemos tener presente que los juegos para móvil se suelen jugar en tiempos muertos que los usuarios tienen en su día a día.

Diseño artístico y historia

Concluimos que es muy importante que el diseño artístico sea **atrayente** puesto que desde el momento de descargarse el juego, este entra por los ojos en la mayoría de los casos.

Sobre todo, sea un diseño más o menos cuidado, lo imprescindible es que los diseños guarden **coherencia entre sí**, presentando un apartado visual donde los diferentes elementos cuadren dentro de un mismo estilo.

Por el mismo motivo, también es importante que dentro del Android Market el videojuego tenga imágenes, videos y una buena descripción, para conseguir de esta forma que los usuarios puedan ver de qué va el juego, antes si quiera de descargarlo y observen que es un proyecto serio, presentado de forma cuidada.

La historia, por otro lado, es preferible que sea muy simple para no aburrir al usuario, que lo que persigue en principio es poder jugar durante un tiempo breve.

Pero también es importante. Si añadimos una **historia simple** los personajes tendrán **mayor carisma** y esto facilitará el éxito tanto de un primer videojuego como de segundas partes. Para no cansar al usuario, se podría evitar poner videos largos o demasiado texto, muchas veces con un pequeño video sin texto o a través de los propios niveles del juego ya podemos dar ese significado a la historia que consiga que la gente recuerde a los personajes y al entorno.

Controles del juego

En cuanto a los controles, lo más importante es que sean **cómodos** e **intuitivos**. Pero además si introducimos algún tipo de control diferente al que los usuarios están acostumbrados a ver, o más realista, ayudará a que el juego guste más. Ejemplo de ello son los gestos empleados para utilizar el tirachinas en Angry Birds o el uso del acelerómetro en Doodle Jump.

Aspectos económicos

Si nos centramos ahora en los beneficios, podemos sacar unas conclusiones bastante valiosas:

En primer lugar, se puede apreciar que la diferencia entre el número de descargas cuando un juego es gratis o cuando es de pago es enorme.

Pero es que además se ha observado que es importantísimo que el juego tenga **al menos una demo gratuita**, donde los usuarios prueben la mecánica del juego y se puedan "*enganchar*". Si no existe esa demo es mucho más difícil que los usuarios, ante tanta oferta gratuita y de pago opten por comprarlo.

De hecho podemos apreciar cómo un juego como es Doodle Jump, que parte con la ventaja de haber sido promocionado en muchísimos medios, habiendo causado mucho revuelo entre los usuarios, con un precio muy reducido y con una mecánica enfocada a un conjunto más general de usuario, tiene menos ventas que Robo Defense.

Recordemos que Robo Defense este último es un juego exclusivo para Android, poco promocionado, con un aspecto visual menos atrayente y una mecánica no tan enfocada a todos los públicos como la que tiene Doodle Jump. Pero eso sí, Robo Defense tiene una versión gratuita, que permite que los usuarios prueben el juego antes de comprarlo.

Las estadísticas expuestas por Google muestran conclusiones parecidas. En una conferencia de Mayo de 2010, se indica que el 80% de los juegos de pago más descargados del Market (observando los 10 primeros) tienen demo o versión gratuita.

En lo que al precio respecta, el **precio promedio** de las aplicaciones de pago más vendidas **es de 1,80€** (cogiendo como muestra los 10 juegos más populares del Market a día 16 de Mayo de 2011).

Por otro lado, no hay que olvidar la opción de recibir dinero por un juego basándonos únicamente en la publicidad. A pesar de que podamos poner demo gratuita o no, las versiones de pago siempre son descargadas muchas menos veces que una versión gratuita. Robo Defense Free tiene entre 5 y 10 millón de descargas, una diferencia muy grande con respecto a las entre 500mil y 1 millón de descargas de la versión de pago. Y si comparamos con las descargas de Angry Birds aún es más grande dicha diferencia.

De hecho los desarrolladores de Angry Birds reconocen en diversas entrevistas que en Android es más difícil vender aplicaciones que en otros medios como la tienda digital de Apple. Siendo **a través de la publicidad** como se puede ganar más dinero con un juego para Android.

Esto tiene mucho sentido si tenemos en cuenta el medio a través del cual gana dinero Google con sus usuarios, que normalmente pasa por ofrecer todo gratuitamente y recibir ingresos por publicidad. Por tanto, es comprensible que los usuarios de Android prefieran las aplicaciones gratuitas y no les importe tener una pequeña publicidad en la pantalla de juego a cambio.

Estadísticas extraídas por la empresa Distimo nos ayudan a ponernos en situación (ver figura 2.14) y nos muestran como más de la mitad de las aplicaciones del Android Market son gratuitas. Siendo el porcentaje de aplicaciones gratuitas (65%) el más grande de entre todas las tiendas de aplicaciones analizadas. Y, además, observamos como el Android Market tiene más aplicaciones gratuitas que cualquiera de las tiendas de la competencia.

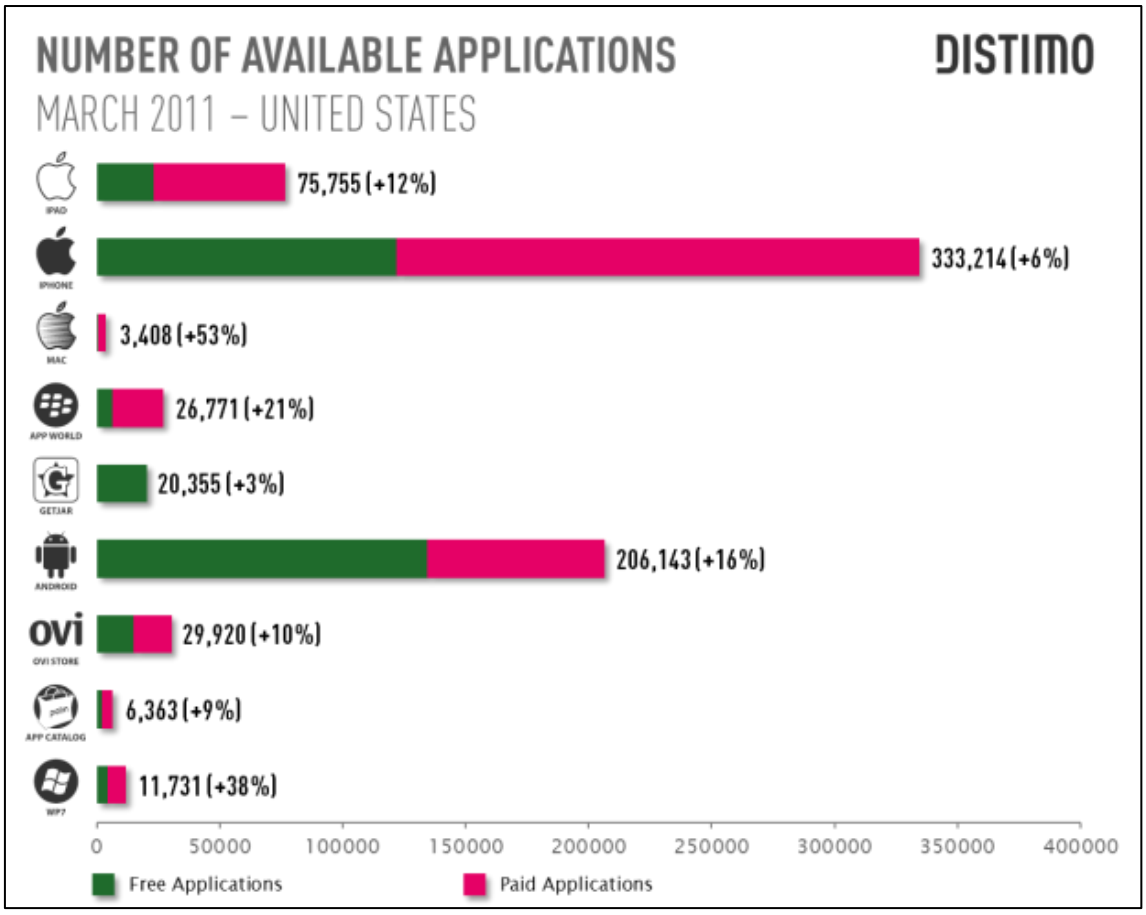


Figura 2.14: Estadísticas proporcionadas por la empresa Distimo correspondientes a Marzo de 2011 en Estados Unidos. En ellas podemos ver el número de aplicaciones total de cada una de las tiendas virtuales de los diferentes sistemas operativos móviles, junto con el porcentaje de crecimiento del último mes.
Además podemos ver la proporción entre aplicaciones gratuitas y de pago de cada sistema.

Nótese que aunque la estadística se haya llevado a cabo con el Android Market de Estados Unidos, la diferencia de las tiendas virtuales entre países suele estar en las aplicaciones que tienen restricción regional y no se distribuyen en todos los países. Así que puede haber una pequeña diferencia entre las estadísticas en EEUU y en España, pero no la suficiente como para alterar las conclusiones extraídas.

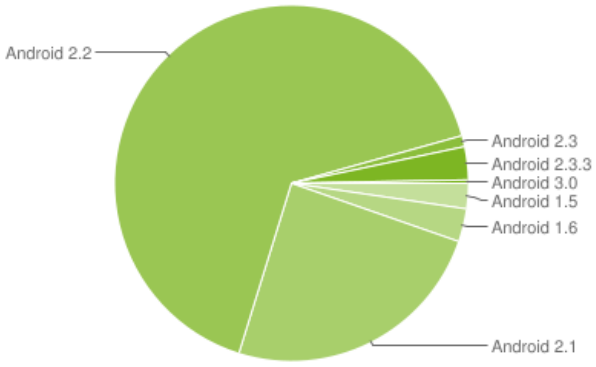
Por otro lado, por lo que se observa viendo las múltiples partes que tienen muchos de los juegos que se encuentran en el Market, parece que una buena forma de obtener beneficios es **desarrollar un juego con un motor que facilite el añadir elementos y pantallas**.

De esta forma es posible desarrollar nuevas pantallas con nuevos enemigos que reten de nuevo al jugador para venderlos a un precio muy asequible o distribuirlos gratuitamente con publicidad. Es habitual leer comentarios de usuarios que puntúan a los juegos que les gustan en el Market solicitando más pantallas para los diferentes juegos, puesto que quieren nuevos retos.

Por tanto, parece que la **extensibilidad**, **portabilidad** y **reusabilidad** del motor del juego que desarrollemos es uno de los objetivos que deberíamos perseguir.

Compatibilidad entre versiones de Android

Podemos observar como **la mayoría de los juegos son compatibles con la versión 1.6** y superiores de Android. Vamos a intentar analizar si el motivo se debe a una mayoría de usuarios con esta versión en el momento del desarrollo o a que, a pesar de que a día de hoy tan solo un 3% de los usuarios de Android estén utilizando la versión 1.6 (ver figura 2.15), sigue siendo la más recomendable para el desarrollo de videojuegos.



Platform	API Level	Distribution
Android 1.5	3	2.3%
Android 1.6	4	3.0%
Android 2.1	7	24.5%
Android 2.2	8	65.9%
Android 2.3	9	1.0%
Android 2.3.3	10	3.0%
Android 3.0	11	0.3%

Figura 2.15: Porcentaje de uso de cada versión del S.O. Android, sobre el total de usuarios de Android, a día 2 de Mayo de 2011. Este porcentaje se mide monitorizando los accesos al Android Market.

En caso de que a día de hoy desarrolláramos un videojuego para la versión 2.2 o superior estaríamos dejando fuera a una gran cantidad de usuarios que tienen versiones antiguas (especialmente la versión 2.1 es muy utilizada con un 24,5%). Aún así, la versión 2.2 puede ser una opción a tener en cuenta si vamos a desarrollar un videojuego en 3D. Esto es debido a que la compatibilidad con OpenGL ES 2.0 permite el uso de *shaders* y, por tanto, facilita algunas cosas mientras que posibilita otras nuevas. Estamos llegando a un 70% de los usuarios de Android y conseguimos cosas que no podemos conseguir de otra forma.

Si, por el contrario, quisiéramos trabajar con una versión más moderna y nos diera por escoger la versión 2.3 como requisito mínimo, estamos dirigiéndonos a poco más de un 4% de los usuarios. Por tanto, parece que a día de hoy sacar un juego para la versión 2.3 de Android o superior no vale la pena aunque las mejoras que esta versión del sistema operativo aporta puedan facilitar bastante la tarea de desarrollo de videojuegos.

Podríamos deducir, por tanto, que lo más inteligente si no necesitamos *shaders* es desarrollar para la versión 2.1 (nótese que cuando desarrollamos una aplicación para una versión esta funciona en ésta versión y en las superiores), que por otro lado, no aporta casi mejoras en el desarrollo de videojuegos con respecto a la versión 1.6, con lo cual la mayoría de desarrolladores se decantan por **la versión 1.6 aún a día de hoy como requisito mínimo**. De esta forma consiguen llegar prácticamente a todos los usuarios.

Esto supone tener que añadir un selector para distinguir entre controles de pantalla *singletouch* y controles de pantalla *multitouch*. Puesto que el soporte para pantallas *multitouch* es la mejora más notoria, de cara al desarrollo de videojuegos, que se añade en la versión 2.1 de Android.

Por tanto, habría que hacer un inciso, puesto que si los controles del videojuego no se prestan a ser implementados mediante una pantalla mono táctil, ya que se limita la *jugabilidad* haciendo que la experiencia de usuario se vea excesivamente afectada, también sería una buena elección trabajar para la versión 2.1 de Android. De esta forma, ahorraríamos el tiempo necesario para implementar y probar un control mono táctil que va a ofrecer una mala experiencia de usuario.

Tampoco tenemos que olvidar que, cuando finalicemos el desarrollo del videojuego, el grafico habrá cambiado, y las versiones 1.5 y 1.6 serán, con toda probabilidad, más minoritarias de lo que ya lo son actualmente, cuando solo representan al 5,3% de los usuarios de Android.

Herramientas de desarrollo

En cuanto a las herramientas de desarrollo a utilizar, es importante tener en cuenta que muchos de los videojuegos son multiplataforma, y salen para diferentes sistemas operativos. Esto es debido a que tal y como está actualmente el mercado, hay muchos S.O. compitiendo y aún no hay un claro vencedor. Con lo cual, si conseguimos portar fácilmente el juego a diversas plataformas nos aseguramos más beneficios.

Concluimos por tanto que es aconsejable el **uso de herramientas multiplataforma** que nos permitan: portar nuestro juego de un sistema operativo a otro casi sin demasiado esfuerzo o utilizar los conocimientos que adquiriremos en el desarrollo para futuros desarrollos en otras plataformas.

Por este motivo se suele utilizar OpenGL por encima de la librería gráfica de Android, aparte de que OpenGL presenta ventajas en cuanto a eficiencia y potencia. También se pueden utilizar motores multiplataforma como hacen, por ejemplo, desde Rovio Ltd. (desarrolladores de Angry Birds) donde se optó por utilizar un motor de físicas multiplataforma como es Box2D para desarrollar el videojuego.

Por último, decir que a la hora de coger muestras para realizar este apartado se ha optado por guiarse a través de los juegos que el Android Market presenta en su web como videojuegos más populares. Aunque no se sabe que algoritmo se sigue para extraer los que son los videojuegos más populares, se entiende que se basan en la puntuación (número de puntuaciones y nota asignada en estas puntuaciones) y el número de descargas, en un intervalo de tiempo corto.

Por tanto, para asegurarnos de que son juegos que realmente son valorados y descargados no solo en un momento puntual sino durante un tiempo prolongado, se ha apoyado la selección en la observación del Market durante un período de tiempo largo y en la búsqueda de información sobre los videojuegos tratados a través de otros medios, entre los que se incluyen otras plataformas de venta de videojuegos.

8. Análisis de herramientas para el desarrollo de videojuegos en Android

En este apartado se van a analizar las diferentes herramientas que el desarrollador de videojuegos tiene disponibles en la plataforma Android. A través de este análisis y de las consideraciones extraídas del análisis de antecedentes, se han escogido las herramientas que posteriormente se han utilizado en el desarrollo de los dos videojuegos.

Primero de todo, debemos tener claro que lo que se ha buscado en este proyecto es centrarse en el desarrollo de videojuegos en 2D. Es por ello que hay que buscar herramientas para el desarrollo de este tipo de juegos.

8.1. Desarrollo a bajo y a alto nivel

Además, a la hora de desarrollar un videojuego, podemos optar por dos alternativas. En primer lugar, utilizar un motor ya prefabricado, en cuyo caso trabajaríamos a alto nivel sobre este motor. O decidir crear nosotros nuestro propio motor, con lo cual deberíamos trabajar con librerías de gráficos y de físicas, aunque estas últimas también son prescindibles si queremos trabajar a un nivel aún más bajo.

Las ventajas de trabajar a bajo nivel son que tenemos un mayor control de nuestra aplicación, pues trabajamos con pocas capas por debajo, y podemos entender con mayor facilidad todos los entresijos de la implementación de nuestro videojuego.

Además, las librerías a bajo nivel suelen estar muy bien documentadas, pues son librerías multiplataforma muy utilizadas y además algunas forman parte del soporte oficial de Android para el desarrollador. Son el caso de la API de Android o la librería de gráficos OpenGL ES 2.0. En cambio los motores suelen estar peor documentados, por el simple hecho de que muchos están construidos por otros desarrolladores muchas veces sin ánimo de lucro.

En todo caso, pueden llegar a haber motores muy bien documentados, que estén hechos por empresas realmente importantes, pero estos suelen ser de pago y en este proyecto se busca utilizar recursos gratuitos. Una mala documentación junto con el trabajo a alto nivel puede provocar que un error cueste más de encontrarse, es más, si el error proviene del propio motor realmente puede ser un calvario.

En cuanto a las desventajas de trabajar a bajo nivel, está el hecho de que es más complicado, pues hay que construir todo el motor y, en cambio, trabajar sobre un motor ya fabricado nos permite acelerar en gran medida el desarrollo.

Por tanto, concluimos que valdrá la pena utilizar un motor en nuestro proyecto, si este está pensado para videojuegos en 2D, es multiplataforma, nos permite trabajar con él de forma gratuita y, al mismo tiempo, está bien documentado.

Analizando las alternativas disponibles, descartamos Unity3D y Havok, por ser motores para desarrollar juegos en 3D.

Existe también otro problema y es que a la hora de gestionar las físicas, si lo hacemos en Java el mecanismo de recolección de basura y la lentitud del lenguaje provoca que no podamos tener demasiados elementos físicos en el escenario, por este motivo quedan descartados también motores cuya librería de físicas está basada en Java, como Cocos2D.

Después de analizar los motores disponibles, vemos que hay dos alternativas muy interesantes, son LibGDX y AndEngine. La gracia de estos motores, es que viendo el problema de gestionar las físicas en el lenguaje Java, el equipo de desarrollo que hay detrás de LibGDX decidió coger la versión en C de la librería Box2D, una librería de físicas muy famosa, y portarla a Android creando una librería en Java que accede a las operaciones C de forma tranparente para el desarrollador.

De esta forma, podemos, únicamente utilizando el lenguaje Java, beneficiarnos de la buena eficiencia que aporta el lenguaje C en la gestión de físicas. AndEngine también utiliza la librería de LibGDX para ello.

El problema de estos dos motores es que no son multiplataforma y, por tanto, desarrollar una aplicación para ellos implica mayor dificultad a la hora de portar esta aplicación a otros sistemas operativos. Además, son motores creados sin ánimo de lucro, lo cual provoca que no podamos exigir un motor sin ningún tipo de error. Por estos dos motivos se decidió desarrollar los dos videojuegos sin utilizar un motor ya existen y, en su lugar, crear un motor propio utilizando librerías bien conocidas.

De esta forma los dos videojuegos son más sencillos de portar a otros sistemas operativos, pues utilizamos Java junto con unas librerías que se encuentran disponibles en muchas plataformas y lenguajes diferentes.

En concreto, para el segundo videojuego se ha utilizado la librería de físicas Box2D [10] que como ya se ha dicho es multiplataforma y, además, uno de sus puntos fuertes es que está pensada para funcionar en dispositivos con pocos recursos, pues prioriza la eficiencia a la exactitud en los cálculos cuando esta exactitud no es demasiado importante a ojos del usuario. En cambio, para el primer videojuego, al ser bastante más simple se han implementado las físicas sin utilizar ninguna librería.

Eso sí, como ya se ha explicado no se puede utilizar la versión en Java de Box2D pues es ineficiente, así que se ha utilizado la versión que nos ofrece LibGDX [11]. El utilizar

solo la librería de físicas de LibGDX ha comportado un problema, y es que el modo *debug* de físicas se ha tenido que construir a mano específicamente para el videojuego, pues utilizar el que incluye LibGDX implicaba utilizar todo el motor, lo cual se quería evitar por los motivos antes expuestos.

8.2. Escogiendo una librería de gráficos

En cuanto a la librería de gráficos ha emplear, hay dos opciones, pues Android a parte de la conocida librería OpenGL ES a la cual da soporte, implementa una librería de gráficos propia. La librería incluida con Android es fácil de utilizar, pero su uso está pensado para videojuegos muy simples, pues no utiliza aceleración por hardware, lo que significa que, en lugar de utilizar la GPU, lleva a cabo toda la creación del *render* a través de la CPU. Esto hace que su la creación del *render* sea mucho menos eficiente.

Figura 2.16: Logo de la librería de gráficos OpenGL ES.

Por tanto, se decidió utilizar OpenGL ES (ver Figura 2.16), librería que aparte de ser multiplataforma está muy bien documentada y existen diferentes libros a los que poder acudir en caso de tener alguna duda sobre su funcionamiento [12].

Nótese que la variante ES de OpenGL es una librería adaptada a dispositivos móviles y que, por tanto es más exigente, pues restringe algunas funciones de la librería y obliga a hacer uso de los mecanismos más eficientes disponibles para llevar a cabo la creación de los *renders* [13]. Elimina, por tanto, duplicidades.

Pero aun queda un tema más que tratar, pues existen diferentes versiones de la librería OpenGL ES. En concreto la versión 2.0 permite el uso de *shaders*, que vienen a ser una serie de algoritmos que se ejecutan en la propia GPU y que permiten mayor personalización y eficiencia en nuestro videojuego [14]. A pesar de ello se decidió hacer uso de la versión OpenGL ES 1.0 en ambos videojuegos, pues así la aplicación puede estar dirigida a usuarios de la versión 2.1 o superior de Android. En cambio, si se hace uso de OpenGL ES 2.0 solo podemos dirigirnos a las versiones 2.2 (*Froyo*) y superiores, que es cuando se añade el soporte.

Adicionalmente, se ha implementado en el segundo videojuego un uso opcional de características concretas de la versión OpenGL ES 1.1. Esta es una versión soportada por el S.O. desde versiones anteriores, pero la cual no tiene por qué tener soporte por parte de todos los dispositivos. Lo que se ha hecho para conservar la compatibilidad con todos los dispositivos es implementar un selector, que consulta si el hardware es compatible con la versión 1.1 y, en caso afirmativo, utiliza la extensión que permite el uso de VBOs [15]. Si, por el contrario, no hay compatibilidad con esta versión se ejecuta el nivel del videojuego sin hacer uso de dicha extensión.

Capítulo 3 – Primer videojuego

Entramos en la parte de la memoria donde se describen en profundidad los diferentes productos construidos, así como las decisiones que se han tomado para su desarrollo, la explicación de cómo se han adaptado los conocimientos adquiridos hasta ahora y los aspectos conceptuales directamente relacionados con funcionalidades de estos.

En primer lugar se ha desarrollado un videojuego con una mecánica simple, he aquí toda la información sobre este.

1. Descripción del videojuego

"Ping Pang Pung" está basado en un antiguo videojuego de recreativas muy conocido. Este producto responde al tipo de videojuego que, tal y como concluimos en el análisis de antecedentes, tiene más éxito en las plataformas móvil, es decir, que se trata de un juego con una mecánica muy simple y adictiva que no requiere de mucha atención para el aprendizaje por parte del jugador.

Figura 3.1: Captura de pantalla del videojuego "Ping Pang Pung". Aparecen los diferentes elementos del juego señalizados: A) Personaje; B) Bola; C) Disparo

Elementos del juego.

El juego consta de tres elementos principales que interaccionan creando la experiencia de juego (ver Figura 3.1):

El personaje: se trata del personaje que nosotros controlamos y que se mueve por un escenario rectangular.

Las bolas: por el escenario rebotan unas bolas que, en caso de tocar al personaje, le dañarán. Lo que provocará el uso de una vida para poder continuar jugando. Disponemos de tres tipos de bola diferentes: la bola pequeña, la bola mediana y la bola grande. Los tipos se diferencian en tamaño, pero también en velocidad y altura (contra más pequeña sea la bola más rápida es y más alto rebota). (Ver Figura 3.2)

Figura 3.2: Comparación de tamaño entre los tres tipos de bolas.

El disparo: nuestro personaje se puede defender ante las bolas gracias al cañón que posee, el cual lanza una flecha que recorre la pantalla de abajo a arriba y que al golpear cualquier bola le provocará un daño. Eso sí, solo podemos tener una flecha en pantalla al mismo tiempo, por tanto, el jugador debe pensar bien donde disparar, puesto que ante un fallo quedará expuesto a las bolas sin nada para defenderse.

El disparo recorre la pantalla de abajo a arriba y desaparece cuando toca el techo del escenario o cuando toca a una (o varias) bolas.

Al recibir daño, las bolas tendrán una reacción según su tamaño. Si es de tamaño pequeño la bola desaparece, pero si no es así, dicha bola se convierte en dos bolas de tamaño un nivel inferior al nivel de la bola alcanzada.

Mecánica del videojuego.

Puestos en situación, la meta del juego es conseguir el máximo número de puntos posibles antes de que todas las vidas del personaje sean arrebatadas. Los puntos son otorgados cuando provocamos daño a una bola.

Es preciso aclarar que el juego no tiene final, de forma que cada nivel se inicializa con tantas bolas grandes como el número del nivel (siendo el nivel inicial el 1) y una vez que acabemos con esas bolas (reduciéndolas de tamaño hasta eliminarlas) se inicia el siguiente nivel.

El juego se ha llegado a probar hasta el nivel 10, un nivel muy difícil de superar y el funcionamiento en el dispositivo móvil utilizado para las pruebas es totalmente fluido.

Manejo por parte del usuario.

El control del personaje se realiza a través de la pantalla táctil. El usuario toca el punto del escenario a donde quiere que el personaje se desplace y este se dirigirá hacia la X marcada (hay que tener en cuenta que el personaje no salta ni, en definitiva, se desplaza a lo largo del eje Y).

En cambio si lo que se quiere es disparar, el usuario deberá presionar sobre el área cubierta por un botón medio transparente que muestra el texto “¡Disparar!”.

El videojuego se ha desarrollado con el objetivo de poderse reproducir en el máximo número de versiones de Android disponible, objetivo que, como concluimos en el apartado donde se trataron los Antecedentes es muy importante, y más cuando tenemos un juego que no requiere demasiada potencia y que, por tanto, debería funcionar correctamente en dispositivos poco potentes.

Por tanto, se trata de un videojuego preparado para reproducirse en la versión 1.5 y superiores de Android. Esto, sin embargo, comporta un problema: debido a que el soporte para pantallas *multitouch* se añade en las versiones 2.0 y superiores podemos o añadir soporte para ambos tipos de pantalla o dar soporte únicamente a las pantallas que admiten un único punto de contacto (*pointer* en inglés).

En este caso, debido a que se trata de un producto que se debía implementar en un período de tiempo corto se ha optado por implementar solo el soporte a pantallas táctiles que admiten solo un punto de contacto. Esto significa que tenemos una aplicación compatible con casi la totalidad de versiones del sistema operativo Android y que además funciona en móviles que tengan cualquier tipo pantalla táctil. Consiguiendo, por tanto, la compatibilidad con la mayoría de dispositivos.

El punto negativo es que los usuarios que tengan pantalla *mutitouch* no la aprovecharán.

El mecanismo de funcionamiento del receptor de las entradas en pantalla táctil detecta como entrada la acción de ponerse en contacto con la pantalla, es decir, el momento en que el dedo del usuario entra en contacto con la pantalla. Al ser una pantalla que no distingue entre varios puntos de presión, para que el juego funcione correctamente deberos retirar el dedo de la pantalla antes de presionarla nuevamente con otro dedo, puesto que en caso de que presionáramos antes de retirar el primer punto de presión, el dispositivo detectaría la acción como un movimiento del primer dedo y no se mostraría resultado alguno.

Interface con la información de la partida.

El Hud, en inglés, es el nombre que recibe una interface de la que disponen la mayoría de los juegos y que contiene información de utilidad para el usuario acerca de la partida que se está jugando en un momento dado (ver Figura 3.3).

En este primer proyecto el Hud nos informa acerca del valor de dos variables vitales para la partida en cuestión: se trata de la puntuación alcanzada y del número de continuaciones (vidas del jugador) restantes. Ambas variables son actualizadas en tiempo real.

Además, también forma parte del Hud el botón que recubre el área donde, en caso de presión del usuario, se efectuará un disparo.

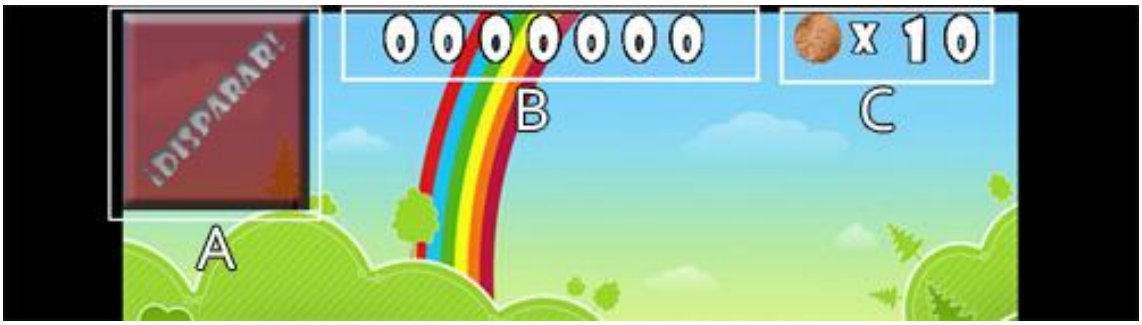


Figura 3.3: Captura de pantalla del videojuego donde se resaltan los diferentes elementos del *Hud* de este. A) Botón de disparo; B) Marcador de Puntuación; C) Número de continuaciones restantes.

Herramientas utilizadas.

En cuanto a las herramientas utilizadas, la aplicación se ha desarrollando utilizando el lenguaje Java, el entorno de desarrollo que proporciona el *plugin* ADT para Eclipse y la implementación de OpenGL ES 1.0 que Android proporciona.

La aplicación funciona en las versión 1.5 y superior de Android y por este motivo se trata de un juego con compatibilidad con la totalidad de dispositivos Android del mercado.

Por otro lado, no se ha hecho uso de Box 2D, puesto que este se utilizará en el siguiente producto, que posee una física más compleja. Por tanto, lo que se hace para mover las bolas es utilizar la función del coseno, que nos ofrece un movimiento suficientemente realista.

2. Especificación del videojuego: Casos de Uso

A continuación se presenta la especificación del videojuego. La fase de Especificación ha sido ajustada al tamaño del proyecto; como se trata de una aplicación sencilla y que además se debe hacer en un período de tiempo reducido no es viable hacer una especificación propia de un proyecto más complejo y largo.

2.1. Diagrama de casos de uso

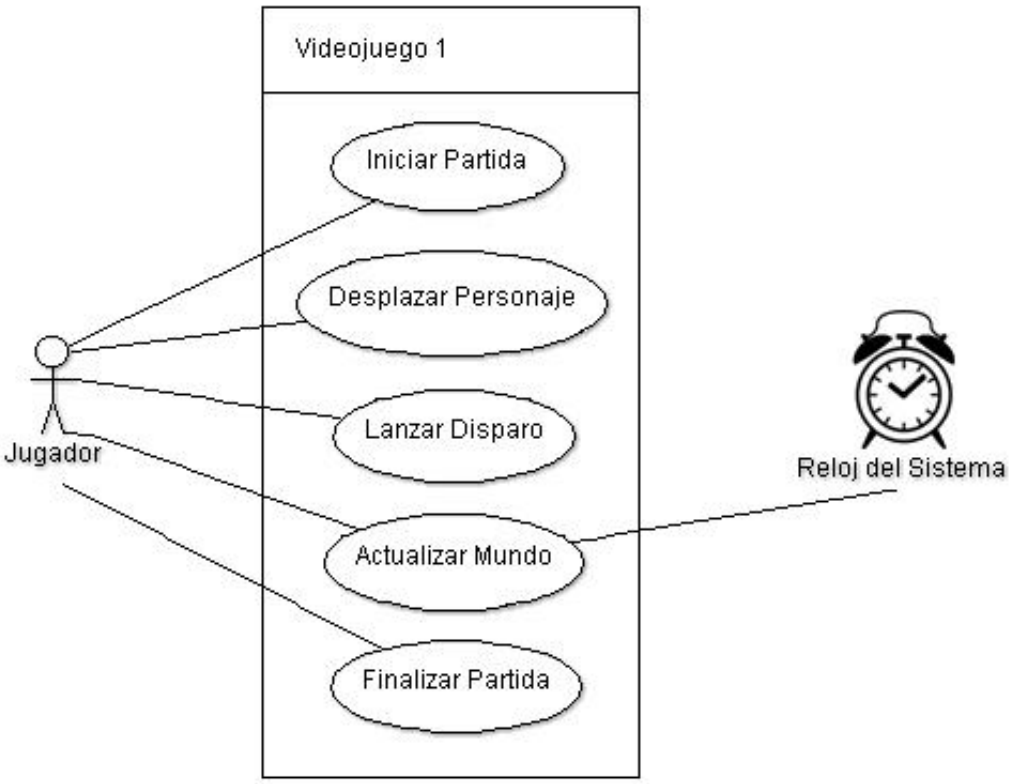


Figura 3.4: Diagrama de casos de uso del videojuego “Ping Pang Pung”.

2.2. Definición de los casos de uso

Caso de Uso: Iniciar Partida

Actores primarios: Jugador

Contexto: El jugador se dispone a iniciar una nueva partida y no hay ninguna partida en curso.

Escenario principal de éxito:

1. El jugador indica al sistema que quiere empezar una nueva partida.
2. El sistema inicializa la puntuación y el número de vidas del jugador a los valores iniciales.
   El sistema carga el escenario, los elementos que en él se encuentran y la interface.

   Por último, el sistema comienza a pintar y mostrar al jugador en tiempo real la evolución del escenario, los elementos y la interface. Se activa, por tanto, el caso de uso "Actualizar Mundo", que se comenzará a reproducir periódicamente.

   *"Actualizar Mundo" se reproducirá hasta que: o el jugador se quede sin vidas y, por tanto, la partida quede en pausa, esperando a la finalización definitiva por parte del jugador; o el jugador decida finalizar la partida antes incluso de que su personaje pierda todas las vidas. (Caso de Uso "*Finalizar Partida*").

Caso de Uso: Desplazar personaje.

Actores primarios: Jugador

Contexto: Hay una partida en marcha y el jugador aún dispone de vidas para poder interactuar con la aplicación.

Escenario principal de éxito:

1. El jugador indica el punto X del escenario a donde quiere que el personaje que controla se desplace.

2. El sistema recibe la X y se encarga de que futuras ejecuciones del caso de uso "Actualizar Mundo" vayan simulando un movimiento del personaje principal; desde la X donde se encuentra en el momento de ejecutar el caso de uso hasta la X indicada por el jugador.

Caso de Uso: Lanzar Disparo.

Actores primarios: Jugador

Contexto: Hay una partida en marcha y el jugador aún dispone de vidas para poder interactuar con la aplicación.

Escenario principal de éxito:

1. El jugador indica al sistema que quiere que se lance el disparo.

2. **Si** hay un disparo en lanzamiento en este momento el sistema no hace más que continuar gestionando el movimiento de dicho disparo.

3. **Si por el contrario** no hay un disparo visible en pantalla el sistema crea el nuevo disparo y lo posiciona en el lugar acorde a la X donde el personaje principal se encuentra en ese momento.

   El sistema se encargará de que futuras ejecuciones del caso de uso "Actualizar Mundo" vayan actualizando su posición, simulando su movimiento.

   Se irá actualizando la posición del disparo hasta que este choque con una o varias bolas, en cuyo caso deberá actualizar el estado de la partida de acuerdo al choque producido; o hasta que choque contra el techo del escenario, momento en el cual el disparo finalizará y otro disparo podrá ser activado en cualquier momento (a través del propio Caso de Uso "*Lanzar Disparo*").

Caso de Uso: Actualizar Mundo

Actores primarios: Jugador

Contexto: El jugador se encuentra jugando una partida y quiere que el sistema simule en tiempo real dicha partida.

Escenario principal de éxito:

1. El reloj avisa al sistema de que ya ha pasado el tiempo que dura un *frame* desde la última vez que se ejecutó este caso de uso.
2. El sistema actualiza los elementos del mundo teniendo en cuenta el tiempo que ha pasado desde la última actualización.
3. El sistema presenta al jugador los nuevos datos actualizados para que, a ojos de este el mundo tenga vida.

   *Para conseguir simular esta vida es preciso que este caso de uso se lleve a cabo continuamente (*frame* a *frame*) mientras la partida este ejecutándose. Y aunque al mismo tiempo se estén ejecutando otros Casos de Uso.

Caso de Uso: Finalizar Partida.

Actores primarios: Jugador

Contexto: Hay una partida en marcha y: o el jugador aún dispone de vidas para poder interactuar con la aplicación o, por el contrario, las vidas se han acabado y, por tanto, la partida esta parada esperando a que se dé por finalizada.

Escenario principal de éxito:

1. El jugador indica al sistema que quiere finalizar la partida en curso.
2. El sistema limpia todo lo referente a dicha partida y deja de mostrar el escenario con elementos e interface (si es que se estaba mostrando), pasando a permitir que el jugador inicie otra partida (Caso de Uso "*Iniciar Partida*") en cualquier momento.

3. Componentes de la aplicación

Nos embarcamos ahora en el diseño de la aplicación. Ya hemos visto, en el apartado 2 del capítulo 2, que las aplicaciones para el S.O. Android se dividen en componentes y estos componentes se clasifican en varios tipos.

En este apartado se van a mostrar los diferentes componentes que conforman el videojuego “Ping Pang Pung” a través de un diagrama llamado Diagrama de Componentes. Dicho gráfico, aparte de mostrarnos los componentes, nos va a permitir entender fácilmente cómo interactúan entre ellos y como el usuario puede navegar por ellos.

Vamos pues a presentar el diagrama (ver Figura 3.5), junto con una descripción detallada de lo que en él se muestra.

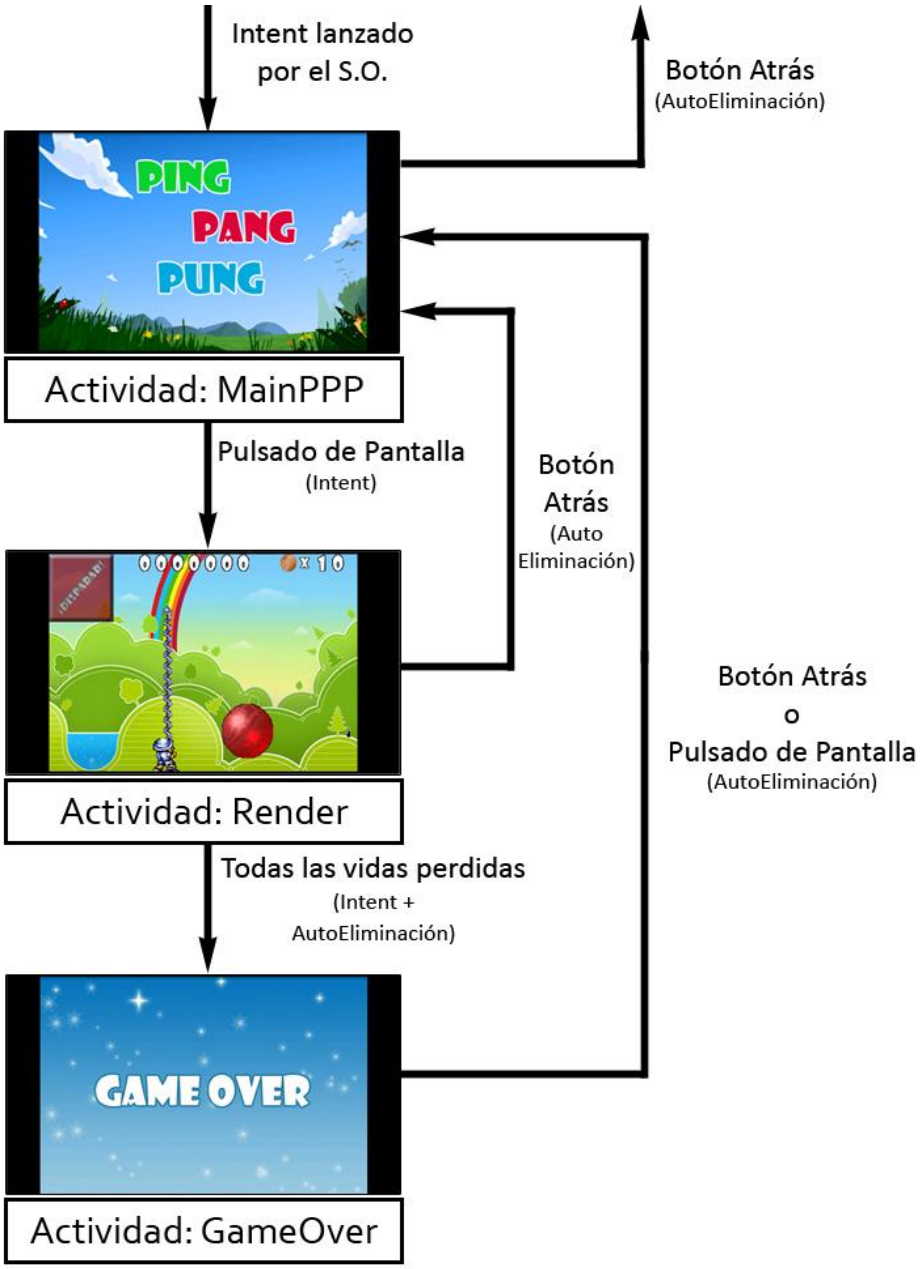


Figura 3.5: Diagrama de Componentes (o Mapa de Navegación) de la aplicación "Ping Pang Pung".

En primer lugar, podemos apreciar como los tres componentes que conforman la aplicación son del tipo Actividad (*Activity*). Como ya vimos en apartados anteriores, tenemos un componente de este tipo por cada pantalla de nuestra aplicación. Podemos decir, por tanto, que las actividades son el equivalente a las *vistas* de las aplicaciones tradicionales. Es por esto que este diagrama de componentes es equivalente al Diagrama de Navegabilidades de diseño, el cual nos muestra los diferentes caminos de navegación que un usuario puede tomar para moverse por la aplicación.

En el diagrama, cada una de las flechas que va de una actividad a otra, representa una navegación que puede ser efectuada durante la ejecución de la aplicación y contiene dos datos muy importantes:

- En primer lugar, el tipo de suceso que se deberá dar o el tipo de acción que el usuario deberá realizar, según el caso, sobre la actividad actual para ir de esta hasta la actividad a la que se dirige la flecha. Un ejemplo puede ser *Pulsar Pantalla*, que, como el propio nombre indica, viene a significar que, para ir de la actividad actual a la siguiente, debemos tocar cualquier punto de la pantalla.

- En segundo lugar, entre paréntesis se encuentran las acciones que realizará la actividad en la que nos encontramos para que al usuario se le muestre la actividad destino, una vez que se ha dado el suceso o acción que se necesita para llevar a cabo el cambio de pantalla. Estas acciones pueden ser tres:
  - *Intent*: lo que significa que la actividad actual crea una nueva actividad, mediante el envió de un *intent*, y se almacena en la primera posición de la pila de actividades. Pasando a ser visible para el usuario la nueva actividad creada. El *intent* no es más que el mecanismo implementado en la API de Android para crear instancias de una actividad desde el código.

  - Auto Eliminación: en este caso, la instancia de la actividad que se está mostrando se elimina a sí misma. Esto da lugar a que la instancia de actividad que está arriba de todo de la pila de actividades vuelva, de nuevo, a ser visible para el usuario. La actividad que se encontraba a la cabeza de la pila era la actividad destino.

  - *Intent* + Auto Eliminación: en este caso se combinan las dos acciones explicadas con anterioridad. La actividad actual se auto elimina, pero no pasa a ser visible la actividad que se encuentra a la cabeza de la pila de actividades. En su lugar, una nueva instancia de actividad, creada por la actividad actual antes de auto eliminarse, pasa a ser visible para el usuario.

Una vez dada la explicación general acerca de cómo entender el Diagrama de Componentes, a continuación se va a describir el funcionamiento concreto de cada uno de las actividades que podemos visualizar en el diagrama anterior:

- En primer lugar tenemos la actividad *MainPPP*, cuya única función es mostrar por pantalla una imagen, la cual al ser pulsada lanzará la actividad *Render*, iniciando una nueva partida de nuestro juego. Cabe decir que es el Sistema Operativo Android el que envía el *intent* que crea la instancia de la Actividad *MainPPP* y lo hace cuando el usuario le indica que desea ejecutar la aplicación "Ping Pang Pung", apretando sobre el icono de dicha aplicación.

- Esta segunda actividad *Render* se encarga de la ejecución de toda la partida: recibe las entradas del usuario, se encarga del pintado de los gráficos que se actualizan en tiempo real y controla al *thread* que ejecuta el bucle del juego (donde se procesa la entrada, se mantiene la física, se comprueba la lógica,…). En el momento en que el jugador se quede sin vidas para poder continuar la partida, esta partida acaba y, por tanto, *Render* lanzara la actividad *GameOver*. No sin antes eliminarse tanto a sí misma como a todos los *threads* que mantiene en ejecución; gracias a esta eliminación nos deshacemos de unos hilos de ejecución que ya no interesan y que están consumiendo recursos.

- Por último, tenemos la actividad *GameOver*. Tal y como la actividad *MainPPP*, solo muestra por pantalla una imagen que en caso de ser pulsada se encarga de auto-eliminarse, dando paso nuevamente a *MainPPP*, que nos permitirá comenzar una nueva partida (recordemos que *Render* ya no está en la pila de actividades debido a su autoeliminación).

Para finalizar este apartado, decir que en cualquier actividad podemos apretar el botón Atrás del terminal para auto eliminar-la y volver a la actividad que se encuentre en la parte superior de la pila de actividades de la aplicación.

4. Diagrama de clases

Continuamos con la fase de diseño de “Ping Pang Pung”. Se ha decidido hacer directamente el diagrama de clases, en lugar de realizar primero el modelo conceptual de los datos, debido a la baja magnitud de este primer videojuego.

Aunque el objetivo de este primer proyecto era realizar un videojuego simple y adictivo, basándonos en los resultados de la investigación de los antecedentes, también se ha querido desarrollar el proyecto creando una primera versión de lo que, en el segundo proyecto será una estructura cambiable. En concreto se ha centrado esta extensibilidad en los elementos del juego (móviles o no). Esta estructura ha facilitado mucho la creación de los diferentes elementos, puesto que buena parte de cada elemento está definida en una clase que cada caso particular extiende y personaliza, si es necesario. Por tanto, a partir de una sola clase diseñamos la base de todos los elementos que habrá en el juego.

A continuación se presenta el diagrama de clases (ver Figura 3.6), incluyendo los atributos y operaciones **más importantes**. Nos centraremos sobretodo en el componente de la aplicación que se encarga de reproducir el videojuego y de gestionar todos los movimientos, entradas e interacciones, puesto que es el más complejo.

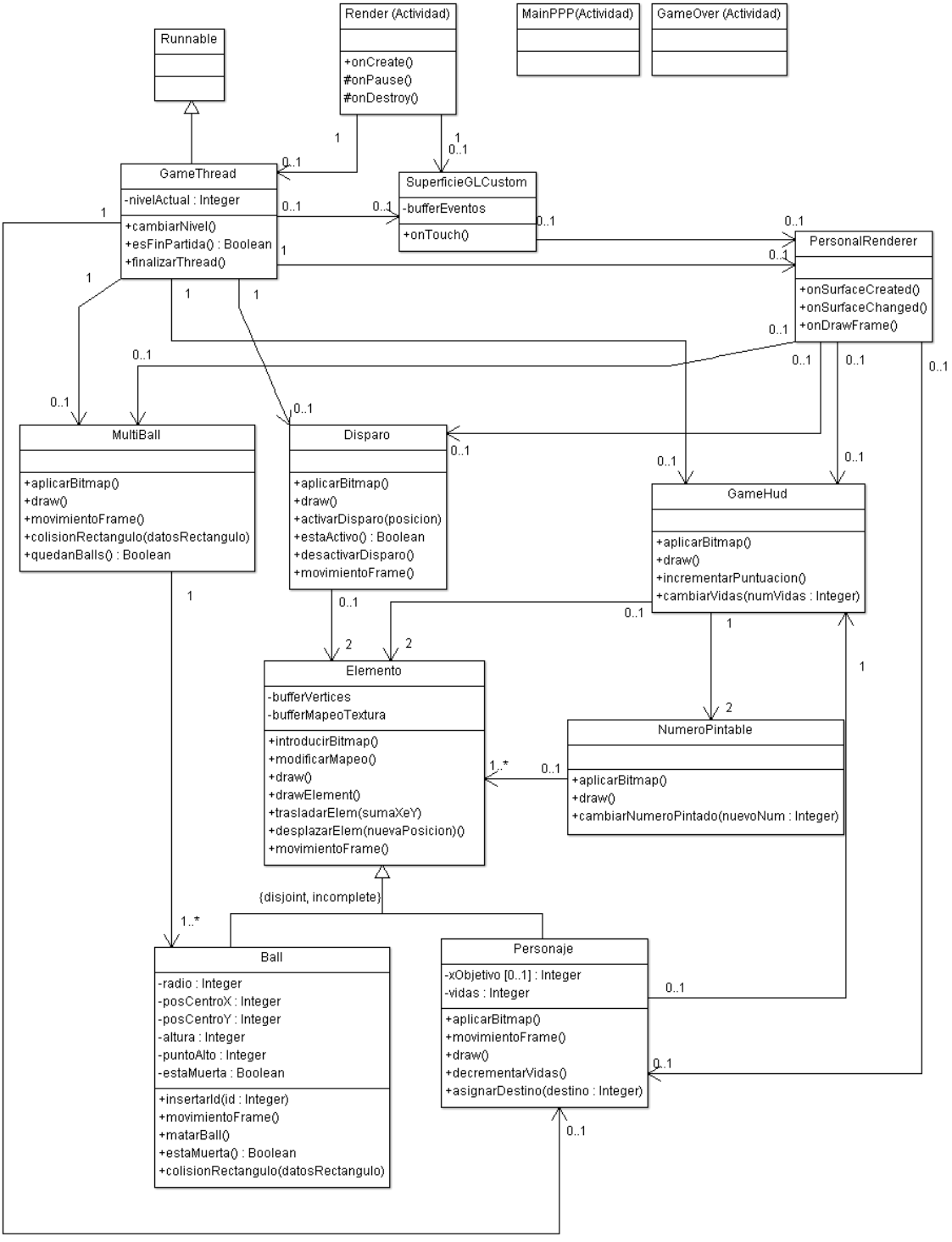


Figura 3.6: Diagrama de clases. Los prefijos en atributos y operaciones tienen los siguientes significados: (-) Atributo/Operación privada, (+) Atributo/Operación pública, (#) Atributo/Operación Protegida.

Se han incluido las navegabilidades que han surgido durante el desarrollo.

Primero de todo, podemos observar que tanto la actividad de inicio *MainPPP*, como la de final de partida *GameOver* únicamente constan de una clase (que hereda de *Activity*). En estas clases únicamente se carga una imagen (clase *ImageView*) con un receptor de entrada, que en caso de activarse realiza una acción fija. (ver figura 3.7)

Sin embargo la actividad *Render* es diferente, esta es la actividad que se encarga de la ejecución del juego.

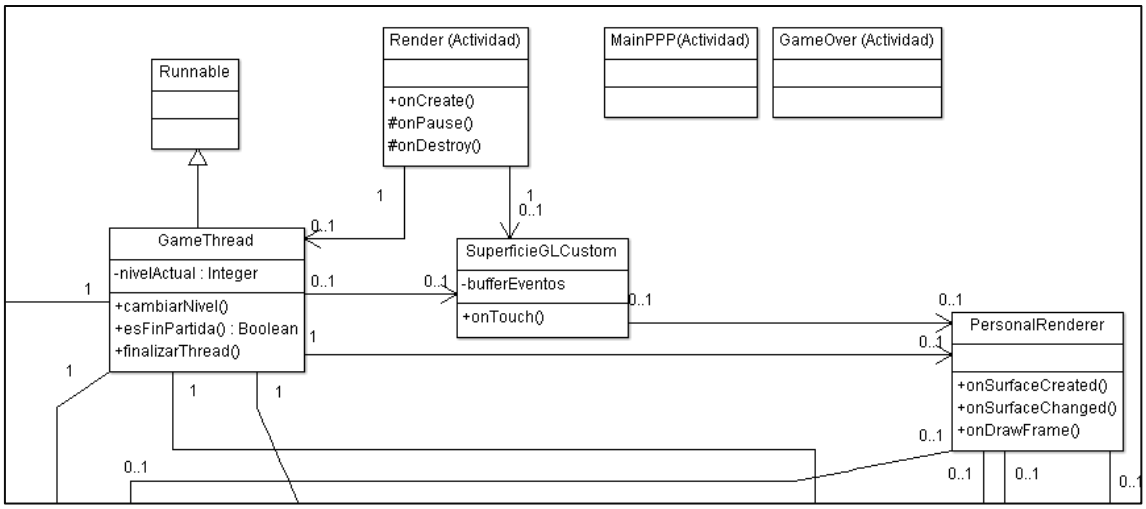


Figura 3.7: Ampliación del diagrama de clases. Visualización de Actividades y superficie de pintado OpenGL.

Podemos ver una clase llamada *SuperficieGLCustom*, al igual que otra clase llamada *PersonalRenderer* que cuelga de la primera (figura 3.7). Estas clases tienen que ver con el mecanismo utilizado por Android para llevar a cabo la unión entre la interface y la librería OpenGL (para más detalles ver el apartado 6 del capítulo 2).

En este videojuego el mecanismo concreto es el siguiente:

1. En primer lugar disponemos de una clase que hereda de la clase *View* (para más información sobre la estructura de las interfaces ir al apartado 2 del capítulo 2) llamada *GLSurfaceView*. Esta *view* consiste en una superficie sobre la cual se insertará el *render* extraído del proceso de *renderizado* llevado a cabo por OpenGL.

   Por tanto, disponemos de una superficie, un elemento más de entre los elementos que Android predefine para crear interfaces, encima de la cual se pinta nuestro *render*.

   En caso de querer tratar la entrada que un usuario pueda enviar sobre dicha superficie deberemos extender la clase (en nuestro caso *SuperficieGLCustom*) definiendo los eventos a tratar, tal y como se realiza en cualquier otra *view*.

2. Ahora solo queda resolver una duda ¿Dónde introducimos nuestro código con las llamadas a OpenGL?

   Para hacerlo, la superficie tiene una Inner Class (en Java, una clase que está dentro de otra [16]) llamada *Renderer*, la cual extendemos y implementamos. En nuestro caso nace la subclase *PersonalRenderer*. Una instancia de esta clase deberá ser introducida en la superficie antes de empezar a pintar.

   Por último, la clase que hereda de *Renderer* debe implementar tres operaciones a las que el sistema ira llamando automáticamente, que son las que contendrán las llamadas a la librería OpenGL (ver apartado 6 del capítulo 2 para más detalle).

Por otro lado observamos una clase llamada *GameThread* que hereda de *Runnable*. Este no es más que un *thread*, concretamente el *thread* que se encarga de la ejecución del bucle del videojuego (ver apartado 6 del capítulo 2). Es importante que este bucle se lleve a cabo en un *thread* aparte, ya que en caso contrario bloquearía totalmente el UI Thread impidiendo una respuesta inmediata.

Si nos fijamos en el resto del diagrama observaremos la estructura que se ha creado para facilitar el añadir elementos nuevos a nuestro videojuego. El centro de todo es la clase Elemento (figura 3.8), dicha clase se encarga de, dado un vector con los vértices de un rectángulo, un mapa de bits con la textura a pintar sobre el triangulo y un vector con el mapeo de posición de textura a vértices del rectángulo, gestionar todo el pintado de dicho rectángulo.

Para esta gestión dispone de las operaciones *introducirBitmap()* que a partir del mapa de bits introducido ajusta todos los parámetros de la textura y deja nuestro rectángulo listo para ser pintado. Luego, la operación *draw()* es la que se encargará de llevar a cabo dicho pintado. Sin olvidar que tenemos varias operaciones para cambiar el vector de mapeo de coordenadas de textura a coordenadas de vértices y para cambiar la posición del rectángulo.

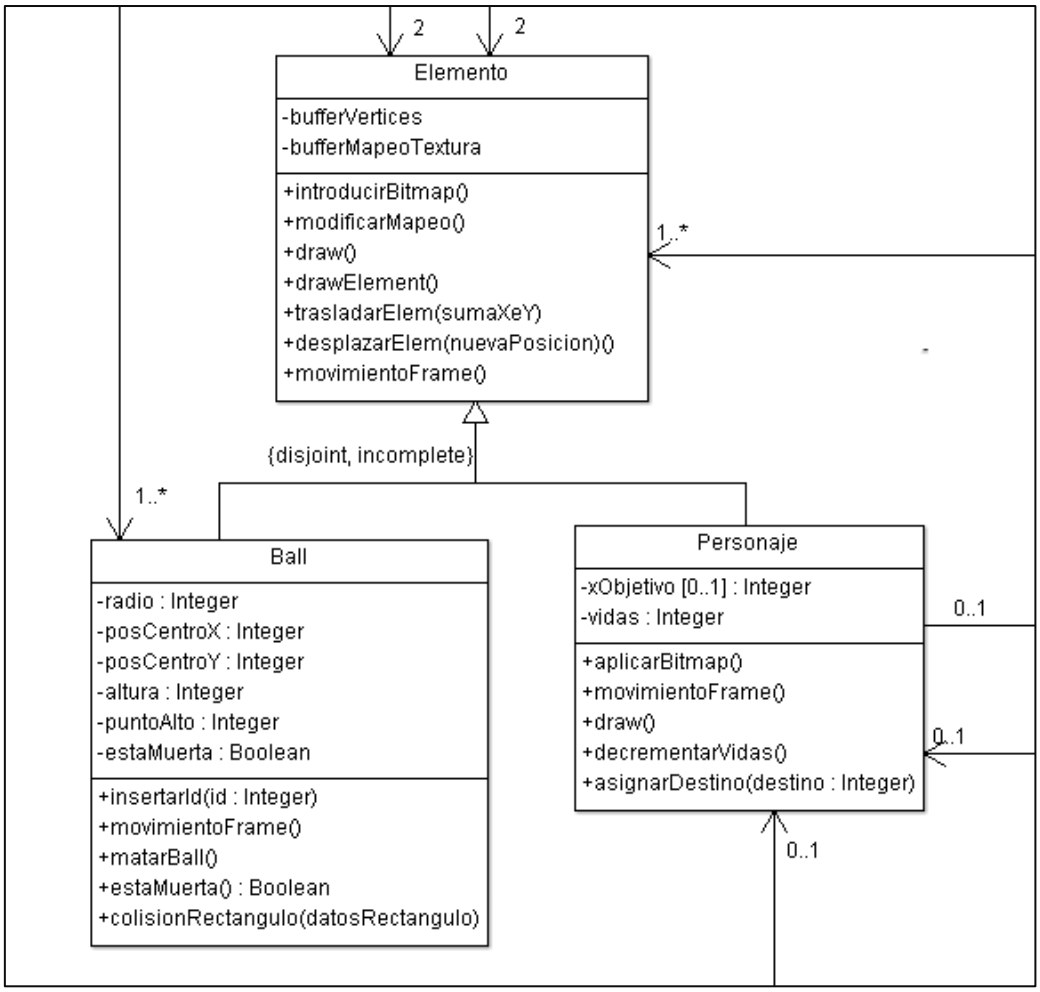


Figura 3.8: Ampliación del diagrama de clases. Visualización de la clase Elemento y sus subclases.

Es importante resaltar que en general todo elemento de un videojuego se acaba pintando a través de un rectángulo, incluso aunque sea una circunferencia, puesto que el área de colisión es independiente del área de pintado y gracias a las transparencias podemos hacer que no se aprecie la forma de la trigonometría real utilizada.

Pues bien, partiendo de esta clase podemos seguir dos caminos, según nos convenga:

- Si el elemento a definir se puede pintar en un solo rectángulo, heredamos de *Elemento* (como ocurre con *Ball* y con *Personaje* en esta aplicación).

  En esta clase se deberá definir una operación *aplicarBitmap()* que se encargue de traer el mapa de bits que nos interese de entre los recursos y enviárselo a la operación de *Elemento introducirBitmap()* donde se preparará para que sea pintado, tal y como queremos. También podemos, como alternativa, hacer una llamada a *insertarId(id: integer)* donde pasamos el identificador correspondiente a la textura ya configurada y preparada (útil para cuando varios elementos comparten una misma textura).

Además, si queremos realizar algunas comprobaciones en el pintado deberemos implementar el método *draw()* y dentro de esta operación llamar a *drawElement()*, definida en *Elemento*, para proceder al pintado del rectángulo. Esta operación, en *Elemento*, solo llama a *drawElement().* El objetivo es que al final todas las subclases llamen a *draw()* para pintar, necesiten comprobaciones adicionales, como el caso de *Personaje*, o no y, por tanto, podamos tratar los elementos de forma genérica desde el GL thread sin entrar a mirar si es de una subclase u otra.

Por otro lado, si el elemento tiene movimiento, deberemos definir la operación *movimientoFrame(),* que se encarga de mover nuestro objeto, teniendo en cuenta que *GameThread* llamará a esta operación una vez por *frame*. El *movimientoFrame()* por defecto (en *Elemento*) es "*no hacer nada*" de esta forma abrimos la puerta a que un conjunto de elementos, futuramente se traten de forma genérica desde el *GameThread*.

Por último, no podemos olvidar definir:

- Las operaciones para gestionar el estado de la subclase, ya sea desde el bucle de juego o desde una clase que agrupe a varios elementos de un subtipo concreto.
- Las operaciones para comprobar colisiones, si es que serán otros elementos los que acudan a esta subclase a comprobar si han colisionado. O las consultoras del área de colisión en caso de que sea al revés. Es importante tener en cuenta que el área de colisión no tiene porque ser igual al área de pintado, este ultimo mantenido por "*Elemento*".

- A veces, en cambio, tenemos un elemento del videojuego que se compone de varias instancias de *Elemento* o de varias instancias de alguna de sus subclases.
  En este caso definimos una nueva clase que representa a un grupo entero de elementos. Esta nueva clase contendrá las operaciones *aplicarBitmap()* y *movimientoFrame()*, pero en este caso lo que harán es encargarse de mover y preparar la textura de todos sus elementos. Adicionalmente, será necesario definir la operación *draw()* para, dentro de ella, pintar también todos los elementos. Y, por último, tampoco nos podemos olvidar de operaciones para cambiar o consultar el estado del grupo y para comprobar colisiones con todo el grupo.
  En esta primera aplicación *GameHud*, *NumeroPintable*, *Disparo* y *MultiBall* son ejemplos de estos grupos.

Como curiosidad *NumeroPintable* (ver figura 3.9), es una clase que mantiene un grupo de "*Elementos*", con la peculiaridad de que estos elementos representan los dígitos de un número. Se tiene una textura con todos los números (del 0 al 9) y, mediante sus métodos, podemos hacer que las texturas pintadas sobre cada rectángulo formen el número de varios dígitos indicado a través del método *cambiarNumeroPintable(nuevoNum: Integer)*. Esta clase nace debido a que OpenGL no tiene implementada ninguna forma de representar números o letras por pantalla, lo que exige que sea el programador el que monte su propio mecanismo para dicha representación. El marcador y el número de vidas restantes del Hud del videojuego se implementan mediante esta clase.

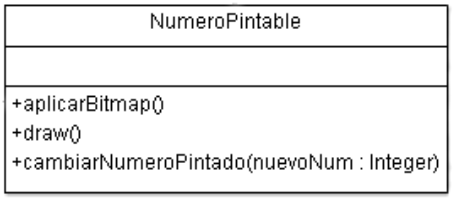


Figura 3.9: Clase *NumeroPintable* del diagrama de clases.

A pesar de haber intentado realizar una interface de desarrollo cambiable, el primer proyecto tenía como objetivo familiarizarse e integrar la mayoría de nuevas herramientas vistas en la fase de estudio del proyecto y los conocimientos previos. Es por esto que la extensibilidad aún puede ser mejorada y este será uno de los objetivos a alcanzar en el segundo proyecto.

Entre las mejoras, la definición de una superclase para todos los grupos, que permitirá referirse a ellos de forma genérica desde el *thread* de *renderizado* o una nueva clase que abstraiga el cambio de *sprite* para simular el movimiento de los elementos dinámicos, son mejoras que no se han llevado a cabo en este primer proyecto por las dimensiones de este.

Por último, cabe decir que, como observamos en el diagrama de clases, desde el *GameThread* se accede a una instancia de *GameHud*, *MultiBall*, *Personaje* y *Disparo*, igual que desde *PersonalRenderer*, debido a que son los elementos que forman parte de nuestro juego y estos se deben mover y pintar.

5. Los tres *threads* que componen el videojuego

Como ya se ha explicado en apartados anteriores, durante la ejecución del videojuego, se mantienen tres hilos de ejecución ejecutándose concurrentemente. En este apartado, aparte de recordar las tareas que lleva a cabo cada *thread* se describe la comunicación entre estos *threads* implementada en "Ping Pang Pung".

Esta estructura es la que se ha decidido implementar después de agrupar los conocimientos adquiridos acerca de la estructura de las aplicaciones para Android, la estructura de un videojuego y el funcionamiento de la librería OpenGL en Android.

Por un lado tenemos el UI Thread, que es el nombre que recibe el hilo de ejecución donde se ejecuta la actividad y donde el sistema operativo avisa de los eventos que el usuario provoca. Este no puede estar bloqueado puesto que esto afectaría a la respuesta de la aplicación ante un evento y al pintado de los elementos que conforman la interface de la aplicación, si se diera el caso, el usuario notaria que la aplicación no responde correctamente. Así que, debido a la naturaleza asíncrona de las aplicaciones para el sistema operativo Android, nos vemos obligados a dejar esta *thread* libre de la carga del bucle principal del videojuego.

Dicho bucle se ejecutará en otro hilo de ejecución, el llamado Game Thread.

La fase que se encarga de la lectura de la entrada por parte del bucle del juego, queda condicionada, por tanto, por la necesidad de que estas entradas sean recibidas en el UI Thread. Para la correcta resolución de los eventos de entrada se deberá llevar a cabo una comunicación entre ambos *threads*, que se detalla en el apartado posterior.

Otro punto que en Android difiere a la hora de ejecutar el bucle del videojuego es el pintado del *frame*. En Android, OpenGL se utiliza a través de la interface definida por la clase *Renderer* y dicha interface se ejecuta en otro *thread* que el sistema automáticamente crea, gestiona durante su vida y elimina cuando sea preciso, automáticamente. Se trata del llamado GL Thread.

En este caso, se ha decidido mantener el pintado automático que por defecto ejerce dicho *thread*, por lo que desde el bucle del videojuego no deberemos activar dicho pintado. Se pinta automáticamente, bloqueando el *thread* donde se ejecute, cosa que gracias a las facilidades que ofrece Android no nos tiene que preocupar.

Como los objetos se comparten entre el Game Thread y el GL Thread, los cambios de posición que se hagan en el Game Thread automáticamente serán recibidos en el GL Thread. Esto constituye uno de los puntos más difíciles a la hora de desarrollar videojuegos para el sistema operativo Android y que más adelante se tratará a fondo.

En el bucle del videojuego, por tanto, únicamente se procesará la entrada y se tratará la lógica y la física.

En la Figura 3.10 podemos ver un esquema que presenta los tres *threads* que conforman el videojuego.

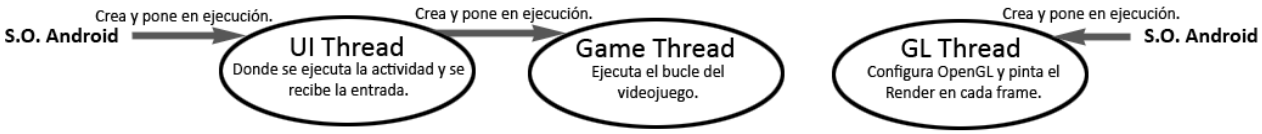


Figura 3.10: Representación de los tres *threads* que se ejecutan concurrentemente. Incluye el responsable de la creación, puesta en marcha y gestión de cada uno de estos.

5.1. Comunicación entre los tres *threads*

Una vez introducidos los tres *threads* que conforman el videojuego vamos a centrarnos en detalle en la interacción que se lleva a cabo entre estos.

5.1.1. Inicialización de los *threads*

En un principio, el sistema operativo pone en ejecución el UI Thread, donde se ejecuta la actividad. Esta creará y configurará el Game Thread, enviándole la instancia de *SuperficieGLCustom* que va a añadir a su *contentView*. Será la clase Game Thread la que se encargue de crear la instancia de *PersonalRenderer* introducirle los elementos que compartirán el Game Thread y el GL Thread y introducirlo dentro de *SuperficieGLCustom*, lo que provocará que el GL Thread comience su ejecución y, por tanto, su pintado.

Por último, la actividad *Render* pondrá en marcha el *thread* con el bucle del juego. Para ello crea un *thread* y dentro del él pone a ejecutar el método *run()* de la clase *GameThread*.

Llegados a este punto ya tenemos los tres hilos de ejecución funcionando (ver figura 3.10).

5.1.2. Comunicación durante la ejecución del videojuego

Durante la ejecución del videojuego la interacción existente entre el GL Thread y el Game Thread se reduce a compartir las clases que contienen los elementos del videojuego (ver figura 3.12). Desde el Game Thread procesamos cada *frame* y actualizamos, por tanto, el estado de cada uno de estos elementos. Mientras que desde el GL Thread se esta accediendo a estos mismos elementos para pintarlos en la posición pertinente.

Como ya se ha comentado es este mecanismo el que ocasiona la mayor dificultad cuando desarrollamos un videojuego para el sistema operativo Android.

Tenemos objetos que son accedidos desde dos *threads* y modificados desde uno de ellos, todo sin un orden predecible. Por tanto, estas clases compartidas entre *threads* deben ser *thread-safe*, es decir deben poderse acceder desde varios *threads* al mismo tiempo imposibilitando que el acceso simultaneo en cualquier orden posible provoque que uno de los accesos se lleve un dato erróneo (o, como mínimo, evitando que el error perdure lo suficiente como para que el usuario se dé cuenta).

En este primer proyecto la mayor problemática la tiene la clase *MultiBall*, puesto que las bolas se crean y se eliminan cuando el usuario las mata. Lo que significa que tenemos un vector que cambia de tamaño. Para asegurar que nunca se acceda a una posición inexistente del vector, se ha optado por no eliminar ninguna bola, en lugar de ello, se indica que están muertas.

Se ha optado por esta solución porque en este videojuego no supone demasiado gasto de memoria mantener durante un tiempo unas bolas que ya no existen en la lógica del videojuego, debido a que el número de elementos que tenemos es muy pequeño.

En cambio, cuando cambiamos de nivel es inevitable que *Multiball* cambie, puesto que aparecen nuevas bolas. En este caso definimos una operación específica que nos guarda el nuevo *Multiball* dentro del *PersonalRender* como un elemento auxiliar y en el momento en que el *PersonalRender* esté preparado para hacer el cambio, lo llevará a cabo, eliminando el antiguo *MultiBall*.

Vamos ahora a tratar la interacción entre el UI Thread y el Game Thread:

Como se ha explicado con anterioridad, las entradas son recibidas en el UI Thread y estas debe llegar al Game Thread para que sean procesadas sobre los elementos del juego.

Esta comunicación se lleva a cabo a través de un buffer donde la operación *onTouch()* del *SurfaceGLCustom* escribe la entrada concreta recibida, cuando el S.O. le hace saber que el usuario ha tocado la pantalla; todo esto desde el UI Thread. Al mismo tiempo el Game Thread al inicio de cada iteración comprueba este buffer para ver si ha llegado un nuevo evento de entrada, en cuyo caso le da respuesta (en este primer proyecto la entrada puede perseguir: o disparar o mover al personaje hacia una posición X dada).

Además, existe otro caso en el cual se precisa una comunicación entre el Game Thread y el UI Thread. Se trata del momento en el que se acaba la partida y la actividad *Render* debe desaparecer, o el momento en el que el usuario decide abandonar el juego aún sin haber perdido todas las vidas que el personaje posee. En ambos casos es necesario tener un mecanismo de comunicación entre ambos hilos de ejecución, para ello se define la variable *finPartida* dentro de la clase *GameThread*. Esta variable puede ser modificada tanto desde el bucle del juego (cuando el personaje se queda sin vidas), en cuyo caso advertiremos a la actividad que se ejecuta en el UI Thread de que debe finalizar su ejecución; o desde la actividad, en cuyo caso estaremos advirtiendo al bucle del juego de que la partida ha terminado y, por tanto, debe cesar su iteración acabando así con el *thread* de juego.

Para finalizar este apartado se presenta a continuación el diagrama de clases indicando, para cada clase, en que *threads* se ejecuta. Además, como la clase Personaje, se ejecuta en dos *threads* al mismo tiempo, se presenta un esquema donde se indica qué operaciones son exclusivas de cada *thread*. Figura 3.11 y 3.12.

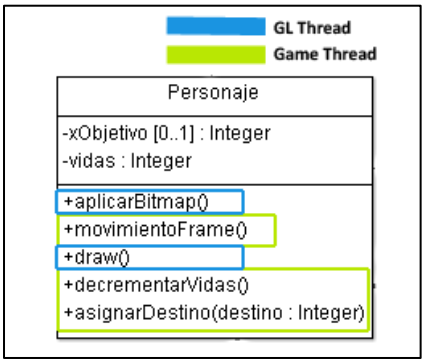


Figura 3.11: Clase Personaje donde se puede apreciar: cuales son las operaciones que se ejecutan desde el *GL Thread* (las que preparan la textura y las que pintar el elemento). Y cuáles son las operaciones que se ejecutan desde el *Game Thread* (las que actualizan el estado del elemento en cada frame y las que cambian el estado ante un suceso específico).

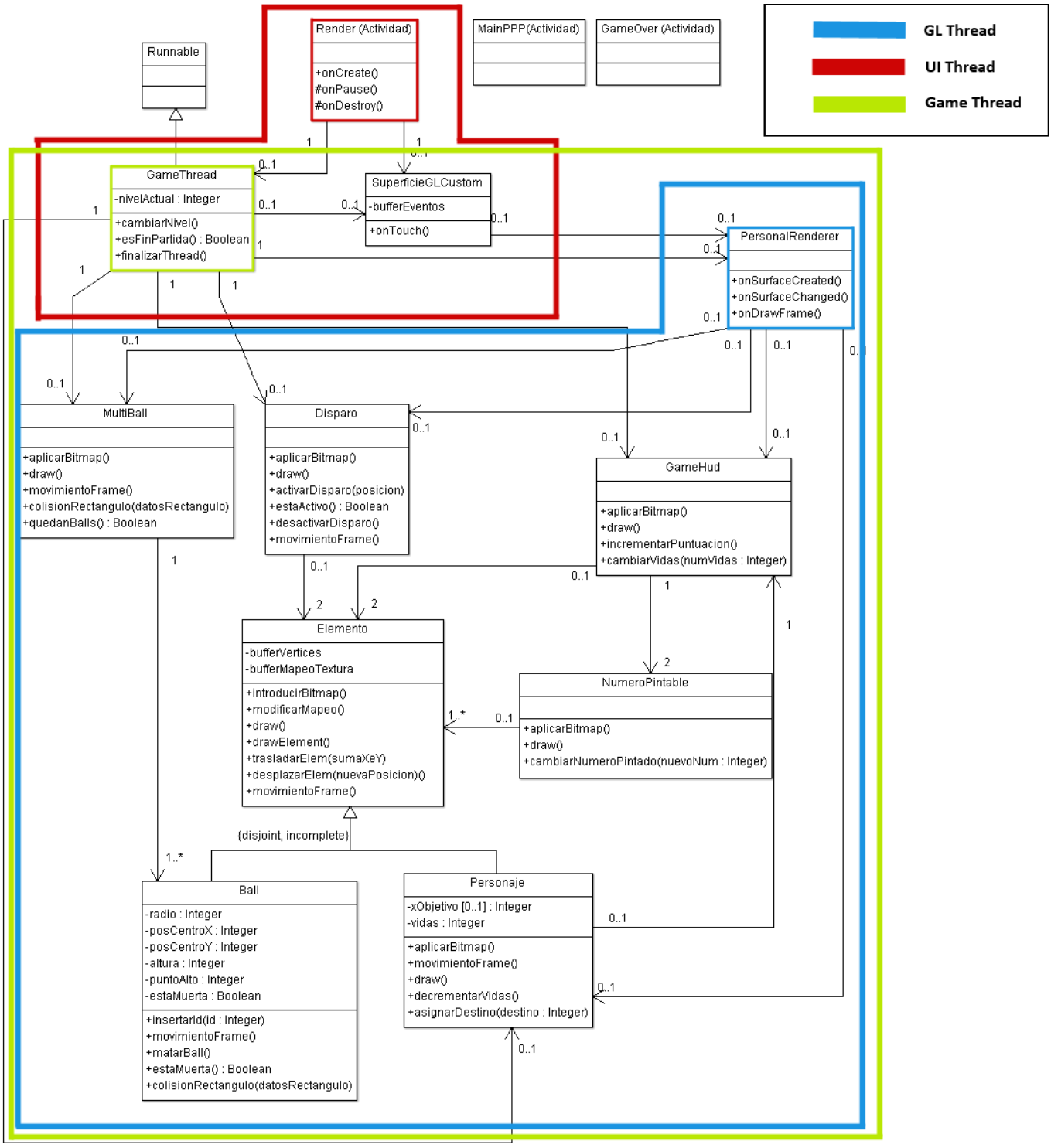


Figura 3.12: Diagrama de clases donde se indica, para cada clase, desde que *threads* es accedida. Además para cada *thread* hay una clase cubierta con su color, indicando que esta es la clase principal, que dispone del método *run()* o de los métodos a los que el S.O. accede (*callbacks*), a partir de los cuales se accede al resto de clases.

6. Diferentes espacios de pintado

Hemos visto las diferentes clases que componen el proyecto, así como los *threads* dentro de los cuales se ejecutan estas clases simultáneamente, pero aún nos queda una dimensión más por tratar:

Se trata de la separación entre el espacio de pintado de Android y el espacio de pintado de OpenGL.

6.1. El espacio de pintado Android

Cuando utilizamos el mecanismo que incluye Android para definir interfaces trabajamos en un espacio en el cual las clases que componen este mecanismo se asocian formando un árbol. Como ya vimos, tenemos las *views*, hojas del árbol que visualizan un elemento concreto con una función concreta, y las *viewGroups*, el nodo padre y los nodos intermedios del árbol, que agrupan un conjunto de *views* asignando unas propiedades comunes a todas estas (ver apartado 2 del capítulo 2 para más información).

Estas *views* ocupan un espacio concreto dentro del espacio de pintado Android, que se puede especificar tanto en pixeles (medida no absoluta y que, por tanto cambiará dependiendo de la densidad de pantalla del dispositivo donde la interface se represente) como en pixeles independientes de la densidad (lo que significa que el tamaño físico será siempre más o menos el mismo).

El espacio de pintado Android ocupa toda la pantalla, aunque a medida que bajamos el árbol de *views*, cada *view* tiene disponibles solo el espacio reservado por su padre (leer apartado 2 del capítulo 2 para más información

6.2. Espacio de pintado OpenGL

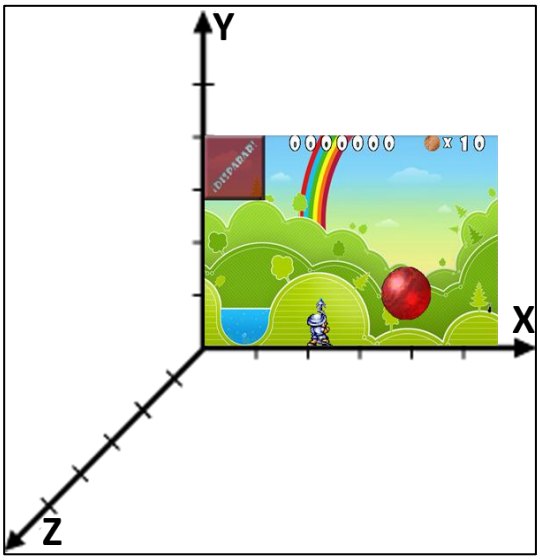


Figura 3.13: Espacio de pintado OpenGL. En este caso se ha pintado un *frame* del videojuego.

Sin embargo, el espacio de pintado OpenGL funciona de forma diferente. Aquí situamos los polígonos (a bajo nivel únicamente triángulos) sobre un espacio en tres dimensiones (ver figura 3.13), utilizando unas medidas independientes del espacio de pintado Android. Además, en este espacio también se debe definir el tamaño de una "ventana" que será la que nos permitirá ver una parte de lo que hay en este espacio y hacer una "foto" de lo que a través de ella se ve, es decir, construir el *render*. Las medidas utilizadas en el espacio de pintado OpenGL solo tienen equivalencia con las medidas del espacio de pintado Android cuando se genera el *render* y este se presenta a través de una *view* que tiene asignada un área concreta en el espacio de Android.

Teniendo en cuenta que nuestro juego se mueve dentro del espacio de pintado OpenGL nos interesa poder procesarlo todo con medidas en este espacio, y que sea una clase externa al bucle del juego y a la interface de pintado (clase "*PersonalRenderer*") la que se encargue de hacer la conversión a la hora de pintar o de tratar la entrada que introduce el usuario. Esto es lo que se ha pretendido en los proyectos realizados, aunque es inevitable que PersonalRenderer tenga alguna implicación con el espacio de pintado Android, pues implementa una interface que trabaja con ambos espacios.

6.3. Comunicación entre los dos espacios

Como hemos visto con anterioridad en el S.O. Android tenemos una *view* que pinta una superficie en el área que le toque dentro del espacio de pintado de Android y dentro de esta superficie se pinta el *render* resultante de OpenGL. Para definir qué superficie de la pantalla ocupa el *GLSurfaceView* simplemente debemos definir una interface tal y como Android dicta e introducir el tamaño de esta *view*. En nuestro caso el *GLSurfaceView* ocupa **toda** la pantalla (ver figura 3.14). Cabe decir que sea cual sea esta área no nos afecta a la hora de trabajar con el proceso y pintado del videojuego, puesto que se trabaja sobre el espacio OpenGL que, como hemos dicho, utiliza unas medidas propias e independientes.

Por otro lado, debemos definir las dimensiones del *render* resultante de la aplicación de OpenGL, que será el que se pinte sobre la superficie antes nombrada. Se ha decidió que si este *render* tiene unas proporciones diferentes a las de la superficie, el *render* no se debe deformar, así que se ampliará hasta que choque con los bordes de la pantalla a lo largo o a lo ancho y el espacio que no quede cubierto con el *render* se verá del color de fondo escogido en OpenGL.

Las dimensiones del *render* en el espacio de pintado Android se definen desde la llamada *onSurfaceChanged()* de la clase *PersonalRenderer*, que como ya hemos dicho será llamada por el sistema operativo cuando la orientación del dispositivo cambie o cuando se inicialice la instancia de dicha clase. En concreto, desde aquí debemos definir: el área del espacio OpenGL que se verá en el *render*, el punto desde donde mirará la cámara que define OpenGL, y por tanto el ángulo desde donde se sacará la "foto" (*render*) y luego el tamaño de esta foto y su posición en la *GLSurfaceView*.

Con estas configuraciones, desde la operación *onDrawFrame()* de la propia clase ya podremos pintar únicamente teniendo en cuenta las dimensiones en el espacio OpenGL. Y, cómo no, el bucle del juego, que posiciona los elementos para que se pinten en un lugar determinado, también deberá trabajar con el espacio OpenGL.

Peor aún queda un punto importante no explicado. Que ocurre cuando se recibe una entrada que, en caso de ser un evento táctil traerá consigo una posición X e Y dentro del espacio Android. Pues bien, en este caso, es el *SurfaceGLCustom* el que debe realizar la conversión, desde el método que recibe el evento, antes de introducir esta información en el buffer que después leerá el bucle del juego. De esta forma conseguimos esta abstracción del espacio de pintado Android, que nos facilita el desarrollo y nos permite hacer cambios en la visualización general del *render* resultante sin tener que modificar prácticamente nada.

Figura 3.7: Espacio de pintado Android. Muestra como se sitúan los dos ejes dentro de este espacio. Así como un ejemplo en el que se pinta un *frame* del videojuego que nos ocupa, que en este caso solo ocupa una parte de la superficie.

Como conclusión, en la aplicación los únicos métodos que trabajan con el espacio de pintado Android son:

- La operación *onSurfaceChanged(),* que se ejecuta en el GL Thread y que trata en todo momento el espacio Android de forma genérica: recibe una anchura y una altura correspondiente a las dimensiones de la superficie y a través de estas dimensiones decide el área del espacio de pintado OpenGL que aparecerá en el render, el tamaño del *render* dentro de la superficie Android en la que se pinta y la posición en la superficie. Por tanto, este método trabaja en ambos espacios de pintado.

- Y algunas clases de las que se ejecutan en el UI Thread como son: la actividad, que tiene una interface Android con la que tiene que tratar, y las *views*, que son las que reciben los eventos de entrada y definen sus tamaños. En concreto, en este primer proyecto, la *view SuperficieGLCustom* es la que recibe los eventos que se dan sobre la superficie y, por tanto, la X e Y del pulsado dentro del espacio de pintado Android.

Es el Game Thread, por tanto, el único *thread* que no tiene absolutamente ningún trato con el espacio de pintado Android.

Capítulo 4 - Segundo videojuego

Una vez que el primer videojuego estaba desarrollado, ya se habían puesto en práctica los conocimientos adquiridos durante toda la etapa de aprendizaje y análisis, así que era hora de realizar el segundo videojuego.

Esta segunda aplicación es mucho más compleja y completa que la primera, principalmente gracias a la experiencia adquirida y a un mayor tiempo dedicado a su desarrollo.

El hecho de disponer de mayor tiempo para el desarrollo, entre otras cosas, ha permitido aplicar todas las conclusiones extraídas del análisis de antecedentes llevado a cabo (referencia a apartado 7 del capítulo 2).

A continuación se presenta toda la información referente a este segundo videojuego llamado “Terrestrial Blast!”.

1. Descripción del videojuego

A diferencia del videojuego anterior, “Terrestrial Blast!” parte de una idea original, lo que significa que no está inspirado directamente en ningún otro videojuego existente.

La creación de un videojuego original es un reto que nos puede aportar una gran satisfacción cuando vemos que el concepto de videojuego que hemos imaginado se ha hecho realidad. Pero también tiene sus desventajas, pues desarrollarlo es más complejo que desarrollar un calco de otro videojuego existente, ya que el concepto de juego imaginado debe ir madurando con el tiempo. Y es que, probablemente, acabemos viendo que algunas ideas que tuvimos inicialmente no son tan buenas como pensábamos en un principio y deben ser sustituidas por otras.

Sin ir más lejos, en “Terrestrial Blast!” inicialmente existía la idea de introducir una serie de ataques espectaculares, pero más tarde se vio que había tres factores que hacían que estas habilidades no fuesen una buena idea: primeramente, la potencia de un móvil es limitada y, por tanto, la implementación de estos ataques sacrificaba clientes, que por tener un móvil inteligente poco potente no podían ejecutarlo; además, el alcance del proyecto no permitía dedicar el tiempo que la implementación de estas habilidades requería y más, teniendo en cuenta, que eran habilidades cuya cualidad residía en el espectáculo visual y no en el aspecto *jugable*; y es que, el último factor, era la creencia de que los usuarios de dispositivos móviles buscan una buena *jugabilidad* en lugar de ataques muy visuales pero poco interactivos (conclusión extraída del análisis de antecedentes).

1.1. Mecánica del videojuego

*“*Terrestrial Blast!*”* es un juego que se puede enmarcar en tres géneros diferentes: en primer lugar, es un videojuego de plataformas, pues debemos recorrer escenarios, salteando obstáculos y investigando como completar cada nivel; por otro lado, también se puede enmarcar dentro del género *shoot’em up*, ya que nuestro protagonista debe desplazarse volando por el aire y disparando a los enemigos que se encuentren a nuestro alcance; y, por último, también es un juego de puzles y habilidad puesto que para completar los niveles deberemos completar unos pequeños acertijos que también requieren de cierta habilidad.

Esta mezcla de géneros desemboca en un juego donde el objetivo es completar una serie de niveles con una serie de peculiaridades:

En “Terrestrial Blast!*”* cada nivel tiene una puerta, el objetivo del nivel es abrir la puerta y entrar por ella, lo que nos llevará a completarlo. Para abrir la puerta se deberán resolver los acertijos que el nivel esconde, los cuales están basados en diferentes puzles cuyas piezas están distribuidas por el escenario. Deberemos recolectar todas las piezas y encajarlas en el lugar que les corresponde para conseguir así que la puerta se abra.

1.2. Manejo por parte del usuario

Una de las conclusiones más importantes que extrajimos del análisis de antecedentes, era que los videojuegos para teléfonos móvil deben adaptar y, en la mayoría de casos simplificar, los géneros de videojuegos que conocemos en otras plataformas, ya que cuando jugamos en el teléfono móvil normalmente estamos cubriendo la necesidad de distracción en pequeños momentos muertos a lo largo del día. Por tanto, buscamos un videojuego que se pueda pausar en cualquier momento y cuyas partidas no requieran de mucho tiempo de dedicación seguida.

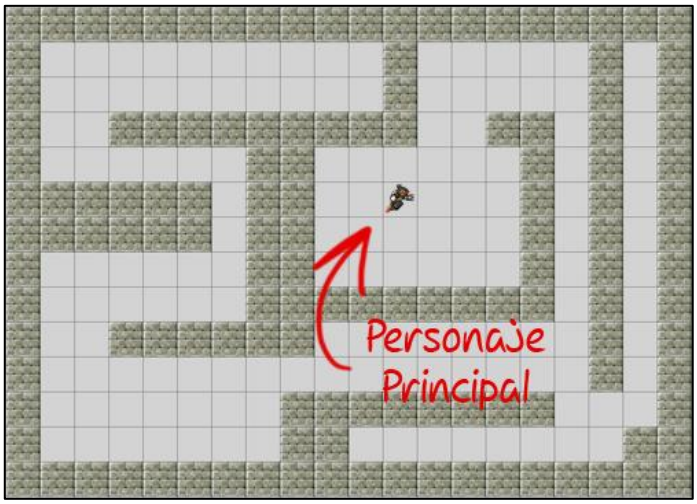


Figura 4.1: Visión global del nivel 1 del videojuego "Terrestrial Blast!" Aparece señalado el personaje principal para poder apreciar el tamaño del nivel.

En este videojuego, se ha perseguido este objetivo y es por esto que "Terrestrial Blast!" permite la pausa en cualquier momento. A pesar de ello no es aconsejable utilizar la pausa más que para pequeños intervalos de tiempo en que debamos responder a una llamada o hacer alguna acción breve, ya que el S.O. Android puede verse sin memoria y matar la aplicación en cualquier momento, lo que provocaría la pérdida de nuestro avance en el nivel pausado, como ya vimos en el apartado 2 del capítulo 2.

Además, el videojuego tiene niveles cortos por tal de que el jugador que quiera simplemente jugar en franjas pequeñas de tiempo pueda completar varios niveles sin problemas y no tenga que estar jugando más del tiempo que desea (ver Figura 4.1).

Pero más allá de adaptar el videojuego a las necesidades que los jugadores persiguen, ante todo, para que sea fácil y divertido jugar al videojuego, debemos implementar correctamente los controles. Por ello, debemos tener en cuenta la tecnología que la mayoría de los dispositivos móviles de hoy en día ponen a disposición del usuario.

Actualmente los teléfonos inteligentes suelen basar su interacción en una pantalla táctil y en una serie de sensores cada vez más avanzados. El problema es que si adaptamos juegos diseñados para jugarse con botones físicos simplemente transformando estos botones en botones táctiles (ver Figura 4.2), si el juego requiere habilidad y precisión probablemente jugarlo sea mucho más incomodo de lo que lo era con el anterior método de control. Esto es debido a que la simulación de botones a

través de una pantalla táctil no suele ser ni mucho menos igual de cómoda y precisa que la disposición de botones físicos.

Es por este motivo que a la hora de desarrollar un videojuego para un teléfono móvil debemos, ante todo, intentar que la *jugabilidad* y los controles sean cómodos y estén adaptados para dispositivos móviles.

Este es el motivo por el cual "Terrestrial Blast!" ofrece la mezcla de géneros que antes hemos indicado. Analizando juegos de plataformas, que simplemente se han calcado de otras plataformas a los teléfonos móvil, descubrimos que una cruceta táctil es incomoda y imprecisa, de forma que, por ejemplo, saltar en el momento justo, para esquivar un obstáculo se hace difícil simplemente por el mecanismo de control utilizado. Por este motivo se decidió mezclar el género de plataformas con el *shoot'em up* e implementar un movimiento en ocho direcciones a través del sensor de orientación.

Figura 4.2: Captura de pantalla del emulador, de la videoconsola Super Nintendo, SuperGNES Lite, disponible para el S.O. Android. Presenta un ejemplo extremo de mapeo de crucetas y botones, originalmente físicos, en la pantalla táctil.

El sensor de orientación nos ahorra tener que pintar una cruceta táctil en la pantalla. Dicho sensor, a través del acelerómetro y la brújula digital permite que con pequeñas inclinaciones del dispositivo nos podamos desplazar.

Inicialmente no se sabía si este control iba a dar un resultado realmente bueno, ya que en los ejemplos encontrados y probados se le había dado uso en juegos puramente *shoot'em up*, que no tenían nada de plataformas. Pero una vez implementado y probado podemos decir que el sensor da una buena respuesta y, aunque los usuarios al principio se suelen quejar de algo de dificultad, una vez acostumbrados acaban estando satisfechos con el manejo.

Por tanto, podemos concluir que el sensor de orientación es una buena alternativa a la cruceta táctil y que no genera tanta frustración, pues en ningún momento se tiene que mirar la cruceta y recolocar correctamente los dedos por posibles fallos de colocación. Y además, permite que el Hud del videojuego quede más despejado: En "Terrestrial Blast!" únicamente es necesario presentar el botón de disparo y la vida del jugador.

Por tanto, tenemos más rango de visión de los escenarios y de los objetos que se encuentran en ellos.

Esto no significa que el sensor de orientación no tenga inconvenientes, también tiene algunos como son: la imprecisión en el movimiento o la obligatoriedad de mantener el dispositivo con una inclinación constante mientras no queramos que el personaje se mueva.

Por último, el motivo por el cual se han añadido elementos del género de puzles es, nuevamente, para aprovechar los controles que el dispositivo nos brinda, pues hacer uso de la pantalla táctil para mover objetos del escenario hasta la posición que les corresponde es novedoso y divertido, pues se trata de una nueva forma de interactuar con el videojuego, diferente a los clásicos botones y más intuitiva.

Por tanto, concluimos que “Terrestrial Blast!” es un videojuego pensado para satisfacer las necesidades de distracción breve qué suele provocar que un usuario juegue con el móvil y para aprovechar los mecanismos de interacción que el dispositivo móvil nos brinda, consiguiendo de esta forma un control cómodo, intuitivo y divertido.

Los niveles cortos y los controles cómodos, junto con unos niveles que exigen gran habilidad provocarán que el juego “enganche” al jugador. Y por tanto, conseguirán que este no abandone el videojuego al poco tiempo de instalarlo.

Para conocer más detalles acerca de la mecánica del videojuego, así como de sus controles y del Hud de este, es aconsejable leer el primer anexo, donde se detalla en profundidad todo lo que a nivel de usuario es útil conocer.

1.3. Diseño artístico, audio e historia

Este es un apartado que se ha cuidado bastante en el videojuego.

Recordemos que el último videojuego, como se trataba de un juego con un tiempo de desarrollo corto, no se añadió audio y, además, carecía de historia. En este videojuego, en cambio, se han cuidado bastante ambos apartados al igual que el diseño artístico, pues ya vimos en el análisis de antecedentes que era muy importante que nuestro videojuego tuviese personalidad propia.

Vamos primero a hablar de las músicas y sonidos del juego, que se han escogido con bastante mimo y dedicación. En el videojuego "Terrestrial Blast!" tenemos una pista de audio musical para todos los menús y otra pista de audio musical para cada nivel. Las pistas de los niveles, han sido seleccionadas teniendo en cuenta que sean acordes con lo que al usuario le espera en el nivel en cuestión.

Además, tenemos efectos de sonidos en menús, para facilitar la interacción del usuario con ellos. Se ha comprobado que la diferencia entre botones silenciosos y botones que emitan sonidos cuando se pulsen es notable. En el caso de introducir sonidos, se mejora la experiencia de usuario a la hora de navegar por los diferentes menús. Si por el contrario estos sonidos no se introducen, el usuario no sabe si realmente ha presionado un botón, lo que le provoca una sensación de incertidumbre, la cual llega a hacer la espera entre pantallas mucho más larga.

Por otro lado, también los diferentes objetos de los niveles presentan sonidos. Estos sonidos son muy importantes, pues dan personalidad a cada uno de los objetos (enemigos, puertas, etc.) que tiene el nivel. Además, avisan al usuario de determinados sucesos, como pueden ser: un disparo emitido por el enemigo, el impacto de un ataque del usuario en un enemigo, etc.

Cabe decir que, por motivos de tiempo y de alcance del proyecto, las músicas y sonidos no son propios, sino que han sido extraídos de diferentes webs con músicas y sonidos libres [17].

En cuanto al diseño artístico, de igual forma se ha tratado cuidadosamente, aunque nuevamente con las limitaciones que el propio proyecto exige. Es por ello que los elementos gráficos también se han extraído de diferentes fuentes y después se han modificado creando composiciones o aportándoles diferentes matices que les han dado un sentido y una personalidad dentro de la historia creada para el videojuego. Por ejemplo, al personaje principal se le ha añadido un jet pack para que su textura fuese coherente con el personaje que la historia del videojuego describe; además, después se ha diseñado una versión más detallada de este para que apareciera en los menús del videojuego (ver Figura 4.3).

Figura 4.3: Comparación entre el personaje principal que controlamos en los niveles (a la derecha) y la versión más detallada de este, que ilustra los menús del videojuego (a la izquierda).

En ambos podemos apreciar el equipo que el personaje del videojuego posee: cañón de una sola mano, jet pack y un traje especifico para la aventura.

Pero, ¿Cuál es la historia que da sentido a todos estos recursos visuales y auditivos?

Pues bien, la historia ideada para el videojuego se presenta a continuación:

"Al inicio de nuestra historia, el protagonista se encuentra en su casa tranquilamente, sin ser consciente de la terrible amenaza que se cierne sobre su planeta.

Un ser malvado ha descubierto que bajo la corteza terrestre del planeta se encuentra un mineral muy preciado que, vendido en el mercado negro, puede generarle riqueza ilimitada. Así que ha construido una base secreta en el interior de una cueva (ver Figura 4.4) desde donde piensa taladrar toda la corteza terrestre, para poder llegar hasta el mineral y extraerlo en su totalidad.

Como es conocedor de las terribles consecuencias que esta acción provocará sobre el planeta; el cual se agrietará provocando hundimientos que acabarán con ciudades, bosques y gran cantidad de vida; ha protegido la cámara desde donde llevará a cabo la maléfica tarea construyendo un laberinto de niveles.

En estos niveles ha posicionado gran cantidad de monstruos y maquinas hostiles que no dudarán en acabar con cualquier intruso que se les acerque. Además, por si el intruso consiguiera destruirlos, ha construido unos puzles para cada nivel, los cuales habrá que completar para poder avanzar en el laberinto de niveles, pues los mecanismos de apertura y cierre de la puerta de cada nivel están vinculados a estos puzles.

Cuando las acciones de nuestro maleante comienzan a llevarse a cabo, la casa y la ciudad donde habita el protagonista de la historia son destruidas por los duros terremotos que la maquinaría, necesaria para taladrar la corteza del planeta, provoca. Como es lógico la destrucción de la ciudad donde el protagonista había vivido desde su nacimiento provoca su enfado.

Decidido a proteger su planeta, el personaje que controlaremos se fabrica un equipo con el que poder entrar en la base secreta y llegar hasta la cámara donde se encuentra el taladro, con el objetivo de detener al malvado ser que está provocando tal destrucción.

El equipo fabricado consta de un jet pack, con el que nuestro personaje podrá planear y volar, evitando así pisar el suelo inestable, el cual se podría hundir fácilmente. Pero aun hay más, puesto que, para acabar con los monstruos y maquinas que prevé que defenderán la base secreta, este fabrica un cañón capaz de disparar bolas de energía, el cual puede controlarse con una sola mano. Además, el protagonista de la historia lleva años entrenando un control de materia a través de la mente que le permite transportar algunos objetos relativamente pesados enviando órdenes sensoriales.

Una vez preparado, el personaje principal se enfunda las armas fabricadas, junto con un traje elaborado especialmente para facilitarle la misión (ver Figura 4.3) y se embarca en la aventura de salvar su planeta. "

Como vemos, a través de esta historia damos un sentido a las habilidades que el usuario podrá utilizar y a la forma de desplazarse del personaje que controlamos. Otros elementos como escenarios, niveles o contrincantes también son dotados de un porqué. Al mismo tiempo damos un objetivo por el que completar todos los niveles. Así como un objetivo para cada nivel, que es el de resolver un puzle para pasar al siguiente nivel.

Cabe decir que la historia no ha podido ser expresada a través de escenas de animación o de textos en los niveles, puesto que el alcance de este proyecto cubría la parte más técnica del videojuego, dando a este apartado menor importancia. Aun así, como ya se ha ido explicando, todos los recursos, desde el icono de la aplicación, hasta la pantalla de "*Nivel Completado*" están diseñados para transmitir esta historia de una forma visual y sonora.

Figura 4.4: Momento en el que nuestro personaje se adentra dentro de la cueva donde el maleante de la historia ha construido su base secreta. Aparece como fondo de una de las pantallas del menú del videojuego.

Como ya concluimos en el análisis de antecedentes, el hecho de haber dotado a los personajes del videojuego de una personalidad, así como el haber dado un porqué a las acciones que llevamos a cabo en este, provocará que "Terrestrial Blast!" sea más atractivo para el jugador.

1.4. Compatibilidad con las diferentes versiones de Android

En este caso, se ha decidido desarrollar el videojuego "Terrestrial Blast!" para que funcione en la versión de Android 2.1 y en las versiones superiores. Deja de ser compatible, por tanto, con las versiones 1.5 y 1.6, con las cuales el videojuego "Ping Pang Pung" sí que lo era.

Esta pérdida de compatibilidad se debe principalmente a un factor, estas versiones tan antiguas de Android no tienen soporte *multitouch*, lo que obliga a tener que implementar dos tipos de soporte, el *multitouch* y el mono táctil, o solo el segundo.

En este caso, la *jugabilidad* del videojuego adaptada a una pantalla mono táctil empeoraría la experiencia *jugable*, debido a que, el usuario deseará poder disparar y arrastrar objetos del escenario al mismo tiempo. Además, la implementación de dos tipos de control también llevaría un tiempo de desarrollo adicional. Y, por último, no podemos olvidar que, a medida que pasa el tiempo, las versiones inferiores a la versión 2.1 son cada vez más minoritarias, siendo solo el 2.3% de usuarios de Android los que utilizan estas versiones a día 3 de Noviembre de 2011, según indica el estudio llevado a cabo por Google [18].

Se tienen en cuenta, por tanto, las conclusiones extraídas del análisis de antecedentes, que valoraban la opción de restringirse a la versión 2.1 en caso de que la *jugabilidad* no fuese adaptable a pantallas mono táctiles.

Aun así, hay que aclarar que, aunque el juego este preparado para funcionar en las versiones 2.1 y superiores de Android, este ha sido probado únicamente en la versión 2.2. Esto se debe a que Android 2.2 es la versión que tienen los dispositivos utilizados para el desarrollo. Con lo cual únicamente se asegura su funcionamiento en esta versión.

Esto no quita que con total probabilidad, versiones superiores puedan ejecutarlo a la perfección, pues Android ha sido desarrollando teniendo en mente el mantener siempre la compatibilidad de aplicaciones antiguas con versiones futuras del S.O. Y, por otro lado, durante el desarrollo se ha tenido cuidado de utilizar funciones de la API de Android que estén en la versión 2.1 y, por tanto, que no se hayan añadido en versiones más recientes, con lo cual su funcionamiento en esta versión del S.O. también debería ser el correcto.

1.5. Herramientas utilizadas

En cuenta a las herramientas utilizadas para el desarrollo de la aplicación, una vez más se ha hecho uso del lenguaje JAVA y del entorno de desarrollo que proporciona el *plugin* ADT para Eclipse, el cual nos permite crear proyectos que utilicen la API de Android.

El motivo por el cual se han utilizado estas herramientas, se debe a que son unas herramientas robustas, bien documentadas y desarrolladas por la propia Google. Lo que nos garantiza que ante cualquier problema tendremos suficiente información para resolverlo y que, en caso de tratarse de errores del propio entorno, la empresa responderá ante ellos y los solucionará sacando nuevas versiones.

En lo que respecta a la librería de gráficos, también se ha vuelto a hacer uso de OpenGL ES 1.0, pues la versión 2.0 no es necesaria para el apartado visual de "Terrestrial Blast!", el cual que no requiere del uso de *shaders*.

El hecho de que sea una librería multiplataforma nos permitirá portar la parte gráfica del videojuego a otras plataformas de forma casi directa, si lo deseamos. Además, nuevamente nos encontramos ante una librería bien documentada y ampliamente utilizada, sobre la que es fácil, por tanto, encontrar información y solucionar posibles problemas.

No se ha utilizado una librería que funcione un nivel por encima de OpenGL, puesto que, como se detallo en el apartado 8 del capítulo 2, lo que tenemos disponible para el S.O. Android son más bien motores que no solo cubren la parte gráfica. Además, por lo general, no suelen estar igual de bien documentados que las herramientas escogidas. Por si fuera poco, raramente son multiplataforma y gratuitos al mismo tiempo.

Tampoco hemos de olvidar que trabajar a bajo nivel nos va a permitir controlar mejor el modo en que dibujamos los gráficos y, por tanto, seguramente podremos hacer una aplicación más eficiente.

Pero no solo se ha hecho uso de OpenGL ES 1.0, pues algunas partes se han implementado haciendo uso de la versión 1.1, la cual ofrece funciones que pueden mejorar la eficiencia de la aplicación, como es el caso de los VBOs [15]. Como no todos los dispositivos son compatibles con OpenGL ES 1.1, se ha implementado un selector que a la hora de utilizar características exclusivas de la versión 1.1 se pregunta si el dispositivo donde se está ejecutando es compatible con estas características. En caso de que no exista compatibilidad, la aplicación lleva a cabo sus funciones haciendo uso únicamente de las características ofrecidas por la versión 1.0 de OpenGL ES.

En cuanto a las físicas, esta vez, se ha hecho uso de una librería externa, se trata de la librería Box2D. El motivo principal por el que se ha escogido esta librería es por ser multiplataforma, lo que nos asegura la fácil portabilidad a otros sistemas operativos. Esta portabilidad es importantísima debido a la variedad de sistemas operativos móviles que hay hoy en día, tal y como concluimos en el análisis de antecedentes.

Gracias al uso de Box2D, se ha podido desarrollar un videojuego que cuenta con un mundo físico muy configurable y bastante parecido a la realidad. El hecho de usar esta librería ha ahorrado mucho trabajo en la implementación de físicas y ha permitido crear un juego mejor en menor tiempo. Además, se trata de una librería que está muy bien documentada y sobre la que es fácil trabajar.

Cabe decir, que no se ha hecho uso de la versión Java de Box2D, pues, como ya vimos se explicó con anterioridad, las librerías de físicas en Java requieren bastantes recursos. Esto hace que, en un dispositivo móvil, no funcionen lo suficientemente bien como para utilizarlas en videojuegos que quieran mantener un gran número de objetos físicos en pantalla, pues los recursos son limitados.

Por este motivo se ha hecho uso de la librería LibGDX. Esta librería, como ya vimos, nos permite utilizar las funciones de Box2D escritas en el lenguaje C a través de unas cabeceras en Java. LibGDX, por tanto, hace uso de la posibilidad de ejecutar, en el sistema operativo Android, código en lenguaje C de forma nativa.

Por tanto, gracias a esta librería podemos hacer uso de la versión C de Box2D sin tener que tocar dos lenguajes durante el desarrollo del videojuego, pues únicamente necesitaremos utilizar el lenguaje Java.

Por último, también se ha utilizado el lenguaje XML, pues este videojuego se ha diseñado pensando en conseguir una aplicación altamente configurable a nivel de entidades y niveles del videojuego. El objetivo ha sido tener pocas clases Java genéricas y muy configurables, en lugar de muchas clases Java poco configurables.

El lenguaje XML es perfecto para configurar las diferentes clases Java, pues nos permite trabajar a un nivel de abstracción de la tecnología mayor, lo que hace que el trabajo de configuración sea más intuitivo.

El tema de los niveles de abstracción y del uso del lenguaje XML se abordará con más detalle en apartados futuros.

Por tanto, podemos concluir que se ha tenido muy en cuenta la portabilidad del videojuego a otras plataformas móviles, puesto que este ha sido el criterio principal a la hora de escoger las herramientas citadas. Además, también se ha valorado mucho la eficiencia que las diferentes librerías utilizadas pueden aportar, pues ambas funcionan a bajo nivel, lo que implica total control sobre la optimización del videojuego. El último

factor importante en la elección ha sido la correcta documentación, la cual evita el quedarse estancado con un problema por culpa de una falta de claridad en la documentación de las herramientas.

2. Especificación del videojuego: Casos de Uso

A continuación se presenta la especificación del videojuego "Terrestrial Blast!" Como el videojuego es realmente grande, se ha decidido introducir en esta memoria únicamente las partes más importantes de la especificación. Es por esto que no se introduce el modelo conceptual de los datos, puesto que es preferible explicar directamente el diagrama de clases de diseño, el cual se detallará en futuros apartados. De igual forma, aunque los diagramas de secuencia del sistema y los contratos de las operaciones del sistema son muy útiles para desarrollar el videojuego, en esta memoria solo conseguirían confundir al lector.

Presentamos por tanto, el diagrama de casos de uso y la descripción detallada de cada uno de estos casos de uso:

2.1. Diagrama de Casos de Uso

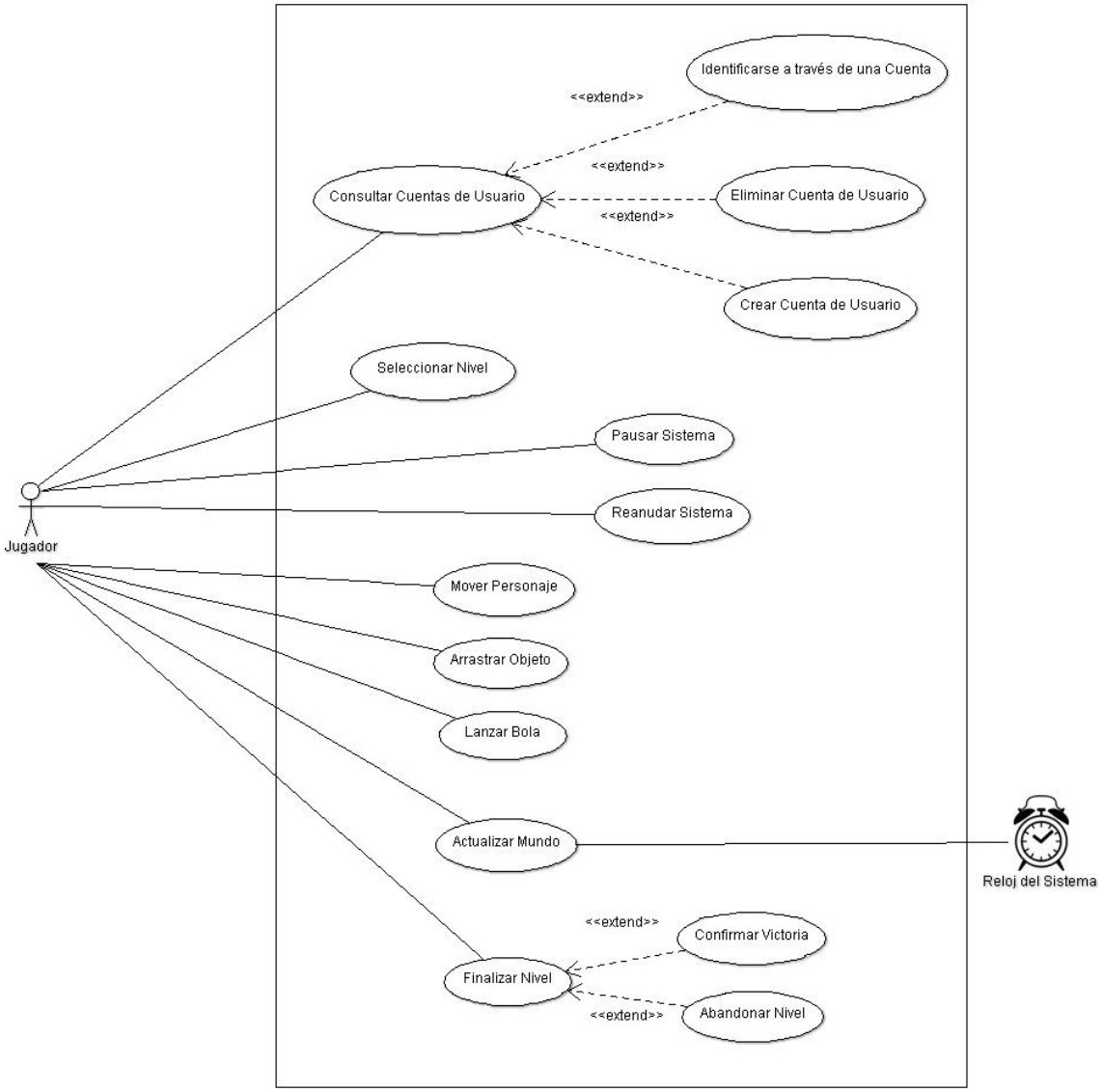


Figura 4.5: Diagrama de casos de uso de la aplicación "Terrestrial Blast!".

2.2. Definición de los Casos de Uso

Caso de Uso: Consultar Cuentas de Usuario

Actores primarios: Jugador

Contexto: El jugador, que no se encuentra identificado a través de una cuenta de usuario, decide que quiere consultar la lista de cuentas de usuario disponibles en el sistema.

Escenario principal de éxito:

1. El jugador solicita al sistema la lista de cuentas de usuario disponibles.
2. El sistema presenta la lista al jugador (con un máximo de 3 cuentas) junto con los bloques de "cuenta de usuario" libres, si es que hay alguno.
3. Si el jugador quiere identificarse a través de una de las cuentas existentes en el sistema: escenario alternativo "Identificarse a través de una Cuenta de Usuario".

   Si, por otro lado, el jugador quiere eliminar una de las cuentas existentes: escenario alternativo "Eliminar Cuenta de Usuario".

   En último lugar, si hay algún bloque de cuenta libre, el jugador puede querer crear una nueva cuenta de usuario: escenario alternativo "Crear Cuenta de Usuario".

Escenario alternativo (<<extend>>): Identificarse a través de una Cuenta de Usuario.

1. El jugador selecciona la cuenta de usuario a través de la cual se quiere identificar.
2. El sistema identifica al jugador a través de esta cuenta y se mantiene a la espera de nuevas solicitudes por parte del jugador identificado.

Escenario alternativo (<<extend>>): Eliminar Cuenta de Usuario.

1. El jugador indica la cuenta que desea eliminar.
2. El sistema pide una confirmación al jugador.
3. El jugador podrá recular, en cuyo caso se salta al paso 2 del escenario principal de éxito. En caso contrario confirmará la eliminación de la cuenta.
4. El sistema eliminará la cuenta y toda la información referente a ella.
5. Salto al paso 2 del escenario principal de éxito.

Escenario alternativo (<<extend>>): Crear Cuenta de Usuario.

1. El jugador indica al sistema que quiere crear una Cuenta de Usuario nueva y le envía el nombre de usuario que identificará dicha cuenta.
2. El sistema registra una cuenta de usuario con ese nombre de usuario e inicializará todas sus configuraciones y puntuaciones.
3. Salto al paso 2 del escenario principal de éxito.

Escenario alternativo: El jugador decide que no quiere hacer ninguna acción. (Paso 3 del escenario principal de éxito).

1. El jugador indica al sistema que no quiere realizar ninguna de las acciones disponibles sobre las Cuentas de Usuario.
2. El sistema vuelve a estar a la espera de recibir nuevas peticiones por parte del jugador sin identificar.

Caso de Uso: Seleccionar Nivel

Actores primarios: Jugador

Contexto: El jugador se encuentra identificado a través de una cuenta de usuario, no está jugando ningún nivel en este momento y decide que quiere listar los niveles disponibles.

Escenario principal de éxito:

1. El jugador identificado indica al sistema que quiere jugar un nivel.
2. El sistema presenta a este jugador identificado los niveles que ha desbloqueado y, por tanto, que puede jugar.
3. El jugador identificado indica al sistema el nivel que quiere jugar (debe ser un nivel existente y al que su Cuenta de Usuario tenga acceso).
4. El sistema prepara el nivel y lo presenta al jugador. Comenzará además a llevar a cabo su simulación en tiempo real, que durará hasta que el nivel finalice o el personaje principal sea reducido.

Escenario alternativo: El jugador no quiere jugar ninguno de los niveles disponibles (Paso 3 del escenario principal de éxito).

1. El jugador indica al sistema que no quiere jugar ningún nivel.
2. El sistema pasa a estar a la espera de nuevas peticiones por parte del jugador que ahora vuelve a estar sin identificar.

Caso de Uso: Mover Personaje

Actores primarios: Jugador

Contexto: El jugador (identificado a través de una cuenta de usuario) está jugando un nivel en este momento y decide que quiere desplazar al personaje por el escenario.

Escenario principal de éxito:

1. El jugador indica con qué intensidad quiere que se mueva su personaje (indicando una fuerza) y hacia qué dirección, al sistema.

2. El sistema se asegura de que, en las siguientes ejecuciones del caso de uso "Actualizar Mundo", se simule un movimiento del personaje hacia la dirección que el jugador ha indicado. El personaje principal estará moviéndose hacia dicha dirección hasta que se vuelva a ejecutar este mismo caso de uso indicando una dirección de movimiento diferente.
   La simulación del nivel no cesará hasta que el nivel finalice o hasta que el personaje sea reducido.

Caso de Uso: Arrastrar Objeto

Actores primarios: Jugador

Contexto: El jugador está jugando un nivel en este momento (identificado a través de una cuenta de usuario) y decide que quiere arrastrar uno de los objetos inertes que se encuentran en el escenario y que tienen la propiedad de poderse arrastrar.

Escenario principal de éxito:

1. El jugador indica el objeto que quiere arrastrar.

2. El sistema se asegura de que este objeto sea interpretado como objeto agarrado durante las siguientes ejecuciones del caso de uso "Actualizar Mundo".

3. El jugador indica la siguiente coordenada a la que se debe dirigir el objeto (coordenadas próximas puesto que se trata de una simulación de arrastrar un objeto).

4. El sistema se asegura de que en futuras llamadas a "Actualizar Mundo" se vaya simulando un movimiento del objeto agarrado previamente hacia el punto recibido.

5. Se salta al paso 3 hasta que el jugador decide dejar de agarrar el objeto y lo comunica al sistema.

6. El sistema se asegura de que las siguientes ejecuciones de “Actualizar Mundo” calculen la velocidad y dirección con la que el jugador estaba arrastrando el objeto y simulen el efecto de soltar este objeto a dicha velocidad.
   La simulación del nivel no cesará hasta que el nivel finalice o hasta que el personaje sea reducido.

Caso de Uso: Lanzar Bola

Actores primarios: Jugador

Contexto: El jugador está jugando un nivel en este momento (identificado a través de una cuenta de usuario) y decide que quiere que el personaje principal lance una de sus bolas de energía.

Escenario principal de éxito:

1. El jugador indica al sistema que el personaje lance una bola de energía.

2. El sistema crea una bola de energía y se asegura de que, en futuras ejecuciones del caso de uso “Actualizar Mundo”, se simule su movimiento de forma realista, a la par que se continúan simulando el resto de elementos del mundo físico.
   La simulación del nivel no cesará hasta que el nivel finalice o hasta que el personaje sea reducido.

Caso de Uso: Actualizar Mundo

Actores primarios: Jugador

Contexto: El jugador se encuentra jugando un nivel (identificado a través de una cuenta de usuario) y quiere que el sistema simule en tiempo real dicho nivel.

Escenario principal de éxito:

4. El reloj avisa al sistema de que ya ha pasado el tiempo que dura un frame desde la última vez que se ejecutó este caso de uso.

5. El sistema actualiza los elementos del mundo teniendo en cuenta el tiempo que ha pasado desde la última actualización.

6. El sistema presenta al jugador los nuevos datos actualizados para que, a ojos de este el mundo tenga vida.

   * Para conseguir simular esta vida es preciso que este caso de uso se lleve a cabo continuamente (*frame* a *frame*) mientras un nivel este ejecutándose. Y aunque al mismo tiempo se estén ejecutando otros Casos de Uso. La única excepción se dará cuando se ejecute el caso de uso “Pausar Sistema”, momento en el cual se dejará de activar este caso de uso en cada *frame* hasta que se lleve a cabo “Reanudar Sistema”.

Caso de Uso: Finalizar Nivel

Actores primarios: Jugador

Contexto: El jugador (identificado a través de una cuenta de usuario) se encuentra jugando un nivel y quiere finalizar este nivel ya sea a mitad del nivel, o porque su personaje ha sido abatido (Fin del juego) o porque ha completado el nivel.

Escenario principal de éxito:

1. El jugador indica al sistema que quiere finalizar el nivel.

2. Si el jugador ha completado con éxito el nivel: extensión Confirmar Victoria.
   Si, por el contrario, el jugador se ha quedado sin vidas: extensión Confirmar Fracaso.
   Por último, si el jugador se encuentra a mitad del nivel o sin vidas para continuar: extensión Abandonar Nivel.

Escenario alternativo (<<extend>>): Confirmar Victoria

1. El sistema registra el máximo nivel completado por el jugador identificado, lo que puede llevar a desbloquear un nuevo nivel *jugable*, asociado a la cuenta de usuario que el jugador identificado está utilizando, y espera a que el jugador le indique qué quiere hacer

2. El jugador podrá elegir entre: seleccionar un nivel cualquiera, caso de uso "Seleccionar Nivel"; repetir el nivel jugador, caso de uso "Seleccionar Nivel" paso 4; o jugar el siguiente nivel (si es que existe un siguiente nivel), caso de uso "Seleccionar Nivel" paso 4.

Escenario alternativo (<<extend>>): Confirmar Fracaso.

- El sistema le indica al jugador identificado que ha fracaso y, por tanto, no ha completado el nivel.

- El jugador puede repetir el nivel, caso de uso "Seleccionar Nivel" paso 4, o seleccionar un nivel cualquiera de entre los que tiene disponibles, caso de uso "Seleccionar Nivel".

Escenario alternativo (<<extend>>): Abandonar Nivel.

1. El sistema pasa directamente al paso 2 del caso de uso "Seleccionar Nivel".

Caso de Uso: Pausar Sistema

Actores primarios: Jugador

Contexto: El jugador, que puede estar en mitad de cualquier caso de uso o de cualquier interacción con el sistema, decide que quiere pausar la comunicación que está llevando a cabo para continuar más tarde por el mismo punto en el que se encuentra ahora.

Escenario principal de éxito:

1. El jugador indica al sistema que quiere que este se pause.

2. El sistema se pausa. Pero antes, guarda su estado para que en el futuro el jugador pueda continuar a partir del mismo punto en el que lo dejo.

3. El sistema no podrá comunicarse ni recibir ninguna petición por parte del jugador hasta que no se ejecute el caso de uso "Reanudar Sistema".

Caso de Uso: Reanudar Sistema

Actores primarios: Jugador

Contexto: El jugador quiere retomar el estado en el que dejo pausado al sistema, para continuar comunicándose con este a partir del punto en el que lo dejo. Solo se podrá ejecutar este caso de uso si el último caso de uso que se llevo a cabo en el sistema fue "Pausar Sistema".

Escenario principal de éxito:

1. El jugador indica al sistema que quiere retomar el dialogo con este.

2. El sistema se reanuda con el estado guardado en el momento en que se llevo a cabo la pausa.

3. Salto al paso del caso de uso que se había interrumpido debido a la pausa. En caso de que no se estuviera en mitad de un caso de uso, el sistema estará esperando alguna petición por parte del jugador (manteniendo, como ya se ha dicho, el estado que tenía antes).

3. Componentes de la aplicación

Cuando diseñamos una aplicación para el sistema operativo Android, esta se compone de diferentes componentes, clasificados en varios tipos. En este apartado se presentarán los componentes que componen el videojuego “Terrestrial Blast!”.

Como ya se explicó en el apartado 3 del capítulo 3, es el Diagrama de Componentes el que nos permite ver todos los componentes que conforman una aplicación. Además, al ser las actividades que en él se presentan equivalentes a las *vistas* de las aplicaciones clásicas, también podemos ver todos los caminos de navegación que el usuario puede tomar para navegar a lo largo de las diferentes pantallas. (ver Figura)

En “Terrestrial Blast!”, podemos ver que únicamente se utilizan componentes del tipo Actividad (*Activity*), tal y como pasaba en el videojuego anterior. Pues no son necesarios componentes de otro tipo.

Pasamos a continuación a describir brevemente el cometido de cada actividad, así como algunos datos que no se pueden apreciar simplemente observando el diagrama:

- Actividad *Inicio*: Se trata de la actividad que aparece en pantalla cuando el usuario decide ejecutar la aplicación “Terrestrial Blast!” Es simplemente una imagen de presentación con: el título del videojuego, el personaje protagonista y el nombre del autor.

- Actividad *SelectUsuario*: Segunda actividad que visualizaremos, en ella se nos presentan los diferentes usuarios que el sistema mantiene en su base de datos.

  Tenemos tres huecos (o slots) de usuario disponibles y la actividad nos permite, para cada hueco: crear un usuario que ocupe este hueco, si es que ninguno lo está ocupando ya; borrar al usuario que este ocupando el hueco, en caso de que haya algún usuario ocupándolo; o identificarnos con el usuario en cuestión.

  Para identificarnos simplemente tenemos que presionar sobre la etiqueta con el nombre de usuario, entonces se nos mostrará la actividad de selección de nivel.

  Para crear un usuario nuevo presionaremos sobre la etiqueta de un hueco vacío, el cual, al estar vacío, en lugar de nombre tendrá el mensaje “*Crear Usuario*”.

Si, por el contrario, queremos eliminar a un usuario ya existente, debemos presionar sobre el icono con forma de cruz, que tiene disponible cada uno de los huecos con un usuario asignado. Entonces, antes de eliminar al usuario, por tal de que confirmemos la eliminación, nos aparecerá el siguiente fragmento de actividad:

  - Fragmento de Actividad - Dialogo de Confirmación: Android permite definir lo que llamamos un fragmento de Actividad, que viene a ser un trozo de la interface de esta actividad, el cual funciona de forma más o menos independiente.

    En este caso, el dialogo de confirmación que se muestra a la hora de eliminar un usuario ha sido implementado a través de un fragmento de actividad, por tal de conseguir así una mayor compatibilidad con las versiones más modernas de Android, pues el resto de alternativas para implementar los diálogos han sido ya depreciadas.

    En cuanto a la interacción con el jugador, este simplemente deberá decidir si está realmente seguro de querer eliminar al usuario en cuestión y presionar el botón que corresponda con su decisión.

- Actividad *CrearUsuario*: Actividad que se nos presenta cuando queramos crear un usuario nuevo.

  La actividad nos pide que introduzcamos el nombre que el usuario a crear tendrá y se encarga de su creación cuando presionamos sobre el botón *Crear Usuario*.

  Si el nombre introducido no es correcto, a la hora de presionar el botón *Crear Usuario*, se nos mostrará un error advirtiéndonos de que ya existe un usuario con ese nombre o de que la casilla donde se debe insertar el nombre está vacía, según el caso.

- Actividad *Selector*: Esta actividad se nos presenta una vez que ya nos hemos identificado con un nombre de usuario concreto y nos muestra los niveles del videojuego disponibles.

  En caso de que haya más niveles de los que caben en los seis huecos presentados, aparecerán unas flechas que nos permitirán avanzar entre páginas de niveles (las cuales tendrán todas un máximo de 6 niveles). Eso sí, puede haber niveles que no estén aún desbloqueados, pues para que un nivel se desbloquee se debe completar el nivel anterior.

  Para distinguir los niveles bloqueados de los desbloqueados, tenemos que fijarnos en el icono que identifica al nivel: los niveles bloqueados tienen un icono especial donde se presenta un candado, mientras que los niveles desbloqueados tienen un icono que indica el número del nivel en cuestión.

Si presionamos sobre el icono de un nivel desbloqueado se pasará a ejecutar dicho nivel.

- Actividad *Nivel*: Se trata de la actividad que ejecuta cada uno de los niveles *jugables* del videojuego. Y es que utilizando una misma actividad diseñada genéricamente para ejecutar todos los niveles conseguimos reutilizar el máximo de código posible.

  Esta actividad se encarga de pintar los diferentes *renders* que se generan a través de OpenGL, así como de recibir e interpretar las entradas del usuario, por tal de que este pueda controlar al personaje principal del videojuego a través de dichas entradas.

- Actividad *FinNivel*: Esta es la actividad que se presenta tanto cuando completamos un nivel, como cuando fracasamos en su ejecución. Ambos avisos se unifican en una misma actividad porque tienen prácticamente todo el código en común y, de esta forma, podemos reutilizar dicho código.

  La diferencia principal entre las dos facetas de la actividad es el fondo y el botón "siguiente", botón que según el caso se encontrará disponible o no.

  En el diagrama de componentes a esta actividad se le asocian dos pantallas, estas son las dos facetas que la actividad tiene, y las flechas que salen de cualquiera de ellas se aplican a las dos facetas por igual. Solo es necesario tener en cuenta que el botón *siguiente* puede no estar disponible.

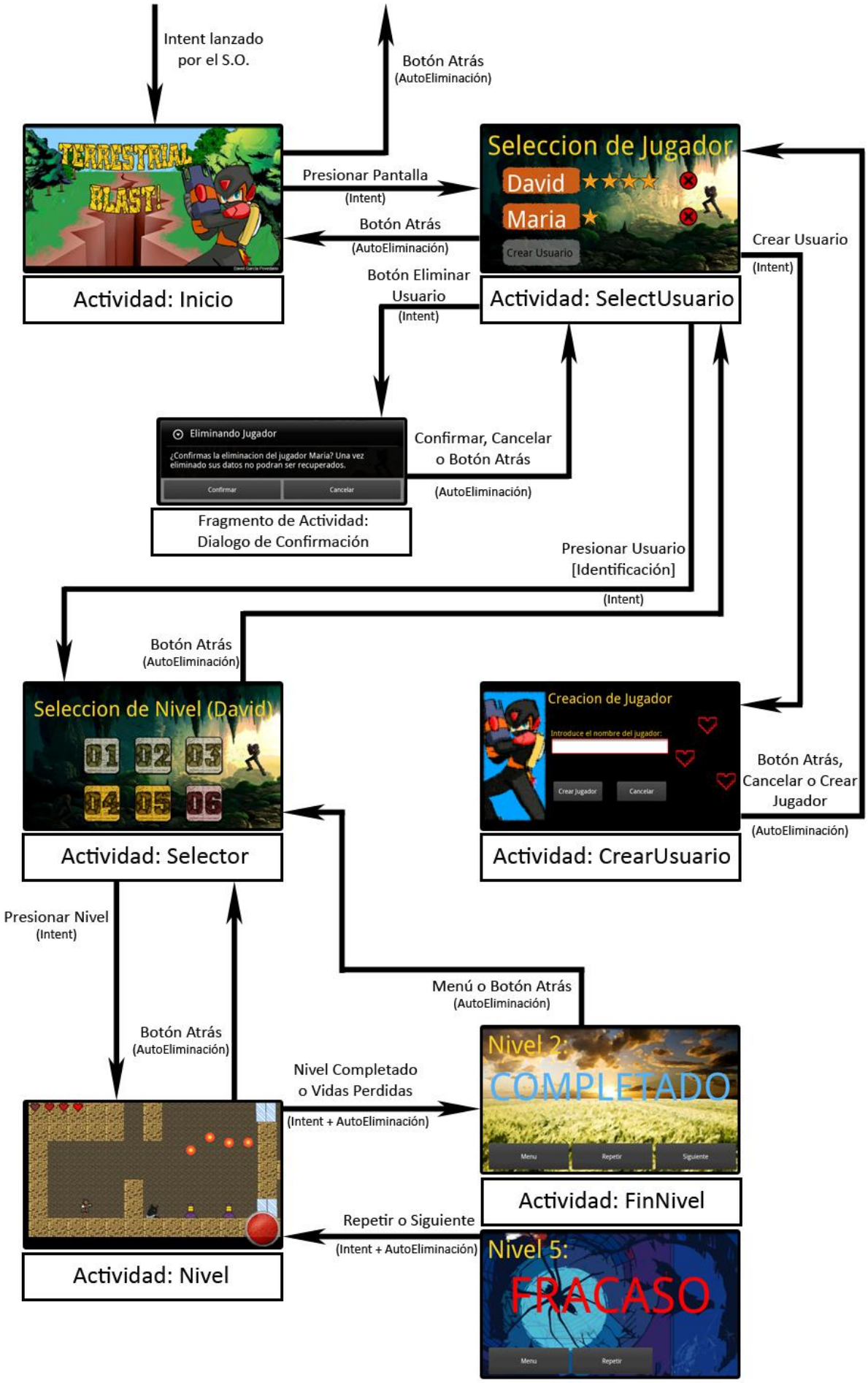


Figura 4.6: Diagrama de Componentes (o Mapa de Navegación) de la aplicación “Terrestrial Blast!”.

Una vez descritas las actividades, vamos a centrarnos en el dialogo e interacción que hay entre ellas:

Si comparamos este diagrama de componentes con el del videojuego anterior, veremos que “Terrestrial Blast!” presenta un diagrama bastante más complejo, pues tiene muchas más actividades. Cuando tenemos un diagrama de componentes más complejo, debemos hacer especial hincapié en cómo pasamos de una actividad a otra (dato que se encuentra entre paréntesis en cada flecha que nos dirige de una actividad a otra).

Como podemos observar en el diagrama, no creamos una instancia de actividad nueva cada vez que queremos acceder a una actividad concreta, pues, como veremos a continuación, hay varios motivos por los cuales no es aconsejable tener más de una instancia de la misma actividad en memoria principal durante la ejecución de una aplicación:

- Desde el punto de vista interno de la aplicación:
    - El tener varias instancias de una misma actividad puede provocar que el estado de los datos del sistema no sea coherente. Pues si la actividad no está especialmente preparada para ello, podría ser que una instancia de la actividad hiciera unas modificaciones y, posteriormente otra instancia, que previamente había sido creada, hiciera otras modificaciones a partir de unos datos ya anticuados, que nos llevarán a solapar acciones ya confirmadas.
    - Además, siempre se consume más memoria principal si tenemos un mayor número de instancias en la pila de instancias de actividad de la aplicación. Pila que podría llegar a tener un número absurdo de instancias si el usuario navegara durante un rato largo por los menús de la aplicación y se estuviesen creando instancias nuevas continuamente.

- Desde el punto de vista del usuario aún son de mayor peso los motivos por los cuales el tener varias instancias de una actividad es perjudicial:

  - El motivo más importante tiene que ver sobre el concepto de aplicación que un usuario de Android debería tener.

    Para el usuario, cada aplicación es una pila de pantallas, una detrás de otra. Y a través del botón Atrás este puede eliminarlas en orden inverso a como se han creado. Por tanto, si un menú concreto se encuentra en una posición de esta pila, luego no puede aparecer en una posición más arriba duplicado, por fuerza, cuando volvamos a ver este menú las instancias que se encontraban sobre este en la pila, deberán haber sido eliminadas. Si no lo hacemos así confundiremos al usuario.

    Este concepto también nos obliga a tener cuidado cuando una actividad crea a otra, pues si la actividad creadora no debe ser visible nunca más para el usuario, esta debe auto eliminarse después de crear la instancia de una nueva actividad. En caso contrario, el usuario podría volver a llegar a la actividad creadora a través de la tecla Atrás. En el videojuego, esto ocurre con la actividad Nivel, pues esta debe desaparecer cuando se acaba la ejecución de la pantalla inmediatamente después de crear la instancia de la actividad *FinNivel*.

  - En segundo lugar, tenemos un problema que puede ser importante en según qué aplicaciones. Diferentes instancias de una misma actividad no comparten el mismo estado, por tanto, una actividad que se dejó con un estado concreto, no coincide con el estado con el que se crea una nueva instancia de la misma actividad, y por tanto, el usuario no tiene la sensación de que le estemos presentando la misma pantalla.

  Nótese que hay casos excepcionales donde la convivencia entre diferentes instancias de una misma actividad tiene sentido dentro del concepto de la aplicación. En estos casos, lejos de dar lugar a una aplicación poco intuitiva para el usuario, la convivencia entre instancias de una misma actividad le blinda funcionalidades nuevas.

  Por ejemplo, en un navegador de Internet que permite la navegación por pestañas, cada pestaña es una instancia de la misma actividad y la gracia de este tipo de navegación es que varias pestañas convivan al mismo tiempo. Esto permitirá que el usuario pueda ir de una pestaña a otra recuperando las páginas web que en cada una de estas estaba visitando. En este caso, las diferentes pantallas/actividades de la aplicación no siguen una estructura secuencial y, por tanto, el uso de múltiples instancias está totalmente justificado.

No es el caso de “Terrestrial Blast!”, donde los menús sí que siguen una estructura secuencial y duplicar instancias provocaría los problemas de usabilidad que hemos visto.

Concluimos por tanto que, a la hora de diseñar las diferentes pantallas de una aplicación para Android, deberemos tener en cuenta qué estructura queremos presentar al usuario, con el objetivo de que esta sea intuitiva y fácil de usar.

Una vez escogida la estructura, hay que diseñar las diferentes actividades, el orden en que aparecerán y las acciones que se llevarán a cabo en el momento en que haya una transición de una a otra. Todo esto teniendo en mente el presentar al usuario la estructura escogida, así como la existencia de la tecla Atrás, la cual siempre deberá dar unos resultados coherentes.

Por tanto, diseñar toda la estructura de actividades para una aplicación medianamente compleja no es una tarea tan trivial como puede parecer y requiere de una buena atención por parte del desarrollador.

4. Arquitectura interna de “Terrestrial Blast!”

Durante la especificación de una aplicación decidimos qué funcionalidades va a tener dicha aplicación; eso es lo que a grandes rasgos podemos extraer de los casos de uso introducidos con anterioridad. A grandes rasgos debido a que hay parte de la especificación que no se ha mostrado, pues lo que en ella se expresa también se expresa a través de los esquemas de diseño que vamos a ver a continuación.

Ahora comenzamos con el diseño, que no es más que el proceso de decidir cómo se van a implementar las funcionalidades que durante la especificación se han indicado.

4.1. Requisitos no funcionales

Lo que definirá la estructura concreta utilizada para implementar el software, así como la forma en que este se implementará, son los llamados **requisitos no funcionales**. Estos requisitos son propiedades genéricas, es decir, no están relacionados directamente con las funcionalidades de nuestra aplicación. Eficiencia, extensibilidad o usabilidad, son ejemplos de requisitos funcionales.

Todos los requisitos funcionales tienen su aportación positiva a la aplicación, por tanto, podría parecer que lo mejor es hacer una aplicación que cumpla con todas estas propiedades, pero resulta que muchas de ellas son contradictorias. Por ejemplo, podemos hacer una aplicación altamente eficiente, pero para ello se tendrá que hacer una estructura menos ordenada, lo que llevará a un bajo índice que portabilidad, extensibilidad, reutilización, etc. Es por ello que tendremos que decantarnos por unas u otras propiedades antes de comenzar a realizar el diseño; o al menos definir qué grado de influencia sobre el diseño queremos que tenga cada una.

En este caso, como estamos construyendo un videojuego, es importantísimo que este vaya lo suficientemente fluido como para que el usuario pueda jugarlo. Por tanto, la **eficiencia** es uno de los requisitos más importantes que la aplicación deberá tener. Pero no nos quedamos solo con este, pues del análisis de antecedentes podemos extraer la necesidad de que nuestra aplicación cuente con otras propiedades también muy importantes:

- **Extensibilidad**: Cuando hacemos un videojuego queremos que este tenga varios niveles, enemigos, objetos, etc. Con lo cual queremos que sea fácil extender el código del videojuego para añadir nuevos elementos como los que se acaban de citar.

  Esta propiedad, además, también facilitará el añadir nuevas funcionalidades al videojuego, como pueden ser nuevos menús con funciones online que puedan llegar en futuras versiones de la aplicación.

De hecho, lo más probable es que en futuras segundas partes de este videojuego se utilice la misma base o motor de juego, con lo cual el número de añadidos podría ser realmente grande.

- **Portabilidad**: Como ya sabemos a estas alturas de memoria, el mercado actual de sistemas operativos móviles cambia continuamente. Hay muchos sistemas operativos diferentes y los porcentajes de utilización de unos u otros van cambiando cada año, de forma que no parece que vaya a haber ningún vencedor definitivo a corto plazo.

  Por tanto, es muy importante que nuestra aplicación pueda trasladarse fácilmente a otras plataformas. Es por este motivo que hemos escogido librerías multiplataforma bastante conocidas para implementar las físicas y los gráficos del videojuego. Pero aun así, no está de más que la estructura de este videojuego se preste a poder portar fácilmente algunas partes del videojuego, como son los datos de la base de datos o el apartado visual de los menús, de una plataforma a otra.

  El objetivo, por tanto, es tener que hace el mínimo número de cambios posibles a la hora de trasladar el videojuego de la plataforma original a otra y, además, que estos cambios estén localizados en un lugar concreto de la estructura de la aplicación.

- **Reutilización**: Como se ha explicado, la aplicación tendrá un gran número de entidades, ya sean niveles, enemigos, objetos, etc. Y a poco que pensemos en entidades diferentes a implementar, nos damos cuenta de que nos encontraremos con muchas entidades que tendrán parte del código en común. Probablemente, habrá un subconjunto que tendrá solo una parte en común, otro que tendrá otra parte en común, ...

  Por tanto, es muy importante que la estructura de la aplicación se preste a que las diferentes partes o componentes de cada entidad se puedan reutilizar en otras entidades, combinándose con otros componentes. Esto nos llevará a tener un número muy amplio de piezas que poder combinar a nuestro antojo sin necesidad de implementarlas en más de una ocasión, formando un gran número de posibilidades diferentes.

  Podemos concluir, entonces, que esta reusabilidad nos ayudará a acelerar el proceso de desarrollo, puesto que nos permitirá ahorrarnos el duplicar código; y, por consiguiente, a la hora de solucionar errores lo tendremos más fácil, pues el mismo error no estará duplicado en diferentes clases.

Por tanto, son cuatro los requisitos funcionales que “Terrestrial Blast!” debe cumplir a través de su diseño. Pero, ocurre que la eficiencia se contradice normalmente con las propiedades de extensibilidad, portabilidad y reutilización. Esto es debido a que, para conseguir estas tres últimas propiedades, debemos tener una aplicación muy ordenada, dividida en diferentes capas independientes; lo cual implica que peticiones del usuario pasen por más clases, así como que las diferentes clases se comuniquen de forma más limitada entre ellas y esto acaba provocando una pérdida de eficiencia.

Llegados a este punto, es necesario decidir qué peso va a tener la eficiencia y qué peso van a tener las otras tres propiedades.

En el caso de “Terrestrial Blast!”, la eficiencia es la propiedad más importante, pues se trata de una aplicación en tiempo real y, por tanto, esta debe dar una respuesta lo suficientemente rápida como para que los niveles reproducidos fluyan con total normalidad para el usuario. Ahora bien, una vez alcanzada la eficiencia suficiente como para que el videojuego dé una respuesta aceptable en teléfonos móviles de potencia media, se da total prioridad al resto de propiedades.

Aún así, nunca llegamos a dar manga ancha total a extensibilidad, portabilidad y reutilización, ya que pueden darse algunos casos en los que haciendo un diseño que potencia altamente estas tres propiedades no se consiga prácticamente mejora en ellas. En estos casos se opta por adoptar la solución desestructurada y, por tanto, eficiente. Pues la implementación de una estructuración más estricta lleva más tiempo de desarrollo y si esta estructuración no nos da una mejora apreciable en los requisitos no funcionales, no vale la pena. Un ejemplo podría ser una funcionalidad cuya lógica sea muy pequeña, en cuyo caso no vale la pena separar la parte visual y lógica en dos capas diferentes.

4.2. Diseño por patrones

Una vez que hemos decidido qué requisitos no funcionales queremos potenciar en nuestra aplicación y en qué grado queremos hacerlo, es hora de diseñar la aplicación. Para ello vamos a utilizar la metodología de diseño por patrones.

Primero de todo, un patrón de diseño es una solución genérica a un problema de diseño.

Por tanto, el diseño por patrones consiste en diseñar nuestra aplicación a través de la adaptación de diferentes patrones de diseño al caso concreto que estamos construyendo. Si nos aseguramos de que estamos utilizando patrones de diseño que ofrecen soluciones, probadas y reconocidas, a los problemas que nos ocupan, el éxito del diseño radica únicamente en que la adaptación de estos patrones se lleve a cabo correctamente.

Como la adaptación en cuestión es muy simple, esta metodología es una buena forma de reutilizar soluciones que otros ya han ideado, evitando así reinventar la rueda, lo que nos llevaría un tiempo de desarrollo mucho mayor.

Eso sí, como se ha resaltado anteriormente, debemos tener muy en cuenta que estas soluciones deben estar probadas y, para nuestra seguridad, reconocidas por los diferentes profesionales del sector. Y es que si partimos de un patrón que no ofrece una solución correcta a un problema, la adaptación a nuestro diseño acabará siendo igualmente incorrecta.

A la hora de diseñar la aplicación, es decir, construir la estructura de esta y decidir como interaccionan los diferentes subsistemas que la componen, existen dos tipos de patrones de diseño a aplicar. Cada uno de estos tipos persigue un objetivo diferente:

- **Patrones Arquitectónicos**: Este tipo de patrones indican cómo se debe estructurar la aplicación a grandes rasgos. Definen una serie de subsistemas, el rol que hace cada uno de estos y la interacción que debe haber entre ellos. Todo con el objetivo de hacer una estructura general que cumpla con los requisitos no funcionales escogidos para la aplicación.

- **Patrones de diseño**: Este tipo de patrones definen como solucionar problemas de diseño más concretos. Se encargan pues de refinar subcomponentes y las relaciones entre ellos. Nuevamente con el objetivo de cumplir con los requisitos no funcionales escogidos.

Por tanto, en este segundo videojuego, con los patrones arquitectónicos se ha definido la estructura general de la aplicación: los subsistemas y componentes que la

componente, así como la forma en que dialogan los subsistemas más grandes. Seguidamente, a través de los patrones de diseño, se han perfilado los componentes más pequeños, dividiéndolos en varios subcomponentes si se ha considerado necesario y especificando como dialogan entre ellos.

4.2.1. Patrones Arquitectónicos aplicados en "Terrestrial Blast!"

En primer lugar, sobre este videojuego se ha aplicado el patrón Arquitectura en dos capas, el cual divide la aplicación en dos subcomponentes bien generales (ver Figura 4.7).

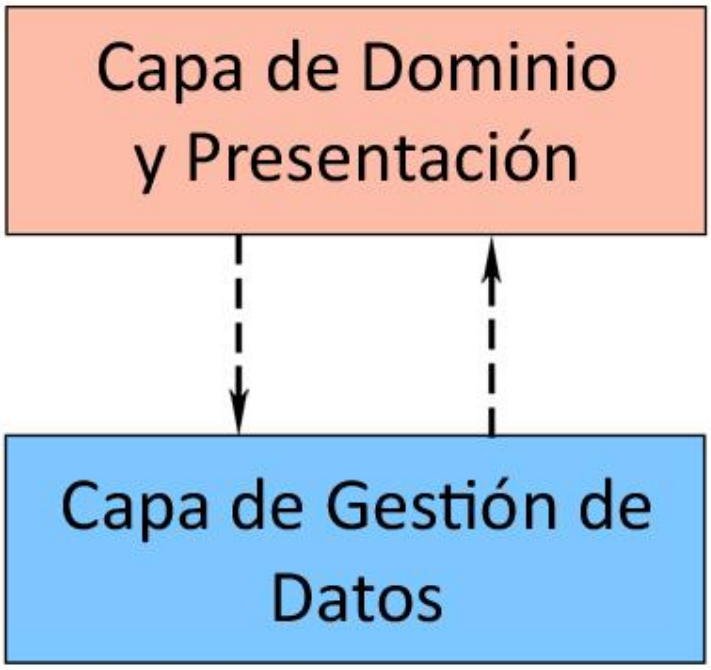


Figura 4.7: Estructura general de la aplicación "Terrestrial Blast!".

El primero de los subcomponentes es la llamada Capa de Gestión de Datos. En esta capa se encuentran todas las clases de Java que gestionan los recursos y datos persistentes de la aplicación. Ya sean los recursos de solo lectura o los datos que la aplicación crea, modifica y consulta a lo largo de su ejecución.

El segundo, es la llamada Capa de Dominio y Presentación, que como su nombre indica, agrupa dos tipos de clases Java:

- En primer lugar, las que se encargan de la interacción con el usuario, reciben sus peticiones y presentan visualmente lo que este ha solicitado.

- Y, en segundo lugar, las clases java que se encargan del dominio de la aplicación: realizar cálculos, simular un mundo físico, etc. En resumen, gestionan los datos que se leen de la entrada/salida de la aplicación o que introduce el usuario tratándolos como sea necesario para generar otros datos que se presentarán al usuario como resultado de sus peticiones.

Esta capa depende de la capa de Gestión de Datos, pues accede a sus funcionalidades (esto es lo que viene a indicar la flecha discontinua de la Figura 4.7). También ocurre lo mismo a la inversa, pues es inevitable que la capa inferior tenga cierta información sobre parte de algunos de los componentes de la capa superior. Esto es debido a que la capa de Gestión de Datos es la que leerá los ficheros donde se definen los diferentes niveles del videojuego, definiciones que serán interpretadas por la capa superior. Y, por tanto, conocerá los atributos a partir de los cuales se crearán instancias en esta capa superior.

Eso sí, para potencia la extensibilidad y la portabilidad de la aplicación, se ha asegurado que el acceso a la capa de Gestión de Datos se lleve a cabo por clases de la capa superior que son lo suficientemente generales como para no generar acoplamientos cíclicos, consiguiendo así que el acoplamiento entre ambas capas sea lo más leve posible. Se explica con más detalle el acoplamiento de capa de Gestión de Datos a capa de Dominio y Presentación en el apartado 5 de este mismo capítulo.

Por otro lado, no se ha hecho distinción entre capa de presentación y capa de domino porque Android realiza otra división de la aplicación, y se ha decidió tomar esta subdivisión, pues, como veremos a continuación, a través de ella se consiguen satisfacer, en el grado marcado, los requisitos no funcionales de la aplicación.

Y es que tal y como se explica en el apartado 2 del capítulo 2, Android divide las aplicaciones en componentes de diferentes tipos. El que más nos interesa de estos tipos de componente son las Actividades, pues es el único que se ha visto implementado en esta aplicación. Como ya se explicó en el apartado antes citado, cada Actividad gestiona una pantalla de la aplicación; ya sea una pantalla de menú o la pantalla donde se ejecuta cada uno de los niveles.

Como en un videojuego surgen actividades que tienen un dominio muy simple y otras que tienen un dominio muy complejo, se ha decidió dividir la capa de Dominio y Presentación en diferentes subcomponentes (ver Figura 4.8). Cada uno de estos subcomponentes engloba las clases que se encargan de la gestión de una actividad concreta, entre las que figurará, por supuesto, la clase Java que implementa la actividad en cuestión. De forma que para cada uno de estos subcomponentes se decidirá individualmente si se efectúa la división entre Capa de Dominio y capa de Presentación o, por el contrario, se mantiene la unión entre ambas capas.

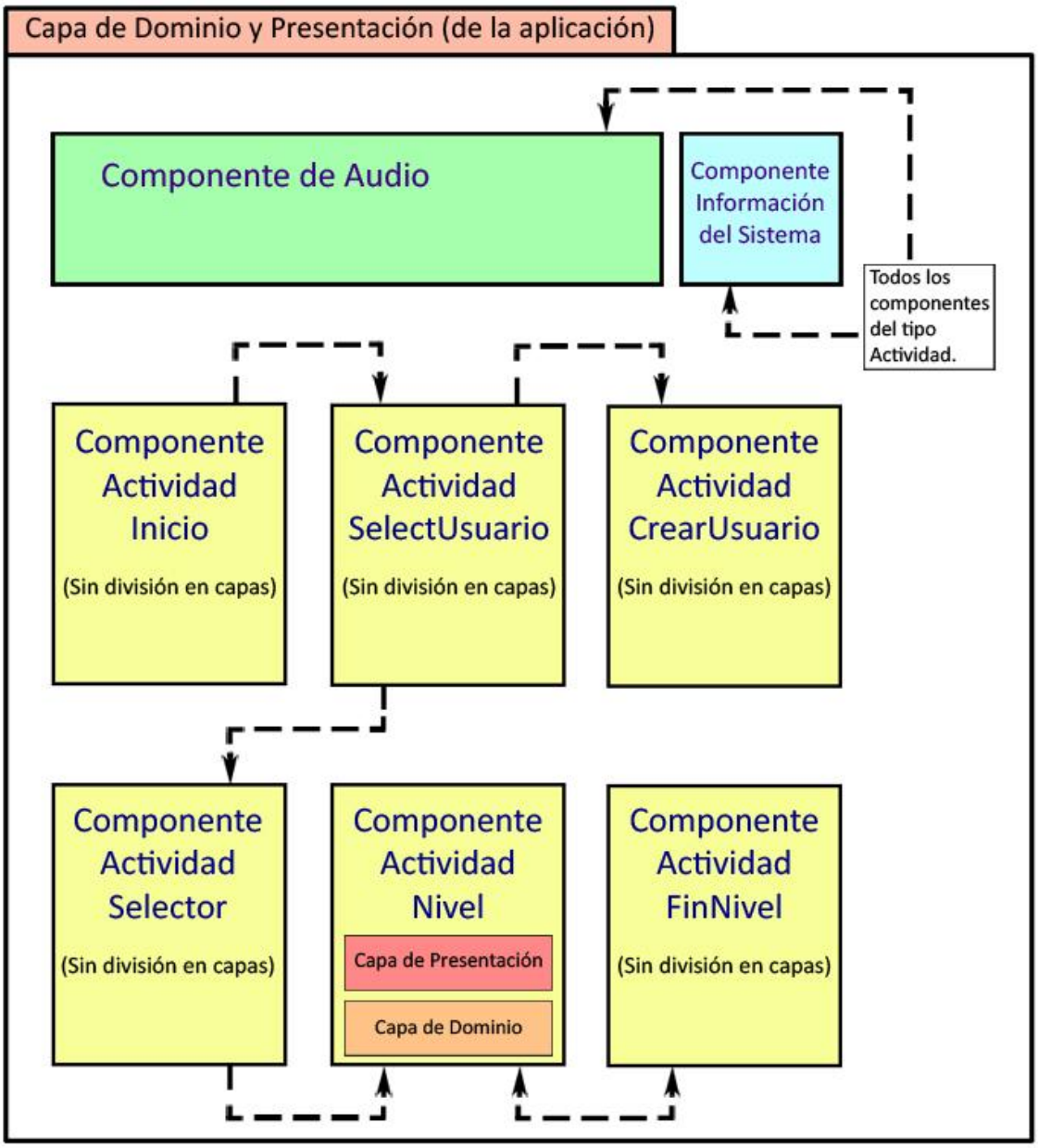


Figura 4.8: Diseño de la estructura interna de la capa de Dominio y Presentación de "Terrestrial Blast!".

A pesar de que esta división puede parecer algo liosa, realmente es muy clara, porque la única actividad que va a subdividirse en dos capas es la *ActividadNivel*, pues es la que se encarga de la ejecución de los diferentes niveles y, por tanto, la que tiene una estructura más compleja.

El resto de actividades tienen una lógica insignificante y esta se mezcla con la presentación por tal de cumplir con el requisito marcado en el apartado anterior, el cual nos decía que no había que potenciar los requisitos no funcionales de extensibilidad y portabilidad, si esta potenciación era ínfima a cambio de una perdida de eficiencia, que además generaba un código lioso a la hora de comprenderlo y costoso en tiempo de desarrollo.

La división de la capa de Dominio y Presentación en subcomponentes no acaba aquí, pues, aunque las actividades por lo general sean componentes independientes, a parte de la ya citada capa de Gestión de Datos también comparten dos componentes más.

En primer lugar, se comparte el Componente de Audio, el cual esconde todas las clases que gestionan la reproducción de las diferentes músicas y sonido de la aplicación.

Hay numerosos motivos por lo cual es recomendable que todo el audio de la aplicación se gestione de forma global a través de un componente, a continuación se presentan los más destacables:

- El primer motivo y el que más importancia tiene, es que de cara al usuario, queda incomodo que cada actividad reproduzca su propio audio. Esto llevaría a que entre actividades diferentes no se pudiese mantener la reproducción de la misma pista musical, pues esta tendría que pausarse cuando cambiásemos de actividad y la nueva actividad tendría que reproducir su propia música. Lo mismo ocurriría con los sonidos, que al dejar una actividad se cortarían aunque no hubiesen acabado su reproducción. Además, este no es un problema que se pueda llegar a dar en pocas actividades, pues como podemos ver en la Figura 4.8, todos los componentes de actividad depende del Componente de Audio (indicado a través de la flecha discontinua) ya que todos reproducen audio.

- Otro motivo importante, es que el mecanismo que gestiona la reproducción del audio se puede implementar de forma mucho más eficiente si todo este es gestionado por las mismas instancias. Por ejemplo, podemos ahorrarnos el cargar un fichero de sonido dos veces en memoria principal cuando hay dos objetos que reproducen el mismo sonido, pues tratándose de la misma instancia de componente musical utilizada por ambos basta con reproducir el mismo fichero dos veces.

- Por último, también hay que tener en cuenta la reutilización, que se potencia centralizando toda la gestión del audio en un mismo componente. De esta forma no tenemos que implementar una gestión para el audio en cada uno de los componentes de actividad, lo que supondría repetir código.

El segundo de los componentes externos a los componentes de actividad que tenemos es el Componente de Información del Sistema. Este componente esconde información acerca del estado actual de la aplicación, como el usuario con el que se ha iniciado sesión en un momento dado, entre otros datos de interés para las actividades.

De esta forma los componentes de actividad pueden acceder a dicha información a través de un componente con instancia única, que se encuentra en todo momento en la memoria principal. En caso contrario, estos tendrían que acceder constantemente a la memoria interna del dispositivo para consultar la misma información una y otra vez, lo que empeoraría la eficiencia de la aplicación, pues este tipo de acceso es costoso. O, por el contrarío, debería pasarse esta información entre actividades, lo que agravaría la dependencia entre componentes de actividad, así como empeoraría la claridad del código.

Por último, decir que de este segundo componente general también dependen todas las actividades, pues todas tienen que consultar o rellenar de información al componente de Información del Sistema.

En cuanto a los componentes de actividad, también a través de la Figura 4.8 podemos ver qué dependencia existe entre ellos. Para poder entender mejor estas dependencias, es aconsejable leer el apartado 3 de este mismo capítulo, donde se presentan las diferentes actividades, describiendo brevemente la función de cada una, indicando qué interacción llevan a cabo entre ellas, así como el desencadenante de cada una de estas interacciones.

Como conclusión, acerca del patrón arquitectónico Arquitectura en Capas y de la división en componentes de Android, junto con la división en componentes de aportación propia, podemos decir que la división de la aplicación en diferentes capas y componentes proporciona los siguientes beneficios:

- Facilita la claridad del código, pues sabemos en qué componente se encuentra la implementación de cada una de las diferentes partes de la aplicación. Esto significa que, en consecuencia, a la hora de ampliar las funcionalidades de la aplicación sabremos exactamente donde tenemos que provocar cambios y añadir código. Es decir, conseguimos aumentar la extensibilidad.

Pero aun puede darse algún problema, pues al haber dependencias entre componentes, el modificar uno de estos puede provocar el tener que modificar otro que depende del primero, este problema se resolverá aplicando el patrón de diseño Controlador, como veremos a continuación; patrón que sin la previa división en componentes no se podría aplicar.

- Por otro lado, la división en capas y componentes también facilita la portabilidad de la aplicación. Y es que al estar todas las funcionalidades separadas en diferentes componentes, podemos cambiar la forma que tenemos de implementar una parte de la aplicación simplemente con cambiar el componente que contiene dicha parte y sin tener que preocuparnos del resto de la aplicación.

  Por ejemplo, si queremos cambiar el sistema gestor de la base de datos podemos hacerlo simplemente tocando la capa de Gestión de Datos y sin tocar las demás. Lo mismo ocurriría si quisiéramos cambiar la forma en que se gestiona el audio, en cuyo caso solo tendríamos que tocar el Componente de Audio; caso que puede darse si decidimos portar la aplicación a otro sistema operativo donde el audio se gestione de forma diferente.

  Aunque, hay que decir que esta portabilidad se ve completada cuando aplicamos el patrón de diseño Controlador, igual que pasaba en el caso anterior. Pues sino la dependencia entre paquetes podría provocar que tuviésemos que modificar también otros paquetes.

- Por último, es importante decir que, por lo general, la eficiencia se ve resentida al dividir la aplicación en capas y componentes. Esto es debido a que con estas divisiones una petición del usuario debe pasar por más clases Java y, por tanto, la respuesta puede ser más lenta. Por este motivo, como se ha explicado a lo largo del apartado, se ha evitado el dividir en capas en casos en los que esta división no ofrecía ninguna mejora apreciable en los requisitos no funcionales de la aplicación o ponía en entredicho el cumplir con la eficiencia mínima que se ha marcado como objetivo en el desarrollo de la aplicación.

  Pero aún así, la división en componentes también puede mejorar la eficiencia de la aplicación, si esta división provoca la gestión conjunta de una serie de acciones que tratadas conjuntamente pueden implementarse de forma más eficiente, como hemos visto que ocurre con el Componente de Audio.

Una vez aplicado el patrón arquitectónico Arquitectura en Capas, así como la división en componentes, obtenemos unas capas y unos componentes que aun no son lo suficientemente específicos. Por tanto, no nos basta con esta división, para poder implementar la aplicación debemos trabajar sobre entidades más pequeñas y concretas. Esto nos lleva a aplicar el patrón arquitectónico Orientación a Objetos.

El patrón Orientación a objeto se aplica sobre cada uno de los componentes y capas, subdividiéndolos en clases (en este caso del lenguaje de programación Java). No es más que la división que ya dábamos por hecha a lo largo del apartado, cuando hablábamos del interior de los componentes y capas generados, pues sabemos que cuando desarrollamos aplicaciones con un lenguaje orientado a objetos la unidad más pequeña de subsistema con la que acabamos trabajando es el objeto.

Un objeto es una instancia, concreta e independiente, de una clase de objetos; y una clase de objetos no es más que un conjunto de atributos y operaciones relacionados entre sí. Dentro de cada capa o componente, los diferentes objetos responden a estímulos externos (provocados por otros objetos que podrían proceder de otros componentes o capas) o internos, lo que provoca que: cambien su estado, intercambien información o produzcan resultados.

Por último, por tal de potenciar la programación orientada a objetos, cada clase debe tener unas responsabilidades parecidas o muy relacionadas entre sí. De esta forma se evita que una clase tenga demasiadas responsabilidades, lo cual sería equivalente a una división menor de los componentes y, por tanto, un uso menos beneficioso de la orientación a objetos que tantas ventajas tiene.

4.2.2. Patrones de diseño aplicados en “Terrestrial Blast!”

Una vez detallados los patrones arquitectónicos que se han aplicado sobre “Terrestrial Blast!” los cuales han llevado a que, en última instancia, estemos trabajando durante el resto del desarrollo a nivel de clases de objetos, es hora de ver qué patrones de diseño se han aplicado para perfilar las clases así como la comunicación entre ellas, por tal de conseguir alcanzar los requisitos no funcionales propuestos.

Para ello, en los siguientes apartados se presentará el fragmento de diagrama de clases de cada componente surgido de la partición anterior. A través de este diagrama podremos ver las clases implementadas junto a los atributos y operaciones **más importantes** de cada una de estas. Con estos fragmentos de diagrama veremos también cómo se asocian estas clases y qué dependencias se crean entre unas y otras. Y, por último, se detallarán los patrones de diseño aplicados, así como qué requisitos funcionales potencian y en qué grado.

Antes de pasar a ver los fragmentos de diagrama se van a describir dos patrones que se han tenido en cuenta durante toda la partición de todos los componentes. Se trata de los patrones Creador y Experto.

En primer lugar, el patrón Creador [19] nos ayuda a decidir en qué clase se debe crear una instancia de otra clase. Todo ello enfocado a conseguir un bajo acoplamiento, es decir, que cada clase sepa la existencia de contra menos clases mejor y en caso de conocer la existencia de una clase, tenga la mínima información posible sobre ella. Para conseguir este propósito nos aconseja que la instancia se cree desde clases que ya conozcan este tipo de instancias o que, en todo caso, manejen los atributos necesarios para la creación de la instancia, entre otras consideraciones.

En según lugar, tenemos el patrón experto, este patrón nos ayuda a distribuir las responsabilidades en clases. Como ya hemos dicho para hacer un buen uso de la orientación a objetos, debemos crear unas clases que individualmente se ocupen de unas responsabilidades muy concretas. De esta forma conseguimos tener una estructura con variedad de clases y no pocas clases muy extensas, lo que dificultaría el desarrollo y posterior mantenimiento de estas. El patrón ofrece medidas a tomar como son asignar una responsabilidad a la clase que tiene la información para realizarla, entre otras.

Como vemos, estos dos patrones son imprescindibles para conseguir un bajo acoplamiento y una alta cohesión en cada clase, lo que acaba potenciando la extensibilidad, la reutilización y la portabilidad. Pero también hay que tener en cuenta que nuestro videojuego tiene que cumplir con un mínimo de eficiencia y por tanto, que debemos aplicar estos dos patrones sin llegar a extremos que sean nefastos para esta eficiencia (pues los dispositivos móviles de gama media no tienen tan buenas prestaciones como para poder desarrollar libremente el videojuego sin tener en cuenta este factor).

De forma adicional a estos patrones se ha aplicado una norma básica en el diseño del diagrama de clases, se trata de la norma que dice que los atributos de las clases siempre tienen visibilidad privada, lo que significa que solo se pueden ver desde dentro de la clase que los define.

De esta forma, clases ajenas no ven las estructuras de datos de que se componen el resto de clases, sino que acceden a estas únicamente a través de las operaciones públicas.

Esta norma es básica para conseguir que la implementación interna de cada clase se pueda modificar a placer sin tener que recurrir a la modificación de otras clases. Pues si tengo que cambiar una estructura de datos, solo tendré que amoldar la operación de la propia clase que permitía el acceso a esta estructura por tal de que siga devolviendo

los resultados que se especifican en su definición. Por tanto, la portabilidad y extensibilidad se ven mejoradas gracias a esta norma.

Además, la eficiencia no tiene porque sufrir un descenso apreciable por culpa de la aplicación de esta norma si definimos unas operaciones precisas, que nos permiten la comunicación entre clases sin demasiadas complicaciones.

Diseño interno de la capa de Gestión de Datos

Empecemos por el diagrama de clases de la capa de Gestión de datos, el cual podemos ver en la Figura 4.9.

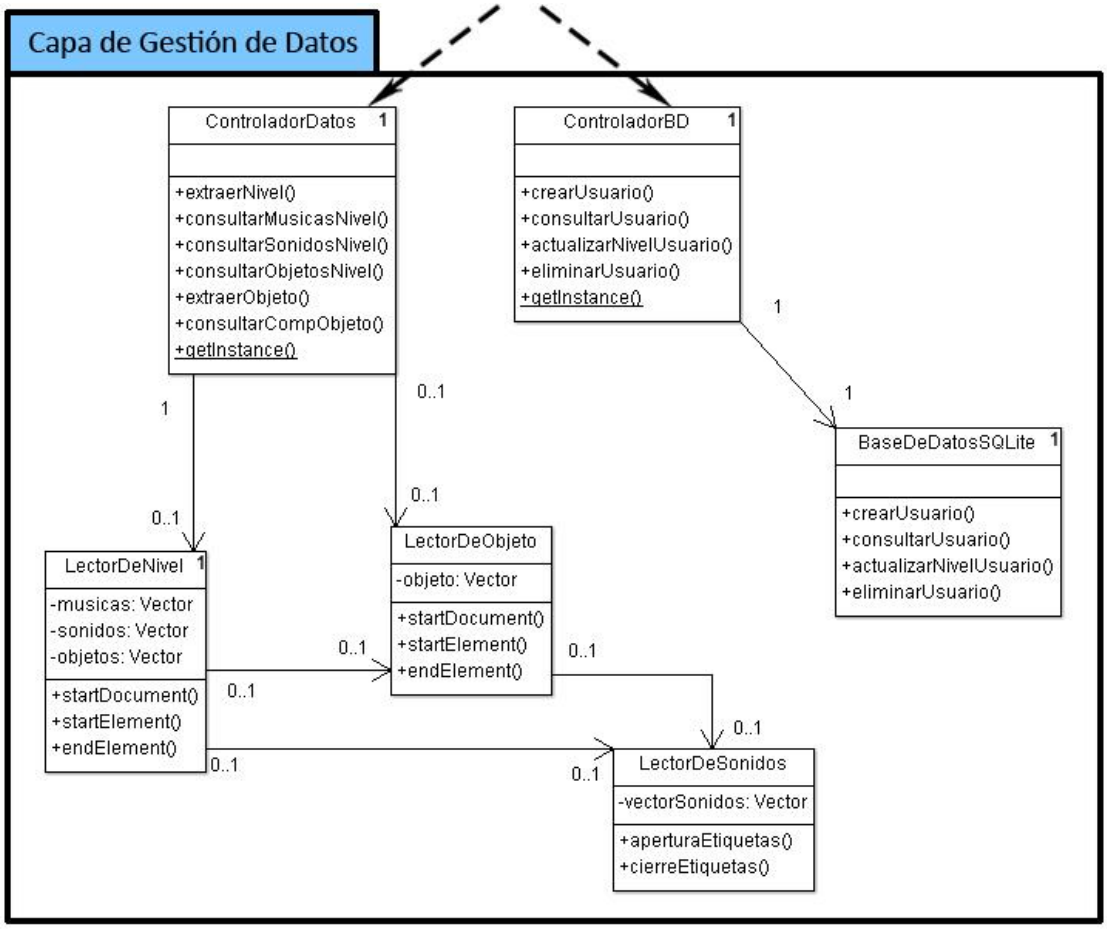


Figura 4.9: Diseño de la estructura interna de la capa de Gestión de Datos de "Terrestrial Blast!".

En este diagrama debemos destacar la aplicación de un patrón que se va a aplicar sobre todo componente o capa definido en la aplicación. Se trata del patrón **Controlador**.

Este patrón de diseño nos permite que el componente se vea externamente como una caja negra. Es decir, el resto de componentes, no saben las clases que componen a este, tan solo conocen la existencia de las clases llamadas *controlador*. Estas clases

controlador, se crean específicamente para ocultar el resto, así que suelen tener únicamente las operaciones necesarias para contestar a todas las posibles solicitudes externas.

Gracias a la aplicación de este patrón, como ya se explicó en el apartado anterior, conseguimos que un componente se pueda modificar o extender sin que el resto de componentes noten ningún cambio, pues se modifican unas clases que para el resto no existe. Ojo esta modificación será viable siempre y cuando la capa o el componente siga pudiendo responder a las solicitudes que otros les piden a través de los controladores.

Por tanto, el patrón Controlador unido al patrón Arquitectura en Capas, que es el que define el área de la caja negra, nos permiten potenciar drásticamente la portabilidad y la extensibilidad. La eficiencia se puede ver afectada, por tener un acceso menos directo a las clases internas, pues el controlador es un paso intermedio. Pero realmente es una perdida pequeña. Aun así en componentes donde hay una o pocas clases muy importantes que dirigen todo el componente, se ha optado por hacer que estas pasen a ser el controlador, evitando así definir nuevas clases controlador que, en la mayoría de los casos lo único que harían es acceder a las tachadas como importantes.

No es este el caso, de los controladores de la capa de Gestión de Datos, los cuales han sido creados específicamente para la ocasión, pues no existía una clase que pudiese hacer de controlador.

Cabe decir, que existen varios tipos de controladores posibles. Algunos son de instancia única, es decir, todo componente externo se comunica con una misma instancia de una misma clase controlador durante la ejecución de la aplicación; este es el llamado controlador Fachada. Otros tipos de controladores, en cambio, definen una clase por cada caso de uso o por cada operación del sistema [19].

En este caso se ha utilizado el controlador fachada, pues así potenciamos la reutilización de las operaciones, así como conseguimos mayor claridad. Pues muchos casos de uso acceden a las mismas operaciones de la capa de Gestión de Datos. Eso sí, como se ha explicado, se parte el controlador fachada en dos clases, pues ambas tratan diferentes tipos de accesos a los datos y esta partición mejora la claridad de la estructura.

En primer lugar, la clase *ControladorDatos*, implementa al controlador que se encarga del acceso a los recursos de solo lectura. Dentro de estos recursos entran: texturas de los objetos que componen un escenario, la definición de los diferentes objetos que componen cada nivel, etc.

En segundo lugar, la clase *ControladorBD*, implementa al controlador que se encarga de los accesos a la Base de Datos. Esta Base de Datos mantiene la información sobre los usuarios que hay registrados en el sistema. Nótese que aunque en este caso, se trata de una BD implementada mediante el sistema gestor SQLite, en cualquier momento podríamos cambiar el sistema gestor de la BD y utilizar uno que, por ejemplo, se encontrara en la red; todo esto sin necesidad de tocar más que la clase *BaseDeDatosSQLite*.

Una vez visto el patrón Controlador, nos queda por ver otro muy importante, pues durante la explicación anterior se ha dicho que el controlador fachada tiene una única instancia a la cual acceden todas las clases que quieran solicitar los servicios de este Controlador.

Como nuestra capa de Dominio y Presentación está dividida en muchos subcomponentes, los cuales, quieren acceder a la capa de Gestión de Datos, compartir la misma instancia de controlador requiere una comunicación entre todos los componentes para irla pasando y utilizando. Esto aumenta el acoplamiento y, por tanto, dificulta la extensibilidad y portabilidad de la aplicación.

Para solucionar este problema tenemos el patrón **Singleton** [19]. Este patrón permite que accedamos a la instancia de cualquier clase con instancia única, desde cualquier sitio y sin necesidad de acoplarse con otros componentes simplemente para compartirla.

Para conseguirlo, en primer lugar, creamos una operación de clase, sobre la clase a la cual se le va a aplicar el patrón, llamada *getInstance()*. Esta operación nos devolverá la única instancia de dicha clase que existe. Por tanto, debe existir también un atributo de clase privado que contenga esta instancia única.

Como las operaciones y atributos de clase no corresponden a ninguna instancia de la clase, sino a la clase en sí, podemos acceder a ellos simplemente indicando el nombre de esta clase desde cualquier lugar, siempre y cuando tengamos visibilidad. En este caso solo la operación tiene visibilidad pública y, por tanto, podrá ser accedida.

En resumen, la clase que quiera acceder al controlador llamara a la operación *getInstance()* y recibirá la instancia deseada, desde la cual podrá llamar a las operaciones que desee.

Hay que tener en cuenta que en algunos cosas, se ha creado una operación para inicializar la instancia única, como por ejemplo *inicializarIntancia()*, mientras que en otros, la instancia se crea automáticamente o la propia operación *getInstance()* se encarga de crearla. Según las necesidades de la clase, por tanto, se tendrá que llamar *inicializarInstancia()* antes de hacer uso de *getInstance()* o no.

Por último acerca del patrón Singleton, es importante advertir que no solo se utiliza en clases controlador, sino que puede aplicarse sobre toda clase que solo tenga una única instancia y queramos poder acceder a esta desde cualquier parte. De hecho en algunos componentes de esta aplicación veremos clases que no son controladores y lo aplican.

Una vez explicados los patrones que se han aplicado sobre esta capa, así como las funciones que se realizan en ella, solo queda decir, que la aplicación se ha desarrollado pensando en la extensibilidad del videojuego, como ya se ha comentado antes y esto ha llevado a permitir dos niveles de desarrollo.

En el nivel más bajo, nos encontramos desarrollando en java, y partiremos del diagrama de clases que se está explicando a lo largo de estos apartados. Y en el nivel más alto, se desarrolla en el lenguaje XML. Es a través de este lenguaje que se definen los niveles, los objetos que componen estos niveles y los sonidos y músicas del videojuego.

Esto nos lleva a tener que implementar un mecanismo que lea los ficheros XML y extraiga esta información. Todo este mecanismo se encuentra implementado a través de las clases *LectorDeNivel*, *LectorDeObjeto* y *LectorDeSonidos* que podemos encontrar en esta misma capa de Gestión de Datos (ver Figura 4.9). Estos dos niveles de desarrollo, así como los solapamientos que implican, se detallan en el apartado 5 de este mismo capítulo.

Diseño interno de la capa de Dominio y Presentación

Vamos a centrarnos ahora en ver el diseño interno de los diferentes componentes que conforman la capa de Dominio y Presentación y que podemos ver en la Figura 4.8.

a. Diseño interno del Componente de Audio

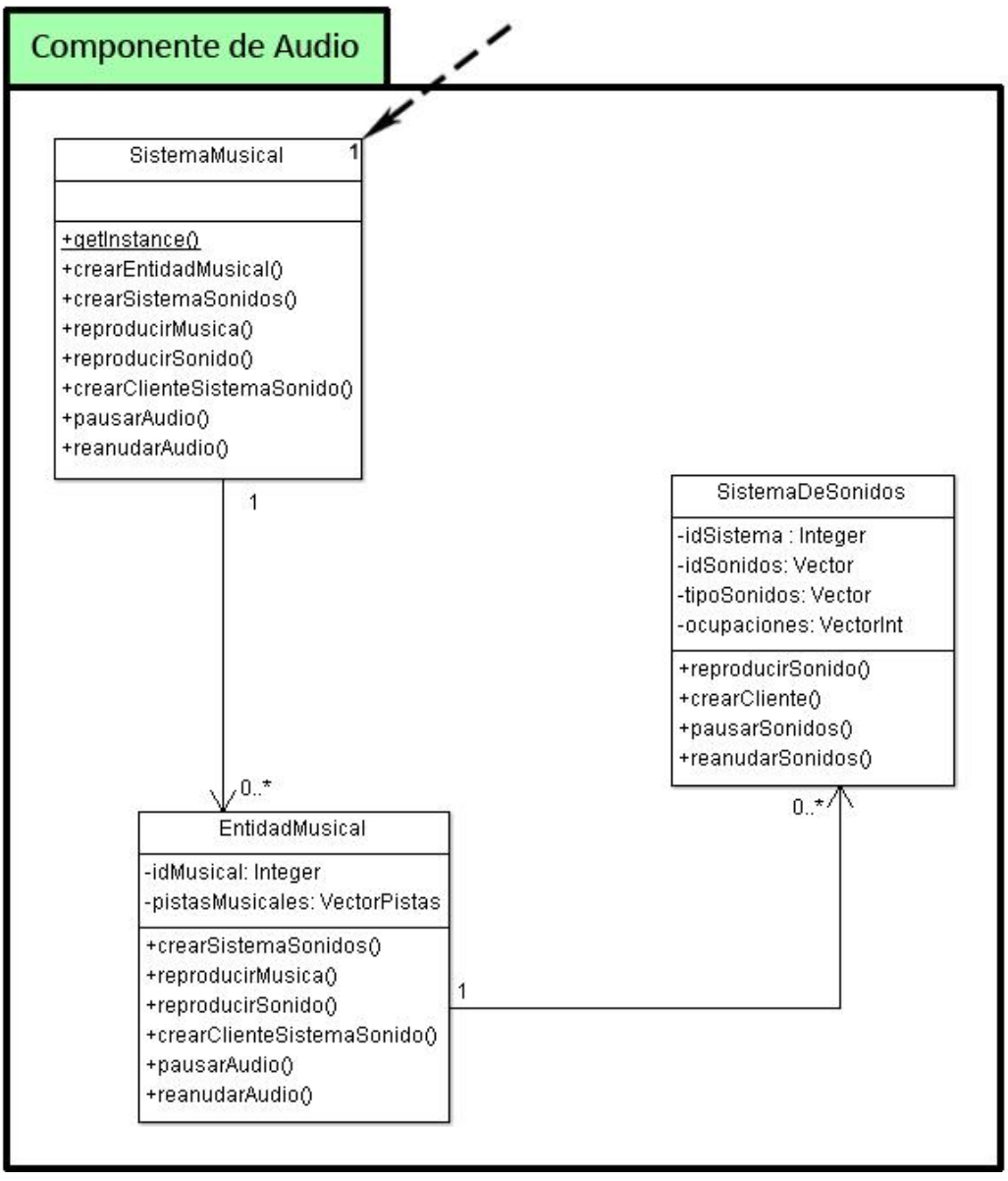


Figura 4.10: Diseño de la estructura interna del componente de audio, el cual se encuentra dentro de la capa de Dominio y Presentación de "Terrestrial Blast!".

Como recordarás este es el componente que se encarga de todo el audio de la aplicación.

Podemos ver en la Figura 4.10 como a la hora de diseñar este componente se ha hecho uso, nuevamente del patrón Controlador, dando lugar a la clase *SistemaMusical*. A esta clase también se le aplica el patrón Singleton, por tal de poder acceder a la única instancia existente desde cualquier punto, sin necesidad de ir pasando esta instancia entre las clases que la utilizan, que son muchas.

En cuanto al funcionamiento del audio, tenemos una clase *EntidadMusical*, que presenta a un conjunto de músicas y a un conjunto de Sistemas de Sonido. En la aplicación se utiliza una entidad musical para músicas y sonidos de los menús, y otra para músicas y sonidos del nivel del videojuego que se pueda estar ejecutando en un momento dado.

Por último, un Sistema de Sonidos, no es más que un conjunto de sonidos muy relacionados que se gestionan conjuntamente. Se utiliza un sistema de sonidos diferente, por ejemplo, para cada tipo de objeto (cañón, murciélago, etc.) que constituye un nivel del videojuego.

b. Diseño interno del Componente Información del sistema

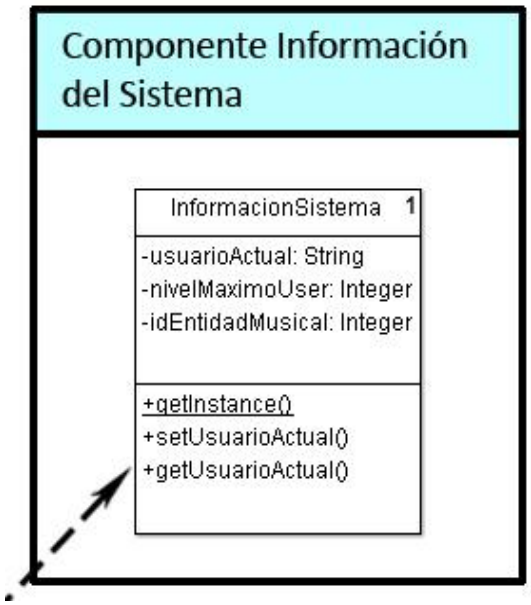


Figura 4.11: Diseño de la estructura interna del componente Información del Sistema, el cual

El componente de Información de Sistema tan solo tiene una clase (ver Figura 4.11), a la que los componentes de actividad acceden para consultar la información acerca del estado actual de la aplicación.

Nuevamente se hace uso del patrón *Singleton* para poder acceder a la única instancia de la clase desde cualquier punto. Además, la clase *InformacionSistema* también hace el rol de controlador, ya que si se tuviesen que añadir más clases al componente, esta clase ocultaría al resto.

c. Diseño interno de los Componentes de Actividad

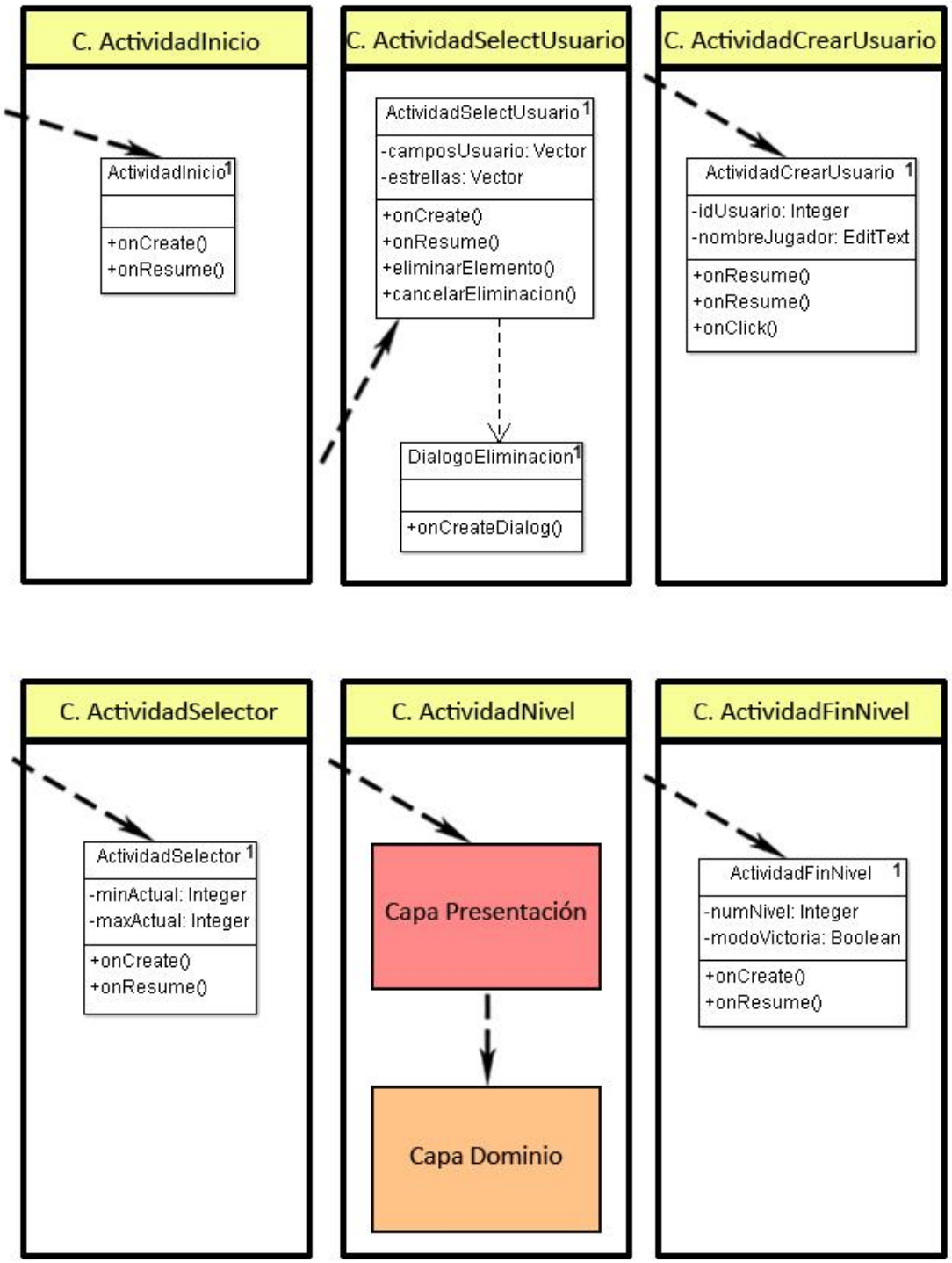


Figura 4.12: Diseño de las estructuras internas de los componentes de actividad, los cuales se encuentra dentro de la capa de Dominio y Presentación de “Terrestrial Blast!”.

En la Figura 4.12 podemos ver de una tacada, el diseño interno de los diferentes componentes de actividad.

Como vemos, la mayoría se componen de una única clase que, si se diera el caso de tener que añadir más clases al componente, haría de controlador.

En este caso no es necesario aplicar el patrón Singleton, pues no se accede a los componentes de actividad para obtener unos resultados, sino para lanzar la actividad en cuestión que, como ya se ha explicado en apartados anteriores, representa a una pantalla independiente de la aplicación. El propio sistema Android nos permite lanzar estas actividades sin necesidad de acceder a su instancia única, puesto que además, cada vez que lanzamos una actividad se crea una instancia nueva.

Como realmente desde fuera del componente de actividad únicamente se conoce el nombre de la actividad para poder lanzarla, podemos decir que esta clase llamada "*Actividad…*" de cada componente hace de Controlador, en caso de existir más de una clase dentro de este componente.

Por último, podemos ver como hay una actividad cuyo diseño interno está dividido en dos capas. Esto es debido a que se trata de la actividad que reproduce el nivel del videojuego. Es una actividad muy extensa y por tanto, se ha decidido dividir en dos capas, para mejorar la extensibilidad y portabilidad de la aplicación, así como la claridad del diseño.

c.1. Diseño interno del componente de actividad ActividadNivel

En esta apartado vamos a describir con detalle el diseño interno de la que ya hemos explicado que es la actividad más importante y compleja de la aplicación.

Se trata de *ActividadNivel*, la actividad que, de forma genérica ejecuta el nivel del videojuego que se le indica durante su creación.

Su diseño interno se ha dividió en dos capas, debido a su complejidad. En la Figura 4.13 tenemos el diseño interno de la capa de Presentación y en la Figura 4.14 tenemos el diseño interno de la capa de Dominio.

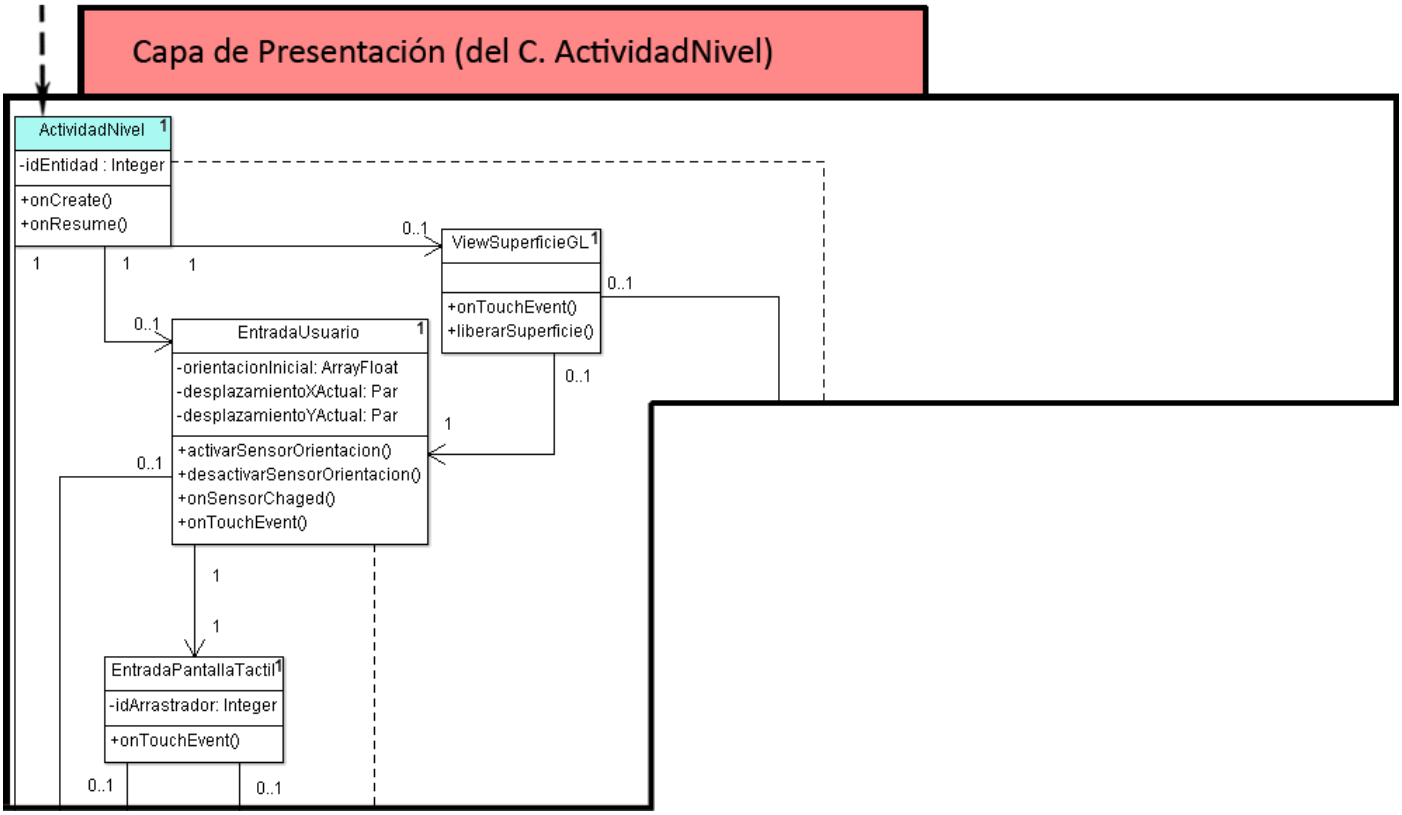
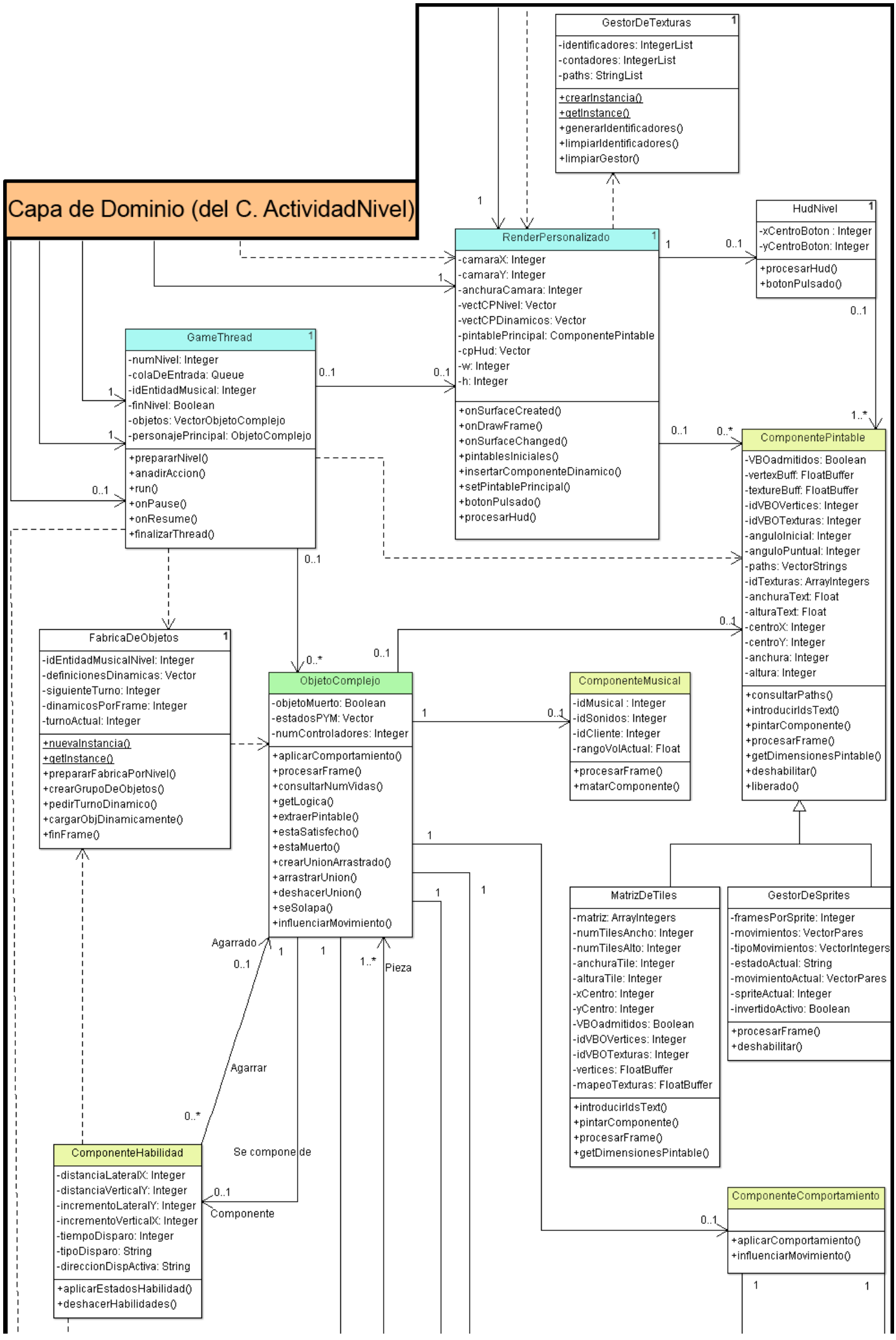
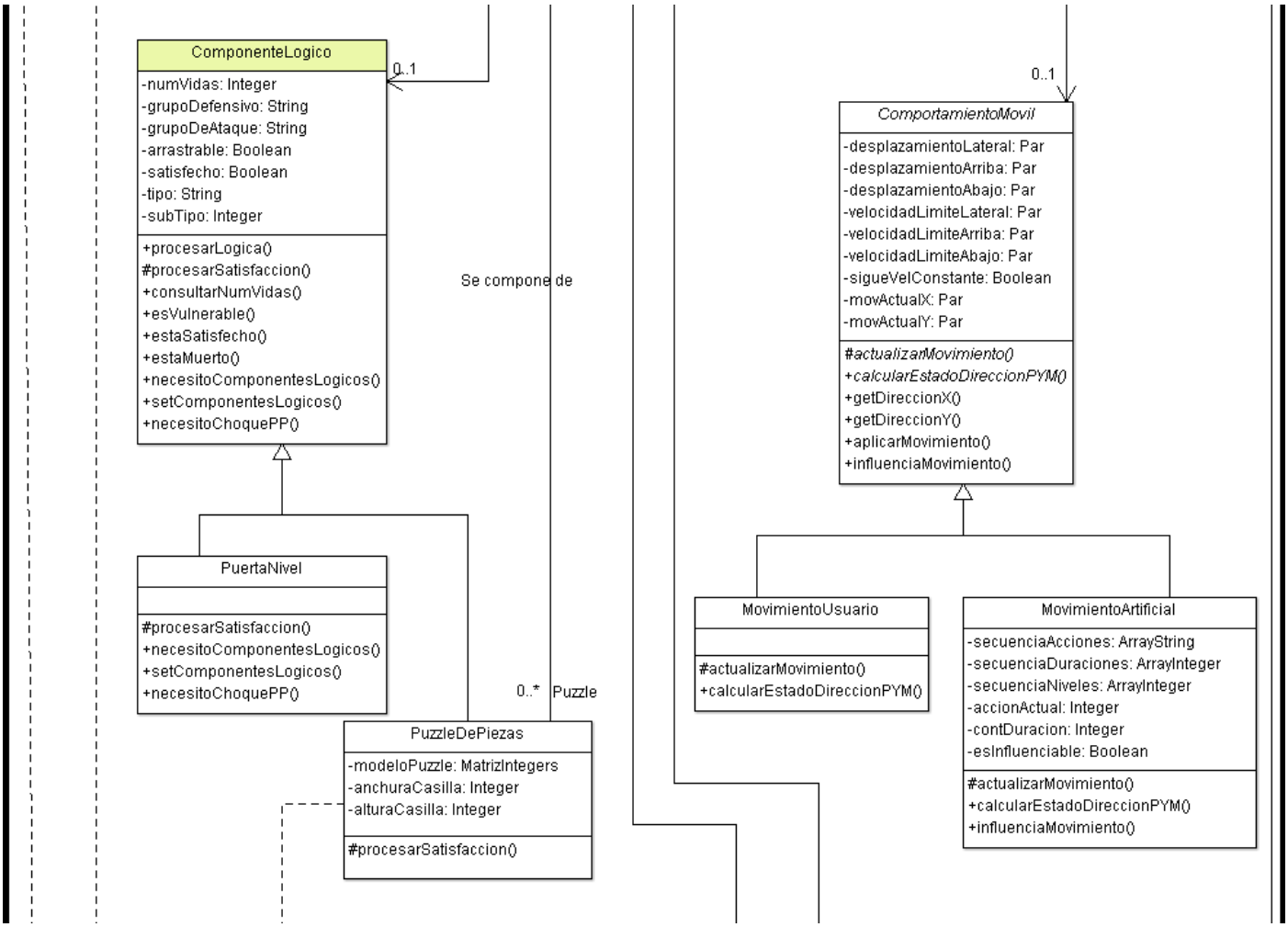


Figura 4.13: Diseño de las estructuras internas de la capa de presentación del componente de actividad *ActividadNivel*.


Capa de Dominio (del C. ActividadNivel)
GestorDeTexturas
-identificadores: IntegerList
-contadores: IntegerList
-paths: StringList
+crearInstancia()
+getInstance()
+generarIdentificadores()
+limpiarIdentificadores()
+limpiarGestor()
HudNivel
-xCentroBoton : Integer
-yCentroBoton: Integer
+procesarHud()
+botonPulsado()
RenderPersonalizado
-camaraX: Integer
-camaraY: Integer
-anchuraCamara: Integer
-vectCPNivel: Vector
-vectCPDinamicos: Vector
-pintablePrincipal: ComponentePintable
-cpHud: Vector
-w: Integer
-h: Integer
+onSurfaceCreated()
+onDrawFrame()
+onSurfaceChanged()
+pintablesIniciales()
+insertarComponenteDinamico()
+setPintablePrincipal()
+botonPulsado()
+procesarHud()
GameThread
-numNivel: Integer
-colaDeEntrada: Queue
-idEntidadMusical: Integer
-finNivel: Boolean
-objetos: VectorObjetoComplejo
-personajePrincipal: ObjetoComplejo
+prepararNivel()
+anadirAccion()
+run()
+onPause()
+onResume()
+finalizarThread()
ComponentePintable
-VBOadmitidos: Boolean
-vertexBuff: FloatBuffer
-textureBuff: FloatBuffer
-idVBOVertices: Integer
-idVBOTexturas: Integer
-anguloInicial: Integer
-anguloPuntual: Integer
-paths: VectorStrings
-idTexturas: ArrayIntegers
-anchuraText: Float
-alturaText: Float
-centroX: Integer
-centroY: Integer
-anchura: Integer
-altura: Integer
+consultarPaths()
+introducirIdsText()
+pintarComponente()
+procesarFrame()
+getDimensionesPintable()
+deshabilitar()
+liberado()
FabricaDeObjetos
-idEntidadMusicalNivel: Integer
-definicionesDinamicas: Vector
-siguienteTurno: Integer
-dinamicosPorFrame: Integer
-turnoActual: Integer
+nuevaInstancia()
+getInstance()
+prepararFabricaPorNivel()
+crearGrupoDeObjetos()
+pedirTurnoDinamico()
+cargarObjDinamicamente()
+finFrame()
ObjetoComplejo
-objetoMuerto: Boolean
-estadosPYM: Vector
-numControladores: Integer
+aplicarComportamiento()
+procesarFrame()
+consultarNumVidas()
+getLogica()
+extraerPintable()
+estaSatisfecho()
+estaMuerto()
+crearUnionArrastrado()
+arrastrarUnion()
+deshacerUnion()
+seSolapa()
+influenciarMovimiento()
ComponenteMusical
-idMusical : Integer
-idSonidos: Integer
-idCliente: Integer
-rangoVolActual: Float
+procesarFrame()
+matarComponente()
MatrizDeTiles
-matriz: ArrayIntegers
-numTilesAncho: Integer
-numTilesAlto: Integer
-anchuraTile: Integer
-alturaTile: Integer
-xCentro: Integer
-yCentro: Integer
-VBOadmitidos: Boolean
-idVBOVertices: Integer
-idVBOTexturas: Integer
-vertices: FloatBuffer
-mapeoTexturas: FloatBuffer
+introducirIdsText()
+pintarComponente()
+procesarFrame()
+getDimensionesPintable()
GestorDeSprites
-framesPorSprite: Integer
-movimientos: VectorPares
-tipoMovimientos: VectorIntegers
-estadoActual: String
-movimientoActual: VectorPares
-spriteActual: Integer
-invertidoActivo: Boolean
+procesarFrame()
+deshabilitar()
ComponenteHabilidad
-distanciaLateralX: Integer
-distanciaVerticalY: Integer
-incrementoLateralY: Integer
-incrementoVerticalX: Integer
-tiempoDisparo: Integer
-tipoDisparo: String
-direccionDispActiva: String
+aplicarEstadosHabilidad()
+deshacerHabilidades()
ComponenteComportamiento
+aplicarComportamiento()
+influenciarMovimiento()
Agarrado
Agarrar
Pieza
Se compone de
Componente


ComponenteLogico
-numVidas: Integer
-grupoDefensivo: String
-grupoDeAtaque: String
-arrastrable: Boolean
-satisfecho: Boolean
-tipo: String
-subTipo: Integer
+procesarLogica()
#procesarSatisfaccion()
+consultarNumVidas()
+esVulnerable()
+estaSatisfecho()
+estaMuerto()
+necesitoComponentesLogicos()
+setComponentesLogicos()
+necesitoChoquePP()
0..1
Se compone de
0..*
Puzzle
PuertaNivel
#procesarSatisfaccion()
+necesitoComponentesLogicos()
+setComponentesLogicos()
+necesitoChoquePP()
PuzzleDePiezas
-modeloPuzzle: MatrizIntegers
-anchuraCasilla: Integer
-alturaCasilla: Integer
#procesarSatisfaccion()
0..1
ComportamientoMovil
-desplazamientoLateral: Par
-desplazamientoArriba: Par
-desplazamientoAbajo: Par
-velocidadLimiteLateral: Par
-velocidadLimiteArriba: Par
-velocidadLimiteAbajo: Par
-sigueVelConstante: Boolean
-movActualX: Par
-movActualY: Par
#actualizarMovimiento()
+calcularEstadoDireccionPYM()
+getDireccionX()
+getDireccionY()
+aplicarMovimiento()
+influenciaMovimiento()
MovimientoUsuario
#actualizarMovimiento()
+calcularEstadoDireccionPYM()
MovimientoArtificial
-secuenciaAcciones: ArrayString
-secuenciaDuraciones: ArrayInteger
-secuenciaNiveles: ArrayInteger
-accionActual: Integer
-contDuracion: Integer
-esInfluenciable: Boolean
#actualizarMovimiento()
+calcularEstadoDireccionPYM()
+influenciaMovimiento()

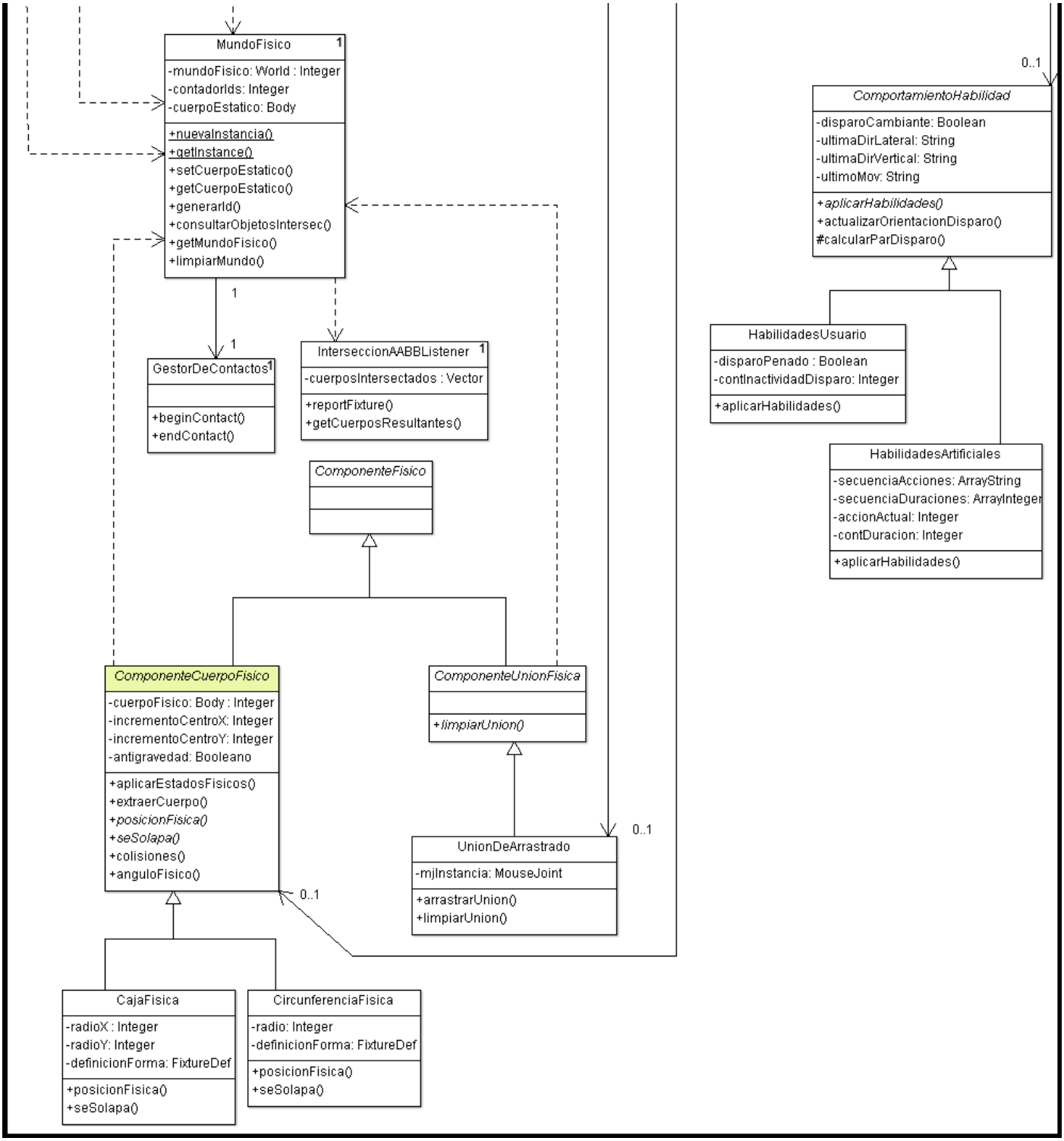


Figura 4.14: Diseño de las estructuras internas de la capa de dominio del componente de actividad *ActividadNivel*.

La capa de presentación, como su nombre indica, se encarga de definir como se le presentan al usuario los diferentes *frames* del nivel.

Por otro lado, la capa de Dominio se encarga de simular el nivel. En "Terrestrial Blast!" un nivel no es más que un conjunto de Objetos Complejos, los cuales están representados por la clase *ObjetoComplejo*, esta aparece en el diagrama de clases en color verde.

Por ejemplo, un escenario es un objeto complejo que contiene la matriz cuyas posiciones contienen la baldosa que las representa visualmente; pues en "Terrestrial Blast!" se utilizan escenarios compuestos de *tiles* o baldosa; pero también está representado por varios objetos complejos que contienen los diferentes rectángulos físicos que representan las paredes físicas de este escenario.

De igual forma, todas las entidades que componente un nivel: enemigos, personaje principal, puertas, etc. Están representados por un objeto complejo.

Al iniciar la reproducción de un nivel, se lee un fichero XML el cual contiene la definición de todos los objetos complejos que constituyen el nivel. Se crea una instancia de *ObjetoComplejo* que represente a cada uno de estos objetos y desde la clase *GameThread* se lleva a cabo el bucle del juego, que va actualizando el estado de los objetos complejos a cada *frame* que pasa a partir de la entrada que recibe por parte del usuario y la Inteligencia Artificial de los propios objetos.

Esto puede generar confusión a la hora de definir el trabajo que debe hacer la capa de Presentación y el trabajo que debe hacer la capa de Dominio.

Pues como se ha dicho, la primera se encargará de la presentación de los diferentes *frames* generados durante la simulación del videojuego, pero no de la generación del *frame* ni, por tanto, de la apariencia gráfica ni sonora de los objetos complejos que aparezcan en dicho *frame*. Pues aunque un objeto utilice una textura que permite presentarlo a ojos del usuario, esta no es más que un atributo del objeto, por tanto, se ha considerado que debe gestionarse desde la capa de dominio.

Lo mismo ocurre con los sonidos que un objeto pueda emitir, son atributos propios que lo definen a nivel lógico también y, por tanto, su reproducción se gestiona desde el dominio.

En resumen, únicamente es jurisdicción de la capa de Presentación decidir cómo se representará la imagen resultado de la creación del *frame* (con márgenes negros o no, con unas dimensiones concretas, etc.), juntamente con recibir la entrada del usuario y entregársela a la capa de Dominio en un lenguaje que esta entienda y sin olvidarnos de la respuesta a las peticiones del Sistema Operativo Android que toda Actividad debe atender, como son la petición de pausa de la aplicación o la petición de eliminación de la actividad.

En cuanto a los patrones utilizados, en esta primera capa de Presentación, se utiliza el patrón Controlador. De esta forma el componente Actividad Nivel se verá como una caja negra, con un único controlador a través del cual se hacen peticiones. La clase que hace de controlador, como en todo controlador de Actividad, es la que implementa a la propia actividad, en este caso *ActividadNivel* (los controladores aparecen pintados de color azul en el diagrama).

En la capa de Dominio se ha aplicado mayor variedad de patrones. En primer lugar, como con la capa anterior, el patrón controlador permite que esta capa sea vista como una caja negra y solo se tenga conocimiento acerca de las dos clases que hacen de controlador. En este caso, se han utilizado dos clases que ya tenían una función dentro de la capa, en lugar de crear dos clases para hacer la función exclusiva de controlador. Pues ambas clases son muy generales y el haber definido nuevas únicamente conseguiría empeorar la eficiencia y la claridad sin llegar a aportar mejoras notables en la extensibilidad, portabilidad o reusabilidad.

Los dos controladores son *GameThread* y *RenderPersonalizado*. La primera clase es la que dirige el *thread* de juego y, por tanto, se encarga de recibir la entrada, procesar las lógicas y físicas de los diferentes objetos complejos en cada *frame* y dar permiso para llevar a cabo la generación de los diferentes *renders*. En cambio, la segunda, es la que se ejecuta dentro del *thread* de *renderizado*, y contiene las operaciones que dan respuesta a las diferentes llamadas que el S.O. Android hace cuando quiere crear el *render* de un *frame* determinado mediante el uso de la librería de gráficos OpenGL ES.

En resumen, *RenderPersonalizado* se encarga de la generación de cada uno de los *renders* a partir del estado de cada objeto complejo procesado previamente por *GameThread*. Para conocer más acerca de los diferentes *threads* que se utilizan a la hora de desarrollar videojuegos para Android, así como de la comunicación entre ellos, es aconsejable leer el apartado 6 del capítulo 2.

Nótese que estos dos controladores solo serán accedidos desde la capa de Presentación del propio componente *ActividadNivel*, pues componentes externos únicamente pueden comunicarse con el componente en cuestión mediante el controlador de la capa de Presentación *ActividaNivel*.

Esto provoca que no sea necesario utilizar el patrón Singleton sobre estos controladores. Ni en los dos controladores de la capa de Dominio, pues todas las clases de la capa de Presentación pueden compartir las instancias únicas de ambas clases sin necesidad de nuevos acoplamientos evitables mediante este patrón. Ni en el controlador de la capa de Presentación, pues como se ha explicado en el apartado anterior, las actividades se comunican entre ellas únicamente indicando el nombre de la clase que representa a la actividad, sin necesidad de tener la instancia, ya que es el S.O. el que decide cuando crearla.

En cambio, sí que se ha aplicado este patrón *Singleton*, en algunas clases que, dentro de la capa de Dominio, gestionan a través de una única instancia todos los accesos y reciben accesos por parte de varias clases diferentes. Si nos dedicáramos a compartir la instancia única de estas clases entre las diferentes clases que acceden a ella el número de solapamientos entre crecería, dificultando, como ya se explico en apartados anteriores, la extensibilidad y la portabilidad.

Es el caso de *MundoFisico*, clase que contiene el objeto definido por la librería Box2D para representar el mundo de físicas de cada uno de los niveles del videojuego. Se accede a esta clase para procesar un *step* o paso, es decir para saltar de un punto en el tiempo a otro actualizando el mundo físico, también para consultar colisiones entre objetos físicos, entre otras cosas.

También ocurre lo mismo con la clase *FabricaDeObjetos*, la cual se encarga de interpretar el formato en que la capa de Gestión de Datos nos entrega los atributos que definen un nivel, creando a partir de ellos los diferentes objetos complejos que conforman el nivel. Como a parte de *GameThread*, la clase *ComponenteHabilidad* también puede ordenar la carga e interpretación de un nuevo objeto complejo, el patrón Singleton nos permite el acceso a esta por ambas partes sin tener que generar acoplamientos nuevos.

Por último, la instancia única de la clase *GestorDeTexturas*, solo es accedida por la clase *RenderPersonalizado*, pero por coherencia con las demás clases con instancia única del componente se aplica el patrón *Singleton* igualmente. *GestorDeTexturas*, se encarga de registrar cada una de las texturas que se vayan a utilizar durante la construcción de los diferentes *renders*, asociándolas con OpenGL ES y devolviendo la id generada a cambio.

La gestión conjunta de todas las peticiones a través de una única instancia nos permite conseguir mayor eficiencia, pues dos objetos que empleen la misma textura compartirán id y no será necesario registrar esta textura dos veces y mantenerla en la memoria por duplicado.

Vamos a describir a continuación el patrón de diseño más importante de todos los utilizados a la hora de diseñar la aplicación, se trata del patrón **Composite** (o Object Composition) [20]. Este patrón se basa en la idea de que una clase, en lugar de definir sus propias propiedades, se compone de otras clases, las cuales en conjunto forman la definición completa del propio objeto.

En "Terrestrial Blast!" como se ha explicado anteriormente, *ObjectoComplejo* es la clase clave dentro de la capa de Dominio, pues todo nivel se compone de instancia de esta clase, y estas instancias representan a toda entidad que aparezca en el nivel. Es por esto que cada instancia puede tener unas propiedades muy diferentes.

Por tanto, implementar una jerarquía en la clase *ObjetoComplejo* donde cada subclase tuviese las propiedades concretas de cada uno de los objetos que conformarán un nivel, es una solución muy poco efectiva. Pues nos encontraríamos con clases enormes y poco claras, que dificultarían la tarea de mantenimiento y extensibilidad; por no decir que, aunque los objetos tengan propiedades diferentes, entre subconjuntos de estos objetos hay gran parte de la definición en común, por tanto, deberíamos implementar en varias subclases exactamente las mismas propiedades y operaciones, sin poder reutilizarlas.

En definitiva una solución que debemos descartar si queremos conseguir extensibilidad y reutilización en nuestra aplicación, junto con la reducción del tiempo de desarrollo.

Además, parece claro que la definición de un objeto complejo se puede dividir en varios subcomponentes claros, en concreto, son seis los subcomponentes que pueden formar un objeto complejo (en la Figura 4.14 podemos ver sus clases en color amarillo):

- Componente Visual (clase *ComponentePintable*): Se trata del componente que define los gráficos que representan visualmente a un objeto complejo, así como la forma en que estos se pintan a la hora de generar los diferentes *renders*.

- Componente Musical (clase *ComponenteMusical*): Se trata del componente que define los sonidos que un objeto complejo reproduce durante su vida.

- Componente Físico (Clase *ComponenteCuerpoFisico*): Es a través de este componente que definimos el cuerpo físico que representará a un objeto complejo concreto dentro del mundo físico. Por tanto, a través del podemos hacer que el objeto choque con el resto.

- Componente Lógico (Clase *ComponenteLogico*): Este es el componente que define la parte "espiritual" del objeto complejo. El número de golpes que soporta antes de morir, la ideología a la hora de atacar a otros objetos o la ideología a la hora de defenderse de estos mismos objetos, entre otras cosas. Dicho de otra forma, si el Componente Físico permite que el objeto tenga una entidad física dentro del nivel y que choque con otros objetos, el Componente Lógico se encarga de definir la interacción que el objeto llevará a cabo con otros una vez producido el choque.

- Componente Habilidad (Clase *ComponenteHabilidad*): Este componente se encarga de gestionar la ejecución de las diferentes habilidades que un objeto pueda llevar a cabo. Como disparar otro objeto complejo, que se creará en el momento de la ejecución del disparo, o como arrastrar a un objeto complejo que se encuentre en el escenario y que tolere dicho control.

- Componente Comportamiento (Clase *ComponenteComportamiento*): Por último, este componente define qué comportamiento tendrá el objeto complejo. Es decir, qué desplazamiento llevará a cabo su cuerpo físico (y con él su cuerpo visual) a través del escenario definido para el nivel. Y no nos olvidemos también de la ejecución de habilidades, pues este es el componente que definirá cuando y como se va a ejecutar una habilidad concreta.

Ante una división tan clara, aplicamos el patrón Composite, por tanto, un objeto complejo no se define en la propia clase *ObjetoComplejo*, sino que se compone de una combinación de los componentes anteriormente descritos para definirse a sí mismo.

Esto nos permite implementar los diferentes componentes de forma independiente y generar una gran variedad de objetos complejos diferentes mediante la combinación de las implementaciones alternativas para cada componente. De esta forma el desarrollo se acelera ya que reutilizamos al máximo el código. Y, además, conseguimos que extender el videojuego, añadiendo un nuevo tipo de objeto complejo, sea tan fácil como ir al componente al cual queremos añadirle una implementación alternativa y definir una nueva subclase.

Por tanto, este patrón consigue que la aplicación sea realmente extensible y reutilizable, pues son estas clases las que van a querer extenderse en un futuro para generar nuevos tipos de objetos complejos o entidades.

Eso sí, para no empeorar la eficiencia no se aplica el patrón tal cual está definido por defecto, ya que en la definición por defecto todos los componentes se deberían tratar de forma genérica desde la clase *ObjetoComplejo*.

Debido a la diferente función de cada componente y de la importancia en la comunicación entre estos componentes (por ejemplo, el cambio en la posición del cuerpo físico del objeto complejo provoca un cambio en la posición donde se pinta su parte visual dentro del escenario), es ObjetoComplejo el que distingue entre los diferentes tipos de componentes y se encarga de gestionar su comunicación para que todos combinan formando realmente un objeto unido. Además, esto nos permite asegurarnos de que cada objeto complejo solo tenga, como máximo, un componente de cada tipo.

En caso de utilizar la definición original del patrón deberíamos diseñar un mecanismo de comunicación genérico entre todos los componentes que empeoraría muchísimo la eficiencia a cambio de ninguna mejora en la extensibilidad, portabilidad o reutilización.

Como se ha dejado entrever, cada componente desemboca en una jerarquía de subclases, donde cada subclase es una implementación diferente del componente, una forma diferente de hacer las funciones que ese componente debe realizar.

Eso sí, *ObjetoComplejo* trabaja de forma genérica con la superclase de cada componente, lo que nos permite crear nuevas subclases extendiendo las posibilidades de personalización del objeto complejo sin necesidad de alterar prácticamente la clase *ObjetoComplejo*. Y decimos "prácticamente" porque es inevitable que a la hora de crear un objeto complejo se deba indicar la subclase a la que cada componente pertenece a la hora de crear su instancia, antes de meterla en un contenedor de la superclase de dicho componente.

Para poder trabajar de forma genérica con la superclase de un componente, hay una serie de operaciones en cada componente que son llamadas desde *ObjetoComplejo* para llevar a cabo la interacción entre estos componentes. Estas operaciones normalmente se implementan en la superclase, definiendo un comportamiento por defecto; y en caso de querer modificar este comportamiento, desde una subclase se redefine la operación.

Pero hay excepciones, pues algunas superclases son abstractas, lo que significa que nunca puede crearse una instancia de esta superclase, solo de alguna de las subclases. Esto es útil cuando la clase padre de una jerarquía mantiene información común, pero necesita siempre de la definición de una parte específica para tener sentido, la cual le aporta cada subclase definida.

En estos casos se aplica el patrón **Plantilla** [19]. Este nos dice que debemos declarar las operaciones que tendrán el comportamiento específico en la superclase, como operaciones abstractas, es decir, indicamos solo la cabecera. De esta forma será obligatoria la definición de estas operaciones en cada una de las subclases y estas operaciones podrán ser llamadas desde otras operaciones de la propia superclase o desde fuera de la superclase como si en ella estuvieran implementadas.

La aplicación de este patrón es útil por dos razones:

- En primer lugar, cuando utilizamos la instancia de una subclase a través de un contenedor de la superclase solo podemos acceder a las operaciones de esta última clase. Así que definiendo estas operaciones como abstractas es la única forma que tenemos de que *ObjetoComplejo* pueda utilizar las operaciones especificas y, al mismo tiempo, tratar la jerarquía de forma genérica, sin pararse a distinguir entre subclases.

  Hay que tener en cuenta que una clase con operaciones abstractas debe ser abstracta, de ahí que sepamos que toda operación abstracta de la superclase va a tener siempre una implementación en una instancia real. Pues recordemos que las clases abstractas no pueden tener instancias.

  Un ejemplo, sería la clase *ComponenteCuerpoFisico*, que es el único componente de *ObjetoComplejo* que tiene una superclase abstracta. Una variable del tipo *ComponenteCuerpoFisico* solo podrá tener una instancia de *CajaFisica* o una de *CircunferenciaFisica*, pues no puede existir una instancia de la superclase. Y, ambas clases han implementado las operaciones abstractas, pues es obligatorio. En resumen, *ObjetoComplejo* podrá utilizar estas operaciones sin problema y sin saber de qué tipo es la instancia que hay dentro del contenedor del tipo *ComportamientoCuerpoFisico*.

- Y, en segundo lugar, hay casos en los que el comportamiento específico de cada subclase se combina con un comportamiento común. Mediante la combinación de operaciones normales y operaciones abstractas en la superclase que este patrón nos aconseja, podemos hacer que la parte común se implemente una sola vez en la superclase y luego se reutilice en cada subclase. En caso contrario, tendríamos que repetir la parte común en cada subclase junto con el comportamiento específico, pues la misma operación sería redefinida individualmente en cada subclase.

Nótese, que cuando hablamos de contenedor, nos referimos al tipo de la variable donde se introduce una instancia concreta. Y es que en Java, cualquier instancia de una jerarquía, sea de la superclase o de una subclase, se puede introducir dentro de un contenedor del mismo tipo que la superclase, sin que quien utilice esta variable tenga conciencia de la clase real a la que pertenece la instancia.

Por último, decir que en el diagrama de clases de la Figura 4.14 podemos ver otras jerarquías que también aplican el patrón Plantilla, aparte del componente antes citado. Y es que este patrón es imprescindible para conseguir mayor extensibilidad y reutilización.

5. Los dos niveles de abstracción de “Terrestrial Blast!”

Haciendo un resumen de lo ya explicado, podemos recordar que todo nivel del videojuego “Terrestrial Blast!” se compone de varios objetos complejos. A su vez, cada uno de estos objetos se define mediante una combinación de varios tipos de componentes, de forma que el objeto complejo puede tener, para cada tipo de componentes, una implementación o ninguna.

Pero es que además, también hemos visto como los componentes se definen individualmente en forma de jerarquía, lo que nos permite tener implementaciones alternativas para cada componente, que lleven a cabo las mismas responsabilidades de diferentes formas.

A todo esto tenemos que añadir que cada clase de las jerarquías de componente, se ha desarrollado intentando que sea lo más genérica posible dentro de su modo de implementar las responsabilidades que tiene que satisfacer. De forma que, a través de los parámetros que recibe la instancia de una subclase o superclase de componente durante su creación, podemos personalizar el modo en que esta clase hará su trabajo.

Esto nos permite potenciar la reutilización lo máximo posible, pues lo poco en común que haya entre varias subclases se reutiliza cuando estas subclases diferentes acaban siendo una genérica.

El hecho de implementar unas clases genérica, además, nos permite ahorrar en el tiempo de desarrollo, ya que conseguimos más alternativas para construir objetos complejos en menos tiempo. Esto es debido a que, no necesitamos implementar varias subclases muy específicas, sino que con una más genérica ya tenemos suficiente.

Claro está, que esto solo tiene sentido cuando las subclases que se agrupan y se hacen una están estrechamente relacionadas en su forma de llevar a cabo las responsabilidades asociadas. Pues en caso contrario, estaríamos construyendo clases muy generales y destruyendo las propiedades positivas que conlleva el uso de una jerarquía de clases.

Normalmente, cuando desarrollamos la jerarquía de un componente surgen diferentes divisiones en subclases muy obvias y, en cada división, vemos que hay diferentes formas de llevarla a cabo variando pocos parámetros, son estas variaciones las que se acaban implementando dentro de la misma clase.

En resumen, para definir un nivel, necesitamos saber qué objetos complejos lo componen y a la vez, necesitamos saber que componentes (de qué tipo) tiene cada objeto complejo y mediante que clase de la jerarquía se implementan las responsabilidades de ese tipo de componente. Pero es que además para cada componente de cada objeto complejo necesito saber qué parámetros de entrada se le entregan, pues esto es lo que decidirá como este se comporta en pequeños aspectos de su funcionamiento.

Como vemos, la aplicación está desarrollada con el objetivo de reutilizar el máximo posible por tal de conseguir las máximas alternativas posibles a la hora de crear niveles y objetos complejos en contra menos tiempo de desarrollo mejor.

Pero, ¿Quien se encarga de definir cada uno de los niveles, con sus objetos complejos y la estructura interna de estos?

En su momento, se barajaron dos alternativas. La primera consistía en crear clases Java que definieran los diferentes niveles. Esto consistía en hacer operaciones que se encargaran de crear instancias de Objeto Complejo a partir de una información sobre cada nivel leída del propio código. Estos objetos complejos se le entregarían al *GameThread*, que sería el encargado de llevar a cabo la reproducción del nivel con ellos. Pues *GameThread* solo necesita un conjunto de *ObjetosComplejos* para reproducir el nivel; a través de ellos sabe cuándo se ha completado el nivel, cuándo se ha fracasado en el nivel o cualquier información que necesite durante la reproducción del nivel.

La ventaja de este método es que la carga de un nivel es muy rápida, pues al estar todo en las clases Java y, por tanto, en memoria principal, no tenemos que acceder a ficheros con la información, lo cual llevaría un tiempo considerable.

Pero los inconvenientes tienen más peso, pues el introducir información estática en el propio código, es una práctica desaconsejada, ya que provoca que el código sea menos claro y que para hacer pequeños cambios, como son el añadir niveles o el cambiar información de estos se debe tocar el código Java. Es decir, la extensibilidad y la portabilidad se ven muy afectadas negativamente por esta forma de definir los diferentes niveles.

Así que en su lugar, se ha optado por implementar la segunda alternativa. Esta alternativa consiste en que, cada nivel y cada tipo de objeto complejo (fruto de una combinación de componentes concreta) se definan en ficheros externos a las clases Java mediante el uso del lenguaje XML [21].

El funcionamiento es el siguiente, en la capa de Gestión de Datos hay un mecanismo capaz de leer ficheros XML que sigan el formato que se exige para definir un nivel u objeto complejo. Este mecanismo permite que la clase *FabricaDeObjetos,* que podemos ver en la Figura 4.14, obtenga cuando lo desee un vector con todos los datos extraídos del fichero leído, en un formato que esta clase sabe interpretar. A partir de estos datos, *FabricaDeObjetos* genera los objetos complejos que componen el nivel el cual *GameThread* le ha pedido que cargue. Devolviendo los *ObjetosComplejos* generados a *GameThread* este puede llevar a cabo la ejecución del nivel sin problemas. Todo este mecanismo se explica visualmente en la Figura 4.15.

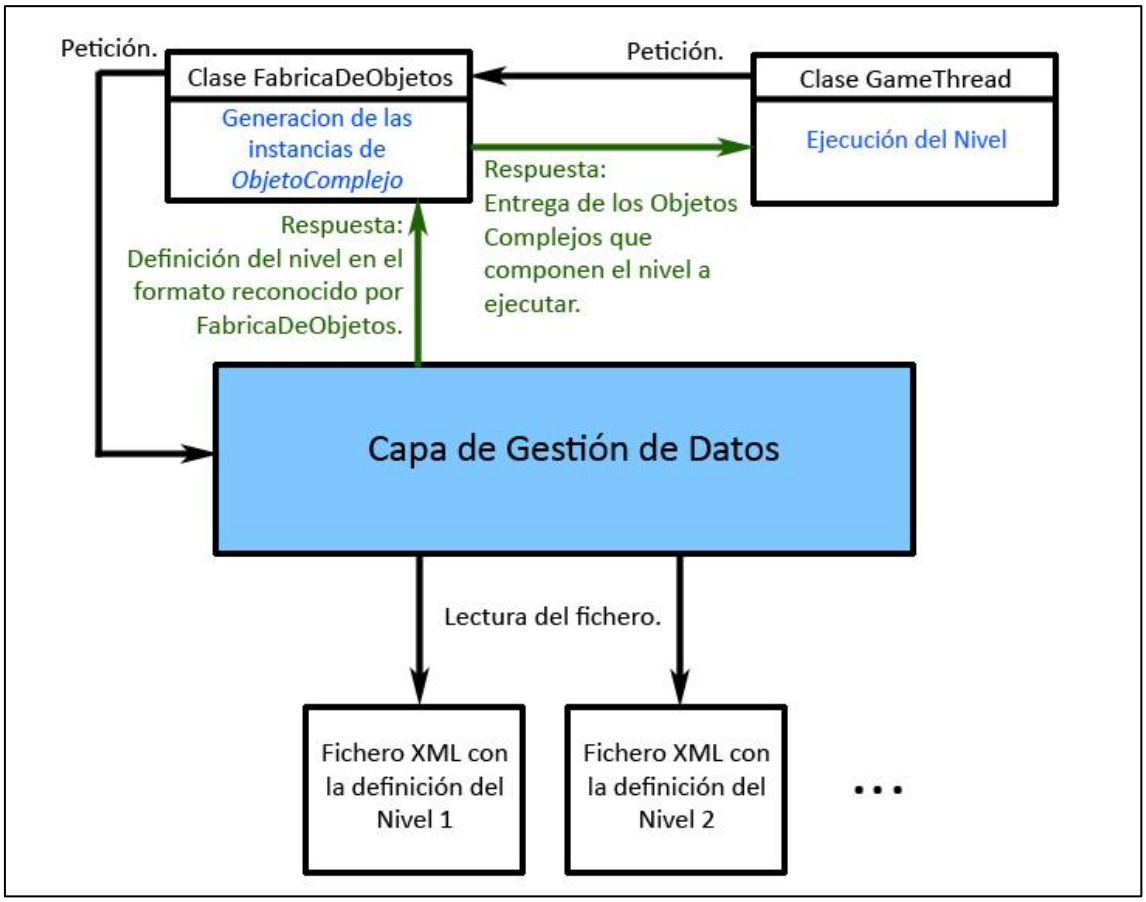


Figura 4.15: Esquema con el mecanismo de comunicación entre: la capa de Gestión de Datos, la capa de Dominio del componente de actividad *ActividadNivel* y los ficheros XML. Este mecanismo tiene como objetivo el cargar niveles y objetos complejos del videojuego definidos en ficheros XML.

Este funcionamiento es el que genera un acoplamiento entre la capa de Gestión de Datos y el componente *ActividadNivel*, pues esta capa lee la información que define a todos los componentes que un objeto complejo puede utilizar y a todas las implementaciones alternativas posibles para estos componentes y los guarda en un formato que *FabricaDeObjetos* pueda entender. Y este es el motivo por el cual, una modificación o una ampliación de las jerarquías de componente dentro del componente de actividad *ActividadNivel,* puede requerir el añadir código a la capa de Gestión de Datos.

A pesar de todo, se ha intentado evitar que el acoplamiento sea problemático y, por ello, los controladores de la capa de Gestión de Datos solo son accedidos desde clases muy concretas y generales, para evitar así un acoplamiento cíclico, el cual sí que podría ser un problema. Por tanto, solo se trata de un acoplamiento leve e inevitable.

Este método nos ofrece numerosas ventajas, pues quedan por tanto diferenciados dos niveles de abstracción claros a la hora de desarrollar el videojuego:

- El nivel más bajo está formado por la estructura de componentes antes descrita. Ampliar la aplicación conllevará trabajar a este nivel de abstracción cuando las alternativas a los componentes, ya implementadas, no cumplan con las responsabilidades que al componente le corresponden de la forma en que queremos hacerlo. En este caso deberemos crear una nueva subclase, que implemente la forma deseada. Es aconsejable, además, que esta clase sea algo genérica, dando varias posibilidades de configuración a quien la utilice desde un nivel de abstracción mayor.

  Una vez implementada, el desarrollador deberá asegurarse de que el mecanismo de la capa de Gestión De Datos sea capaz de extraer de un fichero XML la definición de un componente definido mediante esta nueva implementación. Esto exigirá hacer pequeños cambios en algunos casos.

  Y, por último, el desarrollador tendrá que asegurarse de que *ObjetoComplejo* sepa crear una instancia de esta nueva subclase si los parámetros de entrada se lo piden, para después meterla en un contenedor de componente genérico.

  Como vemos son tres pasos que están muy marcados y claros, así que la extensión de la aplicación con nuevas subclases es relativamente fácil.

- Pero, en la mayoría de los casos, podremos trabajar con el nivel de abstracción mayor que los ficheros XML nos ofrecen. Y es que, en caso de que con las alternativas para cada componente implementadas podamos definir todos los objetos complejos que conforman el nivel que queremos crear, tan solo deberemos crear un fichero con la definición de dicho nivel, utilizando el formato especifico exigido por la aplicación.

  Conseguimos de esta forma una extensibilidad y portabilidad muy altas, pues para cualquier cambio en un nivel del videojuego o cualquier creación de un nivel (posteriormente a la implementación de nuevas alternativas para un componente o no), tan solo debemos tocar el fichero XML y en ningún momento tenemos que ir a mirar el código java, el cual ya ha sido probado y debería funcionar correctamente. Además, trabajar con el lenguaje XML es mucho más fácil que trabajar con Java.

Pero esta alternativa también tiene inconvenientes, y es que leer un fichero XML es mucho más ineficiente que acceder a memoria principal. Pero este problema desaparece si, tal y como ocurre en “Terrestrial Blast!”, la carga de todos los ficheros XML se lleva a cabo durante la carga del nivel, momento en el que no importa que tardemos un tiempo más si después la ejecución del nivel es lo suficiente fluida.

Además, para objetos complejos que se crean dinámicamente durante el nivel, se ha creado un mecanismo con el cual solo es necesario acceder a los ficheros donde están definidos en la primera carga dinámica, puesto que después la definición permanecerá en memoria principal y el acceso a ella será rápido. Aun así, para casos extremos, se dan mecanismos para que el desarrollador, a través del fichero XML donde se define el nivel, se pueda asegurar de que toda definición necesaria durante la ejecución de este nivel se va a cargar en memoria principal durante la carga del nivel.

Por último, para saber con detalle qué responsabilidades tiene cada componente, cuáles son las implementaciones alternativas de cada componente ya implementadas y cómo llevan estas a cabo sus tareas, así como el formato que se debe seguir para crear un fichero XML que represente un nivel, entre otra información relacionada, es preciso leer el *Anexo I*.

6. Los tres *threads* que componen el videojuego "Terrestrial Blast!"

Tal y como vimos en el apartado 5 del capítulo 3 que ocurría con el videojuego anterior, en este caso, durante la ejecución de un nivel, es decir, cuando la actividad ActividadNivel se encuentre en primer plano, también se estarán ejecutando tres *threads* diferentes simultáneamente. Los cuales en conjunto darán vida al nivel del videojuego en reproducción y permitirán una experiencia satisfactoria para usuario.

La funciona de cada uno de los tres *threads* es la siguiente:

- El UI Thread es el que recibe la entrada del usuario y se encarga de presentar el *render* entregado por el GL Thread decidiendo sus dimensiones y su *aspect ratio*.

- Como el UI Thread debe estar desocupado para poder recibir la entrada del usuario correctamente, es el *thread* de juego el que ejecuta el bucle del videojuego.

  Primero recogerá la entrada que el usuario haya podido introducir. Después se encargará de procesar la entrada sobre los objetos complejos que la interpreten así como la inteligencia artificial de los objetos complejos que tengan. Seguidamente se encargará de simular el transcurso del tiempo entre el *frame* anterior y el *frame* al cual estamos dando respuesta a nivel de físicas y de lógicas, teniendo en cuenta el comportamiento que los objetos complejos han manifestado en el paso anterior. Y, por último, dará permiso al GL Thread para que lleve a cabo la construcción del *render* a partir de la posición y la textura que cada componente visual de cada objeto complejo presenta una vez procesado este *frame*.

- Por último, tenemos el GL Thread, este es un *thread* dirigido por el S.O. Android, el cual va llamando a una serie de operaciones definidas en la clase *RenderPersonalizado*, clase que implementa la interface de *Renderer*. Periódicamente el S.O. llamará a la operación *onDraw()* de esta clase para que lleve a cabo la creación de un *render*, el cual le será presentado al usuario de la forma que el UI Thread haya decidido.

Como vemos las responsabilidades de cada *thread* son las mismas que estos tenían en el videojuego anterior, pues es algo impuesto por el propio S.O. Android, que para que el videojuego funcione correctamente exige la división en *threads* de la forma anteriormente descrita.

Es necesario precisar que, para sincronizar el movimiento y la interacción de los diferentes elementos del videojuego con el transcurso del tiempo, se ha utilizado de nueva la sincronización por *framerate*, ajustada a una cantidad de 30 *frames* por segundo. Lo que viene a significar que el mundo creado por el videojuego se representará discretizando el tiempo en 30 instantes por segundo. En cada instante se presentará en pantalla una nueva captura del nivel del videojuego, donde los diferentes elementos estarán en la posición que les corresponda en ese instante, lo que nos llevará a generar una animación que a ojos del usuario será como si realmente el nivel presentado tuviese vida.

Por tanto, el bucle del juego y la creación de un *render* tienen que ser lo suficientemente rápidos, como para que se pueda llegar a ejecutar cada uno de los dos procesos 30 veces por segundo.

Para más información acerca de la sincronización del bucle del juego, así como del papel que cada *thread* realiza es aconsejable leer el apartados 6 del capítulo 2 y el apartado 5 del capítulo 3.

6.1. La comunicación entre los tres *threads*

En este punto también encontramos gran similitud entre los dos videojuegos. Pues las responsabilidades de los tres *threads* son las mismas y esto nos lleva a tener que comunicarnos prácticamente de la misma forma.

En la Figura 4.16 podemos ver el diagrama de clases donde se han bordeado las diferentes clases que son tratadas desde cada *thread*. Podemos ver como igual que ocurría con el videojuego anterior, hay clases a las que se accede desde varios *threads*, como es el caso de *ComponentePintable*. En estas clases, lo habitual es que unas operaciones se accedan exclusivamente desde un *thread* y otras exclusivamente desde el otro.

Por otro lado, habría que dejar claro que el diagrama de la Figura 4.16 no presenta todas las clases del componente *ActividadNivel*, ya que como podemos ver en la figura x+888, el diagrama original es más amplio. Todas las clases que faltan se utilizan exclusivamente dentro del Game Thread y esto es lo que simbolizamos con los puntos suspensivos añadidos en la parte inferior de la figura.

Centrándonos en la comunicación entre *threads*, a la hora de inicializar estos hilos de ejecución los pasos realizados son prácticamente iguales a los que llevábamos a cabo en el videojuego anterior. Con la salvedad de que en este caso algunas clases han cambiado de nombre, de forma que *RenderPersonalizado es* la clase que implementa la interface *Renderer*, mientras que *ViewSuperficieGL* es la clase que hereda de *GLSurfaceView*.

En cuanto a la comunicación entre UI Thread y Game Thread, es prácticamente igual que en el videojuego anterior, con una única diferencia, y es que ahora el buffer donde se depositarán las entradas del usuario se encuentra dentro de la clase GameThread. Mediante este cambio conseguimos que la capa de Dominio no se acople con la capa de Presentación y, por tanto, el acoplamiento solo sea de arriba abajo, evitando un acoplamiento cíclico. Esta medida potencia la portabilidad.

Si nos centramos ahora en la comunicación entre Game Thread y Render Thread, vuelve a ser muy parecida a la comunicación que ambos *threads* tenían en el videojuego anterior.

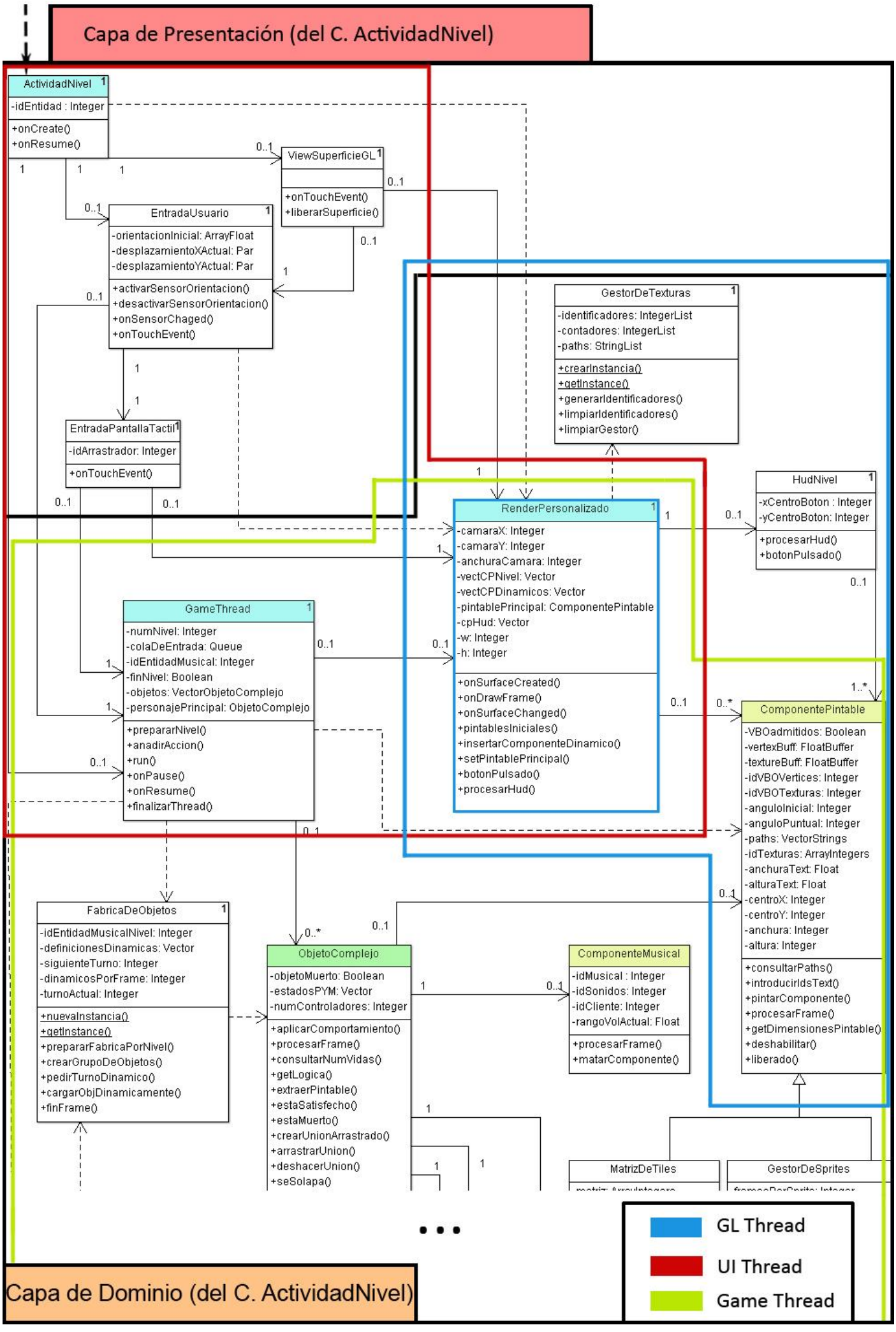
Pero en este caso la estructura de clases esta mucho más generalizada, de forma que Game Thread solo trabaja con objetos complejos, de forma genérica, sin saber acerca de la definición interna de estos objetos, la cual si que conocíamos en el videojuego anterior.

Lo mismo ocurre con *RenderPersonalizado*, este solo trabaja con la clase *ComponentePintable*, de forma que si un objeto complejo no tiene componente de este tipo ni siquiera sabe de su existencia. En el anterior videojuego, la clase que asumía el mismo rol se asociaba directamente con todos los objetos, distinguiendo además entre cada tipo de objeto que el videojuego presentaba, mientras que ahora únicamente conoce el componente visual de estos objetos y ni siquiera sabe a qué clase de la jerarquía de *ComponentePintable* pertenece cada componente.

Estas mejoras nos permiten que el *thread* de *renderizado* acceda a menos clases del diagrama, en comparación al anterior videojuego. Pero sobretodo, la importancia de estas mejoras radica en el aumento radical de la extensibilidad y portabilidad del videojuego. Pues gracias a construir un diagrama de clases tan genérico, hacer cambios en las diferentes clases o añadir nuevas clases es **muchísimo** más fácil, pues tenemos que preocuparnos de menos solapamientos. En especial es muchísimo más fácil el añadir subclases a los diferentes componentes de objeto complejo, que es lo que querremos hacer en la mayoría de extensiones que llevemos a cabo sobre el videojuego.

Por tanto, podemos concluir que "Terrestrian Blast!" ofrece mejoras en la estructuración del diagrama de clases y en la comunicación entre *threads* respecto al anterior videojuego, que facilitan muchísimo la extensibilidad y portabilidad de la aplicación y, porque no decirlo, la claridad y simplicidad del código, cosa que también es siempre muy importante.

Figura 4.16: Diagrama de clases, del componente de actividad *ActividadNivel*, donde se indican las clases que son accedidas desde cada uno de los tres *threads*. Estos tres *threads* se ejecutan concurrentemente durante la ejecución de "Terrestrial Blast!".


Capa de Presentación (del C. ActividadNivel)
ActividadNivel
-idEntidad : Integer
+onCreate()
+onResume()
ViewSuperficieGL
+onTouchEvent()
+liberarSuperficie()
EntradaUsuario
-orientacionInicial: ArrayFloat
-desplazamientoXActual: Par
-desplazamientoYActual: Par
+activarSensorOrientacion()
+desactivarSensorOrientacion()
+onSensorChaged()
+onTouchEvent()
EntradaPantallaTactil
-idArrastrador: Integer
+onTouchEvent()
GestorDeTexturas
-identificadores: IntegerList
-contadores: IntegerList
-paths: StringList
+crearInstancia()
+getInstance()
+generarIdentificadores()
+limpiarIdentificadores()
+limpiarGestor()
RenderPersonalizado
-camaraX: Integer
-camaraY: Integer
-anchuraCamara: Integer
-vectCPNivel: Vector
-vectCPDinamicos: Vector
-pintablePrincipal: ComponentePintable
-cpHud: Vector
-w: Integer
-h: Integer
+onSurfaceCreated()
+onDrawFrame()
+onSurfaceChanged()
+pintablesIniciales()
+insertarComponenteDinamico()
+setPintablePrincipal()
+botonPulsado()
+procesarHud()
HudNivel
-xCentroBoton : Integer
-yCentroBoton: Integer
+procesarHud()
+botonPulsado()
GameThread
-numNivel: Integer
-colaDeEntrada: Queue
-idEntidadMusical: Integer
-finNivel: Boolean
-objetos: VectorObjetoComplejo
-personajePrincipal: ObjetoComplejo
+prepararNivel()
+anadirAccion()
+run()
+onPause()
+onResume()
+finalizarThread()
ComponentePintable
-VBOadmitidos: Boolean
-vertexBuff: FloatBuffer
-textureBuff: FloatBuffer
-idVBOVertices: Integer
-idVBOTexturas: Integer
-anguloInicial: Integer
-anguloPuntual: Integer
-paths: VectorStrings
-idTexturas: ArrayIntegers
-anchuraText: Float
-alturaText: Float
-centroX: Integer
-centroY: Integer
-anchura: Integer
-altura: Integer
+consultarPaths()
+introducirIdsText()
+pintarComponente()
+procesarFrame()
+getDimensionesPintable()
+deshabilitar()
+liberado()
FabricaDeObjetos
-idEntidadMusicalNivel: Integer
-definicionesDinamicas: Vector
-siguienteTurno: Integer
-dinamicosPorFrame: Integer
-turnoActual: Integer
+nuevaInstancia()
+getInstance()
+prepararFabricaPorNivel()
+crearGrupoDeObjetos()
+pedirTurnoDinamico()
+cargarObjDinamicamente()
+finFrame()
ObjetoComplejo
-objetoMuerto: Boolean
-estadosPYM: Vector
-numControladores: Integer
+aplicarComportamiento()
+procesarFrame()
+consultarNumVidas()
+getLogica()
+extraerPintable()
+estaSatisfecho()
+estaMuerto()
+crearUnionArrastrado()
+arrastrarUnion()
+deshacerUnion()
+seSolapa()
ComponenteMusical
-idMusical : Integer
-idSonidos: Integer
-idCliente: Integer
-rangoVolActual: Float
+procesarFrame()
+matarComponente()
MatrizDeTiles
GestorDeSprites
GL Thread
UI Thread
Game Thread
Capa de Dominio (del C. ActividadNivel)

6.1.1. Sincronización entre el *thread* de juego y el GL Thread

A pesar de la similitud entre ambos videojuegos en cuanto a la división en *threads*, hay un apartado que realmente difiere mucho entre un videojuego y otro. No es otro que el de la sincronización entre el *thread* de juego y el GL thread.

Como sabemos, estos *threads* llevan a cabo unas funciones muy ligadas entre ellos, de hecho, en el desarrollo de un videojuego normal, lo que aquí llamamos *thread* de *renderizado* no es más que un último paso en el bucle del videojuego, el cual se ejecuta de forma secuencial respecto al resto.

Una ejecución paralela entre el *thread* de juego y el *thread* de *renderizado*, puede parecer una mejor opción, pues de esta forma podemos aprovechar más la CPU, pues las operaciones de OpenGL ejecutadas durante la creación del *render*, se ejecutan en la GPU dejando a la CPU libre. Si tenemos otro *thread* que se encarga de hacer otra tarea simultáneamente, este podrá aprovechar la CPU durante el tiempo que este libre.

En el videojuego anterior, los dos *threads* funcionaban a su libre albedrio, sin sincronizarse entre ellos, aprovechando al máximo los recursos. Pero realmente, cuando tenemos un videojuego que requiere un mayor uso de recursos, como es el caso de "Terrestrial Blast!", nos damos cuenta que ambos *threads* deben estar sincronizados.

Pues cuando se lleva a cabo la creación del *render*, si el videojuego está modificando la posición de los diferentes objetos complejos al mismo tiempo que el GL Thread esta leyéndola para pintar la textura en un punto concreto del escenario, puede darse el caso de que algunos componentes visuales se pinten en la posición que tenían en el *frame i*, cuyo *render* estamos generando, pero otros se pinten en la posición que ocupan en el *frame i+1*, si este ya está siendo procesado por el *thread* de juego.

Las consecuencias de este libre albedrio son fallos gráficos en el pintado de algún componente visual o la no sincronización entre algunos objetos complejos del nivel del videojuego que tienen algún tipo de relación: como puede ser entre el personaje que controla el usuario y el *scroll*, que ante una desincronización se genera un parpadeo extremadamente molesto para el usuario.

En el videojuego anterior, se daban igualmente estos problemas, pero al requerir pocos recursos y tener pocas entidades en el escenario, las interferencias no eran apreciables por el usuario.

Y bien, una vez sabemos que el GL Thread y el *thread* de juego siempre deben estar sincronizados vamos a ver qué alternativas tenemos a la hora de llevar a cabo esta sincronización:

- En primer lugar, tenemos la opción de implementar una sincronización secuencia. Mediante este tipo de sincronización, el GL Thread solo generaría un *render* cuando el *thread* de juego se lo ordenase, y hasta que no se acabara la generación del *render* no se podría proceder en el *thread* de juego.

  Esta solución es la más simple en cuanto a desarrollo, por tanto, nos ahorramos errores en la implementación, aunque, a cambio no podremos utilizar la CPU durante el tiempo que este libre en mitad de la creación de un *render*.

- La segunda opción, consiste en que las posiciones de los diferentes objetos complejos, no se guarden en un mismo sitio, sino que se haga una copia independiente con todas las posiciones de todos los componentes visuales en un *frame* dado y se le entregue al GL Thread junto con el permiso para comenzar la creación del *render*, al finalizar el resto del procesado del *frame*. Mientras el GL Thread crea el *render*, el Game Thread podrá seguir ejecutándose modificando las variables de posición original de cada objeto complejo.

  Este método consigue que podamos aprovechar la CPU durante el tiempo que permanece libre y, por tanto, la eficiencia es mayor. Pero no se consigue la máxima eficiencia posible, porque debemos duplicar las variables con la posición de cada objeto complejo que tiene componente visual y la generación del buffer con las posiciones duplicadas consume tiempo de CPU. Aunque el consume de CPU compensa con el que después nos ahorramos gracias a la simultaneidad entre los dos *threads*.

- La última de las alternativas consiste en llevar a cabo una sincronización por componente visual. De forma que cuando Game Thread quiera procesar un objeto complejo le preguntará si el rende *thread* ya le ha pintado en el *render* del *frame* anterior, en caso afirmativo procederá a procesarlo y en caso negativo se pausará hasta que el Render Thread consulte esa información. Lo mismo ocurre con el Render Thread que antes de consultar la posición de un componente visual debe preguntarle a este si se encuentra a la espera de que *rendericen* su posición y, en caso contrario debe pausarse hasta que la respuesta sea afirmativa.

  Mediante este mecanismo de sincronización conseguimos algo más de eficiencia que con el resto de mecanismos, pues es una sincronización a muy bajo nivel. Eso sí, a cambio el tiempo de desarrollo es mayor, pues una sincronización de este tipo es complicada de implementar.

A la hora de desarrollar “Terrestrial Blast!” se opto por la tercera opción y como conclusión se puede decir que es la peor de todas y es muy desaconsejable utilizarla.

El motivo principal es la dificultad de desarrollar el mecanismo de sincronización objeto a objeto, pues aunque no pueda parecer demasiado complejo, existen muchos aspectos a tener en cuenta, como son: el orden en que ambos *thread* procesan componentes visuales, el mecanismo de creación de componentes visuales dinámicamente, etc. etc.

A la hora de llevar a cabo el desarrollo de la mecánica de sincronización aparecieron numerosos errores y una vez solventados comenzaron a aparecer otros errores en momentos muy concretos que pausaban el juego, ya que ambos *threads* se quedaban esperando el uno al otro. Para solucionar estos errores hicieron falta muchas horas y de análisis del comportamiento.

Finalmente, se ha conseguido asegurar que el mecanismo funciona correctamente en “Terrestrial Blast!” Aunque para asegurar un mejor funcionamiento, la sincronización objeto a objeto solo la realiza el *thread* de juego, pues el *thread* de *renderizado* recibe una orden cada vez que puede comenzar a generar un nuevo *thread*. Esta modificación reduce bastante las posibilidades de error.

Además, la mejora de eficiencia conseguida con la implementación de esta método es prácticamente inapreciable, sobretodo en comparación con otras mejoras mucho más simples llevadas a cabo que veremos en un apartado posterior y que mejoraron la eficiencia radicalmente.

Estos son los motivos por los cuales se desaconseja totalmente esta implementación de la sincronización entre ambos *threads* y se aconseja la utilización de una de las otras dos alternativas descritas.

7. Los dos espacios de pintado de Android

En “Terrestrial Blast!” la división entre espacios de pintado, así como la comunicación entre estos, es exactamente igual que en el videojuego anterior, explicada en el apartado 6 del capítulo 3.

De hecho ya se ha estado hablando de esta división cuando se ha explicado el papel que llevan a cabo individualmente, sobre la gestión de los gráficos de un nivel del videojuego, la capa de Presentación y la capa de Dominio definidas dentro del componente *ActividadNivel*.

Y es que la división entre ambas capas, entre otras cosas, representa esta división entre espacios de pintado. Eso sí salvando una excepción, pues la operación *onSurfaceChanged()*, que se encuentra en la capa de dominio, inevitablemente también trabaja sobre el espacio de pintado Android, como vimos en “Ping Pang Pung”.

Lo que sí que difiere entre los dos videojuegos, es la forma en que ambos espacios de pintado realizan su trabajo. Pues en este caso el tamaño de la ventana desde la cual se hacen fotos al nivel del videojuego, dentro del espacio de pintado OpenGL, no tiene un tamaño fijo, sino que su tamaño depende del terminal. Lo que se hace es definir una altura fija de 480 pixeles y según las proporciones de la pantalla del dispositivo se define una anchura diferente.

De esta forma conseguimos un *render* que podemos ampliar para que ocupe absolutamente toda la pantalla sin que se deforme. Y esto es precisamente lo que se lleva a cabo desde el espacio de pintado Android, pues la *view ViewSuperficieGL* y el *render* que se pinta en ella se amplía hasta cubrir toda la pantalla.

Gracias a este comportamiento desde ambos espacios de pintado evitamos dejar márgenes negros a los lados de la pantalla cuando presentamos el *render* al usuario, asegurándonos además de que este *render* no se deforme.

8. Mejoras de eficiencia en “Terrestrial Blast!”

Una vez implementada la estructura descrita en los apartados anteriores, nos encontremos con un problema. Y es que la ejecución de un nivel del videojuego a través del componente *ActividadNivel* no era lo suficientemente eficiente como para funcionar fluidamente a una tasa de 30 *frames* por segundo.

En este apartado se van a describir las medidas que se han tomado para mejorar la eficiencia de “Terrestrial Blast!”, hasta conseguir que el juego funcione fluidamente a una tasa de 30 *frames* por segundo. De hecho, como veremos a continuación, el juego podría funcionar a una tasa mayor de *frames* por segundo, pues la eficiencia ha mejorado radicalmente.

Para poder comprobar la eficiencia de nuestra aplicación, se ha hecho uso del Traceview [22], una herramienta que incluye el SDK de Android y que permite evaluar la ejecución de nuestra aplicación. Es un *profiler* y como tal nos permite saber cuántos milisegundos tarda en ejecutarse en media cada una de las operaciones de nuestra aplicación, cual es el tanto por ciento del tiempo de ejecución de una aplicación que está ocupando la ejecución de cada operación, el número de llamadas a cada operación, etc.

Rendimiento del videojuego antes de las mejoras de eficiencia

Antes de parar siquiera atención en el rendimiento de la aplicación, lo primero que se llevo a cabo fue la sincronización entre el Game Thread y el Render Thread, pues el juego durante su ejecución presentaba fallos gráficos debido al libre albedrío de ambos *threads*, el cual debe ser evitado a toda costa en cualquier aplicación. Además, el hecho de que no estuvieran sincronizados podría hacer que alguno de los dos hiciera sus funciones más veces de la cuenta.

Como hemos visto en el apartado anterior, la sincronización entre ambos *threads* es a nivel de objeto complejo, lo cual nos permite conseguir la máxima eficiencia a costa de un tiempo de desarrollo excesivamente largo y complejo. Además, una vez hecha la mejora, se probó el funcionamiento del videojuego con una sincronización menos eficiente, como es la sincronización secuencial entre ambos *threads*. La diferencia entre ambas no es nada notoria, con lo cual se desaconseja la implementación de una sincronización tan compleja.

A través del Traceview, lo primero que se observó es que el problema de eficiencia venía del *Render Thread*, pues tanto el *thread* de juego como el UI Thread estaban realizando su tarea de forma rápida y eficiente. Esto no dejaba de ser extraño, pues todo el trabajo duro se realiza en el *thread* del juego, así que el problema se debía más bien a un uso poco eficiente de la librería de gráficos OpenGL ES.

En concreto, dentro de la operación *onDrawFrame()*, que es la que se encarga de generar el *render* de cada *frame*, la operación que prácticamente abarcaba toda la creación de este era *MatriDeTiles.pintarComponente()*, es decir, la operación encargada de pintar el escenario del nivel.

Como ya se ha explicado anteriormente, en "Terrestrial Blast!" el escenario es una matriz de tiles o baldosas. Este método de representar el escenario es muy eficiente, pues a través de unas pocas baldosas (es decir, texturas cuadradas) podemos, combinándolas, generar un escenario bastante variado.

Esto nos lleva a disminuir el consumo de memoria principal, pues en lugar de tener una imagen enorme con todo el escenario de la pantalla, solo tengo un fichero donde se encuentran las texturas correspondientes a cada baldosa. De forma que las baldosas que se utilizan muchas veces por escenario solo las tengo una vez en memoria principal. Además, el no trabajar con imágenes grandes también puede aumentar la eficiencia en tiempo. Por tanto, conseguimos una mejora de eficiencia notable, tanto en tiempo como en espacio.

El problema, por tanto, venía a la hora de pintar esta matriz, pues algo no se estaba realizando correctamente. El mecanismo de pintado consistía en aprovechar el mismo mecanismo que la superclase ComponentePintable, de la cual MatrizDeTile hereda, para pintar cada una de las baldosas de la matriz. De esta forma, MatrizDeTiles recorría cada una de las baldosas y, para cada una de estas, posicionaba la superclase en el lugar que le correspondía a esta tile, asignándole las mimas medidas y llamaba a la operación *pintarGeometria()* de esta superclase. Operación que también era de las que más tiempo estaban ejecutándose durante la ejecución de un nivel.

Con este método, claro está, contra más grande fuese la matriz de tiles, más tiempo llevaba pintarla.

Reduciendo el número de accesos a los FloatBuffers

La primera mejora llevada a cabo, fue la de evitar utilizar el método *get()* sobre la variable del tipo *FloatBuffer* que contenía las coordenadas de la posición de un *ComponentePintable,* dentro de la operación *pintarGeometria()*. Ya que, viendo la tasa de uso de esta operación y el tiempo que tardaba en ejecutarse una vez, se llego a la conclusión de que algo no funcionaba correctamente en ella.

Buscando información acerca de este problema, se descubrió que la clase *FloatBuffer* no está implementada de forma eficiente en la versión 2.2 y en versiones anteriores de Android. Por tanto, los accesos o modificaciones de una variable de este tipo son muy ineficientes.

Lo primero que se intentó fue dejar de utilizar este tipo de variables, pero las operaciones OpenGL están pensadas para recibir unos parámetros del tipo *FloatBuffer* y la única alternativa son Buffer de números enteros, los cuales no cubren la necesidad de utilizar números decimales. Este es el motivo por el cual, se llevo a cabo la sustitución de todos los *get()* posibles, a cambio de la definición de un atributo de la clase *ComponentePintable* que contiene las coordenadas del centro de dicho componente.

Gracias, a esta mejora, la operación *get()* sobre el tipo *FloatBuffer* paso de ocupar aproximadamente un 24% del tiempo de ejecución del nivel a solo un 3%. Lo que propició que pintarGeometria() también pasara a ocupar menos tiempo de ejecución. Esto a su vez provoco la misma reducción en *MatriDeTiles.pintarComponente()*. Y, en última instancia la reducción de tiempo de ejecución ocupado llegó a la operación *onDrawFrame()*, que antes tardaba en ejecutarse unos 240 milisegundos, ahora tan solo tardaba unos 150 milisegundos (ver Figura 4.17).

Nombre de la operación	% CPU Total	T. Total	Llamad.+Recurs.	T. Llamada
0 (toplevel)	100,0%	3757,601	6+0	626,267
1 com/juego/terrestrial/actNivel/RendererPersonalizado.onDrawFrame	46,7%	1755,460	12+0	146,288
2 com/juego/terrestrial/actNivel/MatrizDeTiles.pintarComponente (Ljav	30,9%	1161,805	25+0	46,472
3 com/juego/terrestrial/actNivel/ObjetoComplejo.procesarFrame (Ljav	18,8%	706,969	2830+0	0,250
4 com/juego/terrestrial/actNivel/ObjetoComplejo.aplicarComportamier	17,8%	669,478	2830+0	0,237
5 com/juego/terrestrial/actNivel/ComponentePintable.pintarGeometria	16,3%	611,845	2876+0	0,213
6 com/juego/terrestrial/actNivel/Debuggeo.pintarFisicas (Ljavax/micro	16,2%	607,633	13+0	46,741
7 java/nio/ReadWriteDirectByteBuffer.putFloat (IF)Ljava/nio/ByteBufl	15,7%	591,424	13251+0	0,045
8 java/nio/FloatToByteBufferAdapter.put (IF)Ljava/nio/FloatBuffer;	13,1%	493,525	9672+0	0,051
9 com/juego/terrestrial/actNivel/ComponentePintable.modificarMapeo	12,4%	466,718	975+0	0,479
10 com/juego/terrestrial/actNivel/ComponenteCuerpoFisico.aplicarEst	8,9%	333,900	2713+0	0,123
11 org/apache/harmony/luni/platform/PlatformAddress.setFloat (IFLo	7,3%	275,749	13251+0	0,021
12 com/juego/terrestrial/actNivel/ComponentePintable.procesarFrame	6,7%	250,819	234+0	1,072
13 com/juego/terrestrial/actNivel/ComponentePintable.trasladarDesde	5,5%	206,072	234+0	0,881
14 java/util/Vector.<init> ()V	5,5%	205,486	5782+0	0,036
15 java/nio/FloatBuffer.put ([F)Ljava/nio/FloatBuffer;	5,4%	204,761	299+0	0,685

Figura 4.17: Resultados del test de eficiencia llevado a cabo por la herramienta Traceview del SDK de Android. Es un test posterior a la mejora que evita el acceso a los diferentes *FloatBuffers* empleados en la creación de los *renders* en la medida de lo posible. Podemos ver las 15 operaciones que han estado un mayor porcentaje de tiempo ejecutándose en la CPU, sumando todas sus llamadas.

En la primera columna vemos el nombre de cada operación. En la segunda se presenta el porcentaje total de utilización de la CPU. *T. Total* presenta el tiempo total consumido por esta operación, sumando los tiempos de todas sus llamadas. *Llamadas+Recursivas* nos presenta, a un lado, el número de llamadas efectuado por cada operación y, al otro lado, el número de llamadas recursivas. Por último, *T. Llamada* nos presenta el tiempo promedio (en milisegundos) que tarda la ejecución de cada una de las llamadas a la operación.

En rojo podemos ver las operaciones más importantes que se ejecutan desde el *thread* de *renderizado*. Además, la primera operación, *(toplevel)*, es una operación global que representa la ejecución de todo el test. En este caso este test duró unos 22 segundos.

Pero el objetivo con este videojuego ha sido el alcanzar los 30 *frames* por segundo, lo cual equivale a que en 33,3 milisegundos se debe ejecutar el bucle del videojuego completo, incluyendo la parte realizada en el *thread* de juego y la construcción del *render* de la que se encarga la operación *onDrawFrame()*.

Por tanto, aunque la mejora era notoria, pues la creación del *render* había pasado de tardar unos 7 *frames* a realizarse a tan solo unos 4,5 *frames*, era evidente que aun quedaba por mejorar mucho la eficiencia si realmente se quería conseguir que el videojuego funcionase de forma fluida.

Pintando solo lo que se ve

La siguiente mejora de eficiencia que se llevó a cabo es la de: no mandar a pintar en el *render* que se está construyendo los componentes pintables que no se encuentren parcial o totalmente en el área que este *render* va a presentar al usuario.

Esta mejora no se llevó a cabo desde un primer momento debido a que Open GL ES ya realiza automáticamente el denominado *clipping*, que consiste en dejar de procesar automáticamente las figuras a pintar que no aparecen en el área que se va a presentar.

Pero resulta que el *clipping* no es lo suficientemente efectivo, y haciendo manualmente una distinción entre componentes que se van a visualizar en el *render* y componentes que no van a aparecer en él, descartando estos últimos, se consigue una mejora de eficiencia bastante grande.

En primer lugar, nos ahorramos hacer una serie de operaciones necesarias antes de ordenar el pintado de los componentes en cuestión. Y, en segundo lugar, como ya se ha dicho, simplemente con la llamada a *glDrawArrays()*, operación de OpenGL a través de la cual se ordena el pintado de una geometría dada, ya estamos consumiendo tiempo de ejecución, pues parece que el *clipping* se lleva a cabo en una fase tardía del procesado del pintado (quizás debido a que los drivers no son lo suficientemente eficientes).

Con esta mejora de eficiencia, por tanto, ahora solo se pinta la parte de la matriz de tiles del escenario que aparece en el *render*. Es decir, no solo nos ahorramos el pintado de objetos complejos individuales, sino también el pintado de la mayoría de las tiles de la matriz, las cuales recordemos que se pintan individualmente como si fueran objetos complejos diferentes.

Esta mejora, nos permite por tanto, que la matriz con el escenario de un nivel pueda tener cualquier tamaño sin afectar a la eficiencia a la hora de reproducir dicho nivel. Esto es debido a que como pintamos solo el área visible, siempre se pintarán más o menos el mismo número de tiles, un número pequeño. Anteriormente, en cambio, a la que añadíamos un escenario demasiado grande la eficiencia se resentía y el número de *frames* por segundo disminuía.

Gracias a esta mejora de eficiencia, se consiguió que *onDrawFrame()* pasara a tardar tan solo unos 32 milisegundos, lo que viene a ser casi 1 *frame*. Si lo comparamos con su duración anterior, que era de 4,5 *frames*, la mejora es muy grande (ver Figura 4.18). Además, el pintado de la matriz, que es el gran protagonista en esta mejora, pasó, de tardar unos 46 milisegundos, a tan solo unos 15.

Pero estos tiempos aun no eran lo suficientemente buenos como para poder mantener la tasa de 30 *frames* por segundo. Pues recordemos que en cada *frame* no solo se tiene que crear el *render*, sino que se debe procesar el bucle del videojuego, con sus físicas, sus lógicas, etc. Teniendo en cuenta que actualmente este Game Thread tarda de media aproximadamente 8 milisegundos en procesar un *frame*, aún teníamos que reducir el tiempo de creación de un *render* unos cuantos milisegundos.

Nombre de la operación	% CPU Total	T. Total	Llamad.+Recurs.	T. Llamada
0 (toplevel)	100,0%	3829,650	6+0	638,275
1 com/juego/terrestrial/actNivel/RendererPersonalizado.onDrawFrame	47,2%	1807,466	57+0	31,710
2 com/juego/terrestrial/actNivel/MatrizDeTiles.pintarComponente (Ljav	43,5%	1664,492	114+0	14,601
3 com/juego/terrestrial/actNivel/ComponentePintable.modificarMapeo	22,3%	854,804	1733+0	0,493
4 java/nio/FloatToByteBufferAdapter.put (IF)Ljava/nio/FloatBuffer;	22,0%	842,174	15800+0	0,053
5 com/juego/terrestrial/actNivel/ObjetoComplejo.aplicarComportamier	19,3%	738,966	2950+0	0,250
6 java/nio/ReadWriteDirectByteBuffer.putFloat (IF)Ljava/nio/ByteBufl	19,2%	736,674	15799+0	0,047
7 com/juego/terrestrial/actNivel/ComponentePintable.pintarGeometria	18,8%	718,270	3185+0	0,226
8 com/juego/terrestrial/actNivel/ObjetoComplejo.procesarFrame (Ljav	18,3%	700,695	2946+0	0,238
9 org/apache/harmony/luni/platform/PlatformAddress.setFloat (IFLorg	8,7%	334,803	15799+0	0,021
10 com/juego/terrestrial/actNivel/ComponenteCuerpoFisico.aplicarEst	8,1%	309,259	2828+0	0,109
11 com/juego/terrestrial/actNivel/ComponentePintable.procesarFrame	6,9%	265,610	242+0	1,098
12 java/nio/DirectByteBuffer.getBaseAddress ()Lorg/apache/harmony	6,5%	247,764	18305+0	0,014
13 java/util/Vector.<init> ()V	5,6%	215,209	6057+0	0,036
14 com/juego/terrestrial/actNivel/ComponentePintable.trasladarDesde	5,6%	213,779	242+0	0,883
15 java/util/Vector.get (I)Ljava/lang/Object;	5,2%	201,008	22042+0	0,009

Figura 4.18: Resultados del test de eficiencia llevado a cabo por la herramienta Traceview del SDK de Android. Es un test posterior a la mejora que evita procesar el pintado de los elementos que no aparecerán en el área representada por cada uno de los *render*. Podemos ver las 15 operaciones que han estado un mayor porcentaje de tiempo ejecutándose en la CPU, sumando todas sus llamadas.

En rojo podemos ver las operaciones más importantes que se ejecutan desde el *thread* de renderizado. Además, la primera operación, *(toplevel)*, es una operación global que representa la ejecución de todo el test. En este caso este test duró unos 22 segundos.

Minimizando el número de llamadas a *glDrawArrays()*

A estas alturas, seguía pareciendo extraño que el *Render Thread* tardara tanto en crear un *render*. Así que, buscando buenas prácticas a la hora de utilizar la librería OpenGL ES, se encontró el motivo por el cual el proceso de generación de un *render* estaba siendo tan lento.

Y es que cuando trabajamos con OpenGL, es mucho más eficiente pintar varias figuras geométricas con una única llamada a *glDrawArrays()* que efectuar una llamada individual de esta operación para cada figura geométrica.

En nuestro caso, tratando cada tile del escenario como un objeto individual, se estaba cometiendo un abuso de la operación *glDrawArrays()*. Así que esta mejora de eficiencia consistió en unificar las llamadas para cada una de las baldosas de la matriz que representa el escenario, en una sola llamada.

Al principio, para no mantener dos buffers enorme en la memoria principal durante toda la reproducción del nivel, se pensó en crear dos *FloatBuffers* grandes en cada *frame*, uno con las coordenadas de la geometría y otro con las coordenadas de textura de esta geometría.

Estos *FloatBuffers* iban a contener la información de todas las baldosas que aparecieran en el *render* de dicho *frame*, para así poder llamar una sola vez a *glDrawArrays()*. El caso es que debido a la ineficiencia con la que *FloatBuffer* se ha implementado en Android, es contraproducente y llega a consumir más tiempo de CPU el crear este buffer cada *frame* que la solución de llamar múltiples veces a *glDrawArrays()*.

Llegados a este punto se analizó el número de megabytes de memoria principal que la aplicación ocupaba durante la ejecución de un nivel y comparando este número con otros videojuegos se llego a la conclusión de que estábamos gastando poca memoria. Pues mientras que, según los datos presentados por el gestor de tareas que incluye el propio S.O. Android, “Terrestrial Blast!” consumía unos 16 MB, el resto de videojuegos analizados consumían un mínimo de 21MB.

Haciendo cálculos, mantener todas las baldosas de la matriz que representa al escenario dentro de dos *FloatBuffers* en memoria principal durante toda la ejecución del nivel no suponía un gasto adicional importante de memoria principal. Así que se optó por implementar un mecanismo que se encarga de construir estos dos *FloatBuffer*, por tal de poder enviar estos buffers directamente a OpenGL ES a la hora de pintar un *render*. Esta mejora supone ahorrarnos muchísimo tiempo de ejecución de CPU en cada *frame*, pues los datos ya están preparados para el pintado.

Aunque los buffers ahora sean más grandes que en la solución anterior, pues anteriormente solo introducíamos las baldosas que eran visibles en un *render* determinado, esto no es un problema porque los *FloatBuffers* se construyen durante la carga del nivel, motivo por el cual no puede tener solo las baldosas visibles en un *render*. Durante la carga del nivel el tiempo adicional invertido en construir los *FloatBuffers* no es importante, ya que el nivel aún no se ha comenzado a reproducir y, por tanto, el usuario casi ni lo nota.

De todas formas no se ha optado por reducir las llamadas de *glDrawArrays()* a una solo durante el pintado de la matriz. En lugar de esto hemos construido una estructura que, a cambio de unas pocas llamadas a la operación de pintado, nos permite pintar tan solo las baldosas visibles en un *frame* concreto del nivel. En concreto, tenemos que llamar a *glDrawArrays()* para cada fila visible de la matriz, indicándole el rango de los *FloatBuffers* introducidos de donde puede sacar la fila de baldosas en cuestión.

Este método tiene varias ventajas. La primera ventaja radica en que, sea cual sea el tamaño del escenario, siempre se tardará lo mismo en pintarlo, pues el área del escenario visible siempre tiene una altura de 480 pixeles, como se explica en el apartados 7 de este mismo capítulo. Eso sí, el tamaño de las baldosas sí que influye en el tiempo de pintado, pero lo normal es que este tamaño sea el mismo en todos los escenarios del nivel.

En los escenarios implementados el tamaño de cada baldosa es de 64 pixeles, lo que significa que la operación *glDrawArrays()* será llamada unas 8 veces por *frame*, una por cada fila visible. Como vemos es una cantidad de llamadas ínfima, sobre todo si la comparamos con las llamadas efectuadas antes de esta mejora, que eran un mínimo de 50 (una por cada baldosa visible), asumiendo el caso de que la pantalla sea cuadrada, y la ventana tuviese 480x480 pixeles, ya que en caso contrario serían más.

La segunda ventaja radica en que nos ahorramos pintar las baldosas no visibles, las cuales normalmente son más numerosas que las visibles y esto compensa el tener que hacer algunas llamadas más a *glDrawArray()*.

Si tenemos en cuenta un caso extremo en que la pantalla tenga el doble de anchura que altura, con unas baldosas de 64x64 pixeles como las empleadas en el videojuego, estaríamos pintando unas 128 baldosas por *frame*. En cambio, si cogemos el escenario más pequeño de todos los que tiene “Terrestrial Blast!”, este está compuesto de 216 baldosas. Si, por el contrario, tomamos como referencia un escenario de los de tamaño medio, este se compone de 336 baldosas.

Y, por último, otra ventaja radica en que también nos estamos ahorrando las operaciones que se llevaban a cabo para preparar el pintado de cada tile, pues ahora solo hay que preparar cada fila y la preparación de esta es muy simple y eficiente. Antes, en cambio, para cada tile teníamos que modificar parcialmente un *FloatBuffer* con sus coordenadas, lo que consumía bastante tiempo de CPU.

Claro está que esta mejora se aprovecha de que las matrices de tile de este videojuego son estáticas: ni se mueven, ni su textura cambia. Por tanto, los *FloatBuffers* se crean una vez y no se modifican más. Si en cambio, lleváramos a cabo modificaciones sobre estos buffers la mejora de eficiencia no sería tan grande.

En resumen, gracias a esta solución se reduce drásticamente el tiempo de pintado de la matriz de tiles, de forma que *onDrawFrame()* pasa a ejecutarse en solo unos 7 milisegundos, cuando antes de esta mejora tardaba 31 milisegundos. Por tanto, se ha conseguido finalmente que la creación del *render* tarde un tiempo coherente. De hecho, como podemos ver en la Figura 4.19, todas las operaciones que tienen que ver con el *renderizado* han dejado de ser las que tienen un porcentaje mayor sobre el tiempo de ejecución de la aplicación, dando paso a otras operaciones del *thread* de juego. Si nos fijamos en *MatrizDeTiles.pintarComponente()* que es la operación donde se ha llevado a cabo esta última mejora, tan solo tarda 1,7 milisegundos en ejecutarse, frente a los 14,6 que tardaba antes.

Nombre de la operación	% CPU Total	T. Total	Llamad.+Recurs.	T. Llamada
0 (toplevel)	100,0%	3630,036	22+0	165,002
1 com/juego/terrestrial/actNivel/ObjetoComplejo.procesarFrame (Ljava/util/Vector;Lcom/	33,7%	1222,991	3313+0	0,369
2 com/juego/terrestrial/actNivel/ObjetoComplejo.aplicarComportamiento (Ljava/util/Queu	26,9%	977,042	3335+0	0,293
3 com/juego/terrestrial/actNivel/RendererPersonalizado.onDrawFrame (Ljavax/microediti	16,7%	606,201	91+0	6,662
4 com/juego/terrestrial/actNivel/ComponentePintable.procesarFrame (IIILjava/util/Vector	13,1%	476,843	1206+0	0,395
5 com/juego/terrestrial/actNivel/RendererPersonalizado.pintarComponentePintable (Ljav	12,8%	463,008	1865+0	0,248
6 com/juego/terrestrial/actNivel/ComponenteComportamiento.aplicarComportamiento (Lj	9,3%	336,070	943+0	0,356
7 com/juego/terrestrial/actNivel/ComponenteCuerpoFisico.aplicarEstadosFisicos (Ljava/ul	8,7%	314,253	3143+0	0,100
8 java/util/Vector.<init> ()V	7,9%	288,110	9733+0	0,030
9 java/util/Vector.get (I)Ljava/lang/Object;	7,6%	275,587	29909+0	0,009
10 java/nio/FloatToByteBufferAdapter.put (IF)Ljava/nio/FloatBuffer;	7,6%	274,956	5432+0	0,051
11 java/nio/ReadWriteDirectByteBuffer.putFloat (IF)Ljava/nio/ByteBuffer;	6,8%	246,165	5592+0	0,044
12 java/util/Vector.<init> (II)V	6,5%	234,307	9741+0	0,024
13 com/juego/terrestrial/actNivel/ComponentePintable.trasladarDesdelCentro (III)V	6,4%	233,437	1206+0	0,194
14 com/juego/terrestrial/actNivel/GestorDeSprites.procesarFrame (Ljava/util/Vector;)V	6,3%	227,974	822+0	0,277
15 com/juego/terrestrial/actNivel/ComponentePintable.trasladarGeometria (II)V	6,3%	227,179	520+0	0,437

Figura 4.19: Resultados del test de eficiencia llevado a cabo por la herramienta Traceview del SDK de Android. Es un test posterior a la mejora que reduce el número de llamadas a *glDrawArrays()* a la hora de pintar el fragmento de escenario del nivel que aparece en un *render*. Podemos ver las 15 operaciones que han estado un mayor porcentaje de tiempo ejecutándose en la CPU, sumando todas sus llamadas.

En rojo podemos ver las operaciones más importantes que se ejecutan desde el *thread* de *renderizado* (en este caso ya solo 1 figura entre las 15 operaciones que más tiempo se encuentran en ejecución). Además, la primera operación, *(toplevel)*, es una operación global que representa la ejecución de todo el test. En este caso este test duró aproximadamente 1 minuto 20 segundos.

Ahora, por tanto, un *frame* del videojuego tarda en ejecutarse: 8 milisegundos del *thread* de juego más 7 milisegundos del *thread* de renderizado. Lo que hace un total de 15 milisegundos, es decir, en media un *frame* se está procesando el doble de rápido de lo necesario. Al haber llevado a cabo la sincronización por *framerate*, una vez procesado cada *frame* los *threads* se pausan hasta que transcurran los 33,3 milisegundos que este debe durar, por tal de que los elementos se muevan a una velocidad correcta.

Por último, en cuanto al consumo de memoria principal este, en un nivel de tamaño medio, ha aumentado a 18 MB desde los 16 MB consumidos antes de llevar a cabo la mejora. Las conclusiones son que no ha aumentado demasiado y sigue siendo inferior al consumo que tienen otros videojuegos examinados.

Dando soporte para el uso de VBOs

A pesar de que la eficiencia del juego es más de la necesaria para alcanzar los 30 *frames* por segundo en móviles inteligentes de gama media, meta que se había marcado como objetivo, como mejora adicional se añadió soporte a los llamados VBOs.

VBO es el acrónimo de Vertex Buffer Object. Se trata de una extensión de OpenGL ES, que viene junto con OpenGL ES 1.1 y que, por tanto, no tienen por qué tener todos los dispositivos a los que "Terrestrial Blast!" va dirigido. Por este motivo, se ha implementado un selector, que en caso de que el dispositivo no soporte la extensión, lleva a cabo la construcción del *render* de forma normal.

Esta extensión nos permite almacenar los *FloatBuffers* utilizados para definir la geometría y las texturas de todos los objetos complejos, que se pintarán en los diferentes *renders*, directamente en la memoria de video.

Y es que el funcionamiento habitual de OpenGL consiste en que cuando le entregamos los buffers de donde sacar la información acerca de las geometrías a pintar, a través de la función *glDrawArrays()*, este copie esos buffers en la memoria de video, la cual está cerca de la GPU. De esta forma se consigue mayor eficiencia, pues a la hora de generar el *render*, el acceso a la información necesaria para realizar las funciones pertinentes en la GPU es más rápido.

El problema si no se utiliza esta extensión radica en que si tenemos unos buffers que representan a una geometría estática, la cual no cambia nunca o no cambia en cada *frame*, enviar estos buffers todos los *frames* desde la memoria principal hasta la memoria de video es un trabajo innecesario, pues estamos enviando siempre la misma información o, al menos, en la mayoría de los casos.

Esta extensión soluciona este problema, pues si gestionamos estos buffers directamente sobre la memoria de video y los mantenemos ahí durante toda la ejecución, nos ahorramos el enviar la misma información una y otra vez de una memoria a otra.

Una vez implementado este añadido, analizando los resultados con el Traceview, se vio que, en el dispositivo utilizado para hacer las pruebas, no hay diferencia entre utilizar VBOs o no. La supuesta mejora realmente no fue tal y el juego tiene prácticamente el mismo rendimiento.

Nótese que se aplicaron VBOs sobre todos los objetos complejos, no solo sobre la matriz del escenario, pues esta extensión puede llegar a aportar alguna mejora de eficiencia en todos los casos.

Se barajan dos motivos por los cuales el uso de VBOs no ha ofrecido mejora alguna en el rendimiento del videojuego. En primer lugar, el dispositivo utilizado para las pruebas no tiene memoria de video, pues la misma memoria interna se utiliza como memoria principal y memoria de video; puede ser que los drivers estén muy bien optimizados para que los buffers no se tengan que copiar en otro lugar de la misma memoria y, por tanto, no es necesaria esta extensión. En segundo lugar, puede que sea al contrario y estos drivers no estén bien implementados, haciendo que los VBOs no ofrezcan mejora de eficiencia alguna cuando deberían ofrecerla.

En todo caso a estas alturas no es preocupante pues como se ha visto en el apartado anterior, el videojuego tiene un rendimiento que cumple de sobras con los objetivos marcados en este ámbito. Además, la extensión puede que sí suponga una mejora de eficiencia importante en otros dispositivos móviles diferentes.

Por último, hay algunas consideraciones generales que deben quedar claras:

En primer lugar, las pruebas realizadas para extraer el rendimiento de la aplicación y poder compararlo después con otras extracciones, tal y como hemos hecho en los últimos apartados y figuras, se han realizado durante la ejecución de un nivel de tamaño grande del videojuego.

Las primeras pruebas se han realizado tomando una medición de aproximadamente 20 segundo, tiempos suficiente para ver que el rendimiento no era óptimo, mientras que las dos últimas (Figura 3.19 y tras VBOs) han tenido una duración de algo más de un minuto, por tal de asegurarnos de que el rendimiento sea óptimo tras jugar algo más de tiempo. En cuanto al nivel del videojuego escogido, como decíamos, ha sido en esencia el mismo para todas las pruebas, salvo en las dos últimas, donde nuevamente se ha sida más exigente y se ha probado un nivel con más elementos dinámicos. Eso sí, en todas las pruebas se han llevado a cabo acciones intensas de disparo y de movimiento, para poner a prueba la eficiencia del videojuego.

En cuanto al dispositivo móvil utilizado para realizar las pruebas de eficiencia, todas se han hecho con el mismo dispositivo, el cual es de gama media. Se llama Motorola Defy, tiene una CPU *TI OMAP3630-800* que trabaja a 800 MHz y una memoria principal (o memoria RAM) de 512MB, también utilizada como memoria de video. Además, la resolución a la que presenta la información al usuario por pantalla es de 854x480 pixeles y, para conseguirla, monta una GPU PowerVR SGX 530, la cual destaca por su buen rendimiento.

Capítulo 5 – Planificación y Coste Económico

En este capítulo se llevará a cabo la comparación entre las horas planificadas para cada tarea y las horas que se han invertido realmente, justificando debidamente el aumento o disminución de horas. Además, a partir de las horas reales calcularemos los costes que habría tenido el proyecto si realmente hubiese sido desarrollado por un equipo de personas en una empresa.

1. Planificación del proyecto

La planificación no es más que un instrumento de trabajo a través del cual podemos acotar el tiempo de trabajo que llevará un proyecto y avistar a tiempo posibles desviaciones en la planificación, por tal de evitar que la duración del proyecto sea mayor de la que teníamos prevista o que, en caso de alargarse, se alargue lo mínimo posible.

Por tanto, la planificación va cambiando a medida que avanza el proyecto, ajustándose a la realidad que se va dando y detallándose cada vez más. A continuación se presentan las tareas y sub-tareas en las que se dividió la totalidad del proyecto, con las horas planificadas para estas tareas en el momento de antes de iniciarlas (ver Figura 5.1).

De forma que, por ejemplo, la planificación que encontramos sobre el desarrollo del segundo videojuego, es la ajustada después de llevar a cabo las tareas previas, pues la planificación para este videojuego anterior estaba hecha muy a groso modo y era imprecisa, pues aún no se tenían los conocimientos necesarios para hacerla más ajustada. Una vez estudiado el S.O. Android y realizado el primer videojuego, fue más fácil planificar el segundo.

Nombre de tarea	Duración
Estudio y Análisis de Android.	**146 horas**
Estudio S.O. Android.	**77 horas**
Estudio de Android y de su funcionamiento.	22 horas
Estudio de la estructura y funcionamiento interno de una Aplicación.	39 horas
Estudio del entorno y herramientas de desarrollo.	16 horas
Estudio de la Estructura de un videojuego y su aplicación en Android.	10 horas
Análisis.	**11 horas**
Análisis de Antecedentes.	5 horas
Análisis y selección de las herramientas para desarrollar un videojuego.	6 horas
Estudio de las librerías.	**48 horas**
Estudio de la librería gráfica OpenGL ES 1.X.	28 horas
Estudio del motor de físicas Box2D.	20 horas
Primer videojuego: PingPangPung.	**50 horas**
Definición, Especificación y Diseño.	10 horas
Implementación, prueba y corrección de errores.	40 horas
Segundo videojuego: "Terrestrial Blast!".	**190 horas**
Definicion.	12 horas
Especificación.	10 horas
Diseño.	45 horas
Implementación.	75 horas
Diseño Artístico.	6 horas
Prueba y correción de errores.	42 horas
Documentación (Memoria, Informe Previo).	65 horas
HORAS TOTALES INVERTIDAS:	451 horas

Figura 5.1: Planificación del proyecto. Se muestran las tareas y las sub-tareas en las que se ha dividido el proyecto y las horas dedicadas a cada una de ellas.

Como vemos, el proyecto se puede dividir en tres fases. La primera de ellas consiste en un estudio y análisis de la plataforma Android. Destacan el análisis de antecedentes y de herramientas para el desarrollo de videojuegos. Además, esto se acompaña de un estudio de la arquitectura de un videojuego y una posterior adaptación de esta arquitectura a Android. Así como del estudio de las librerías empleadas para el desarrollo del videojuego.

En la segunda fase, se llevó a cabo el desarrollo del videojuego más sencillo "Ping Pang Pung". Y la tercera fase está compuesta por el desarrollo de "Terrestrial Blast!" un videojuego mucho más complejo y, por tanto, con bastante más tiempo de desarrollo.

En concreto entre las sub-tareas de este segundo desarrollo entendemos como "Definición" a la tarea de definir el concepto del videojuego, pero también se incluye en ella la definición de los menús que tiene el videojuego, así como la definición y diseño de los niveles (incluyendo escenarios, enemigos, etc.). Por tanto, se incluye dentro en esta tarea toda la programación en XML llevada a cabo en el videojuego (para más información leer el apartado 5 del capítulo 4).

La "Especificación" en cambio ser refiere, como su nombre indica a la tarea de definir de forma más técnica y precisa las funciones que va a tener el videojuego, una vez hecha la definición inicial de este. Seguidamente en la fase de "Diseño" se incluye el diseño de la arquitectura del videojuego, es decir, cómo se van a implementar las funcionalidades definidas durante la "Especificación".

"Implementación" incluye la tarea de diseñar los algoritmos concretos con los cuales se van a responder las peticiones que se hagan a una clase determinada y, cómo no, también incluye la implementación de estas. Nótese que normalmente el diseño de los algoritmos se hace en la fase de "Diseño" pero, en este caso, debido al poco tiempo de desarrollo disponible se ha juntado este diseño de operaciones con la implementación, en una misma tarea.

En cuanto a la tarea de "Diseño artístico", incluye la búsqueda y modificación de los recursos gráficos y auditivos empleados en la construcción del videojuego. Así como la creación desde cero de algunos recursos que no parten de otros ya existente. Por último, tenemos la tarea de "Prueba y corrección de errores", que como su nombre indica ha incluido la prueba de la aplicación y, en caso de un funcionamiento no deseado, la corrección del error.

Después de la fase de construcción del segundo videojuego, se encuentra la fase de documentación. Esta incluye la redacción de la memoria y otras tareas menos extensas, como la creación del Informe Previo.

El orden preciso de ejecución de las diferentes tareas no se muestra, pues se detalla en el apartado siguiente, junto con las horas reales invertidas. El motivo es que se ha seguido el mismo orden planificado sin ningún cambio al respecto.

2. Tiempo real invertido

Una vez vista la planificación del proyecto, vamos a compararla con las horas reales invertidas para realizar dicho proyecto. En este apartado, además de las horas dedicadas a cada tarea y sub-tarea, se presenta el diagrama de Gantt completo, con el orden en que se han llevado a cabo estas tareas (ver Figura 5.2).

Orden que, como ya se ha indicado, el orden ha sido el mismo que se había planificado, pues al ser solo una persona la dedicada al proyecto y siendo las diferentes tareas fuertemente dependientes unas de otras, no habían muchas alternativas posibles.

2.1. Diferencias entre la planificación y las horas reales

En este apartado vamos a señalar y explicar las diferencias principales entre la planificación del proyecto (ver Figura 5.1) y las horas reales invertidas en este (ver Figura 5.2).

Dentro de la fase de Estudio y Análisis de Android, se destaca la tarea de estudio de la arquitectura de un videojuego y la adaptación a Android. Y es que esta tarea se llevó a cabo en tiempo record gracias a que se tenía ya una idea de la estructura genérica de un videojuego y gracias a una conferencia de la feria Google IO 2010 donde Chris Pruett explica la adaptación de esta estructura a Android [23]. De forma que el número de horas dedicadas a la tarea se redujeron a la mitad.

Por otro lado, el estudio de OpenGL ES 1.X se alargó un días más, pues las múltiples versiones existentes de esta librería de gráficos unido al hecho de que la versión ES de esta tiene algunas restricciones importantes y fue difícil encontrar información específica dificultaron la tarea. Lo mismo ocurrió con el estudio de Box2D, el cual también se alargó casi un día más. En este caso debido a que la librería de físicas se utiliza a través de LibGDX y la inclusión de esta en el entorno, unido al estudio de las pequeñas diferencias entre los métodos de Box2D y los métodos utilizados a través de LibGDX, alargaron la tarea.

En lo que respecta al desarrollo del primer videojuego, todo salió como se había previsto y no hubo contratiempos. Cosa que no podemos decir de "Terrestrial Blast!" pues su desarrollo se alargó, principalmente porque al final se quiso realizar un videojuego con unos niveles y un diseño artístico, tanto visual como sonoro, bastante más trabajados de lo que inicialmente se había planteado; esto llevó a que las tares de definición y diseño artístico se alargaran.

También hubo imprevistos como, por ejemplo, que la librería Box2D utilizada a través de LibGDX no nos permitiera hacer uso del modo de *debug* de físicas, factor que implicó el desarrollo de dicho modo de *debug* como una parte más del desarrollo del videojuego. Este imprevisto, junto con otros, así como el hecho de que planificar un proyecto más largo es más difícil, provocó que las tareas de implementación, prueba y corrección de errores se alargaran. En total la etapa de desarrollo del videojuego "Terrestrial Blast!" duró unas 22 horas más.

Por último, la etapa de documentación, más o menos mantiene el número de horas planificado. Eso sí, con su debida desviación, ya que siendo una tarea tan extensa es difícil ser preciso.

En resumen, el proyecto, en su totalidad, se ha llevado a cabo en unas 477 horas, unas 26 horas más de las previstas.

2.2. El orden y las dependencias entre tareas

A la hora de hacer el diagrama de Gantt se ha considerado una jornada de trabajo de 4 horas diarias de lunes a viernes, sin contar días festivos. De esta forma tenemos una planificación más genérica, dividida en semanas que podrían ser de cualquier mes, pues todas son iguales. Eso sí, es evidente que a la hora de realizar el proyecto, el periodo de tiempo se alarga debido a los días festivos o contratiempos que puedan surgir (contratiempos que atrasan el proyecto pero no cuentan como horas de este).

En concreto, la unidad de tiempo presentada en el Gantt es el conjunto de 3 de días. Y, adicionalmente, se indica a qué mes pertenece cada triplete. Es decir, la casilla con un "02" del mes de marzo de 2011, engloba las tareas realizadas el día 2, 3 y 4 de dicho mes.

En el diagrama se puede observar como las tareas de la primera etapa "Estudio y Análisis de Android" se han llevado a cabo secuencialmente, ya que por lo general era necesario tener los conocimientos previos que te aportan las primeras tareas antes de hacer las tareas posteriores. La etapa se lleva a cabo toda seguida, salvo el estudio de la librería Box2D, que al ser necesario solo para el segundo videojuego, se llevo a cabo posteriormente al desarrollo de "Ping Pang Pung".

Cabe destacar que la realización de la documentación se ha dividido en dos subtareas. Pues el Informe Previo junto con la parte de la memoria que describe al primer videojuego se hicieron nada más acabar dicha aplicación y antes siquiera de estudiar la librería Box2D. En cambio, el resto de la memoria se ha llevado a cabo en la última parte del proyecto, una vez finalizado el segundo videojuego.

En cuanto a la realización del segundo proyecto, existe una dependencia clara, que es la que lleva la etapa de "Diseño" respecto a la de "Especificación", la cual debe acabarse antes de poder diseñar la arquitectura de la aplicació. Pero, en cambio, el resto de etapas se han llevado más o menos simultaneamente.

Una parte de la definición del proyecto se llevó antes de empezar siquiera a Especificar, pues era necesario tener claro qué aplicación ibamos a desarrollar, pero también se va perfilando durante todo el desarrollo, hasta llegar a la etapa final, donde más peso tiene pues es cuando se definen los niveles, enemigos, etc. Si nos fijamos en el "Diseño" este en gran parte se hace antes de empezar a implementar, pero cuando empieza la implementación tambien se van perfilando algunos aspectos más concretos de este.

En cuanto a las pruebas y a la corrección de errores, ambos empiezan poco después de comenzar a implementar, pues es necesario ir probando periodicamente las funciones que se estan desarrollando para correguir los errores contra antes mejor. Por último, el diseño artístico empieza cuando el desarrollo esta ya bastante avanzado ya que es cuando se comienzan a implementar los menús y a definir los diferentes niveles con sus respectivos elementos (enemigos, bloques de piedra, etc.).

Figura 5.2: Diagrama de Gantt con las tareas realizadas a lo largo del proyecto, el número de horas reales dedicadas a cada tarea y el orden de realización de dichas tareas.

Nombre de tarea	Duración
Estudio y Análisis de Android.	**148 horas**
Estudio S.O. Android	**77 horas**
Estudio de Android y de su funcionamiento.	22 horas
Estudio de la estructura y funcionamiento interno de una Aplicación.	39 horas
Estudio del entorno y herramientas de desarrollo.	16 horas
Estudio de la Estructura de un videojuego y su aplicación en Android.	5 horas
Análisis.	**11 horas**
Análisis de Antecedentes.	5 horas
Análisis y selección de las herramientas para desarrollar un videojuego.	6 horas
Estudio de las librerías.	**55 horas**
Estudio de la librería gráfica OpenGL ES 1.X.	32 horas
Estudio de la librería de físicas Box2D.	23 horas
Primer videojuego: PingPangPung.	**49 horas**
Definición, Especificación y Diseño.	8 horas
Implementación, prueba y corrección de errores.	41 horas
Segundo videojuego: "Terrestrial Blast!".	**212 horas**
Definición.	16,5 horas
Especificación.	11 horas
Diseño.	45,5 horas
Implementación.	83 horas
Diseño Artístico.	10 horas
Prueba y corrección de errores.	46 horas
Documentación (Planificación, Memoria, Informe Previo).	**68 horas**
Memoria (trozo del primer videojuego) e Informe Previo.	16 horas
Resto de la memoria.	52 horas
HORAS TOTALES INVERTIDAS:	477 horas

marzo 2011
27 02 05 08 11 14 17 20 23 26 29
abril 2011
01 04 07 10 13 16 19 22 25 28
mayo 2011
01 04 07 10 13 16 19 22 25 28 31
junio 2011
03 06 09 12 15 18 21 24 27 30
julio 2011
03 06 09 12 15 18 21 24 27 30
agosto 2011
02 05 08 11 14

3. Coste Económico del proyecto

En este apartado se detalla el coste económico final del proyecto, desglosando los diferentes gastos que nos llevan a la cantidad final. Durante el proyecto ha sido una sola persona la que ha llevado a cabo todas las tareas, pero estas tareas se enmarcan en diferentes roles con diferente reconocimiento cada uno.

En primer lugar, se destaca el rol de director del proyecto, el cual se encarga de gestionar la evolución del proyecto mediante la planificación de este. Al mismo tiempo se encarga de unir y revisar la documentación final. Seguidamente tenemos al analista, encargado de llevar a cabo el análisis de antecedentes y definir los diferentes videojuegos, construyendo, en última instancia, la especificación de los dos productos.

Continuamos con el diseñador, que es el que estudia los entresijos del S.O. Android para poder definir la estructura completa de los dos productos a construir, teniendo en cuenta las propiedades de extensibilidad, portabilidad y reusabilidad, unidas a la propiedad de eficiencia. Este rol ha tenido el papel principal en el desarrollo de "Terrestrial Blast!".

También es necesario un diseñador artístico, que se encargue de los recursos gráficos y sonoros del videojuego. Aunque este no trabaja demasiadas horas, pues el proyecto está más orientado al perfil de ingeniero informático.

Por último, se encuentra el programador, el cual a través del diseño construye las diferentes clases de que se componen los dos videojuegos y lleva a cabo las pruebas para asegurarse de que funcionen correctamente. Este también necesita conocer el S.O. Android, para poder programar utilizando su API.

Cabe decir que aunque algunas tareas las deben llevar a cabo dos roles al mismo tiempo, como este proyecto está hecho por una única persona, no se contarán esas horas por duplicado. Pues al fin y al cabo, no es tan raro que una aplicación para Android sea desarrollada enteramente por una sola persona.

En la Figura 5.3 podemos ver el coste total de cada una de las tareas llevadas a cabo en el proyecto, junto con el rol que la ha llevado a cabo y el precio por hora de este rol.

Nombre de la tarea	Nombre	Rol ejecutor	Coste/ Hora	Coste total
Estudio y Análisis de Android.	**148 horas**	-	-	**2.605 €**
Estudio S.O. Android	**77 horas**	-	-	**1.460 €**
Estudio de Android y de su funcionamiento.	22 horas	Diseñador	20 €	440 €
Estudio de la estructura y funcionamiento interno de una Aplicación.	39 horas	Diseñador	20 €	780 €
Estudio del entorno y herramientas de desarrollo.	16 horas	Programador	15 €	240 €
Estudio de la Estructura de un videojuego y su aplicación en Android.	5 horas	Diseñador	20 €	100 €
Análisis.	**11 horas**	-	-	**220 €**
Análisis de Antecedentes.	5 horas	Analista	20 €	100 €
Análisis y selección de las herramientas para desarrollar un videojuego.	6 horas	Diseñador	20 €	120 €
Estudio de las librerías.	**55 horas**	-	-	**825 €**
Estudio de la librería gráfica OpenGL ES 1.X.	32 horas	Programador	15 €	480 €
Estudio de la librería de físicas Box2D.	23 horas	Programador	15 €	345 €
Primer videojuego: “Ping Pang Pung”.	**49 horas**	-	-	**775 €**
Definición, Especificación y Diseño.	8 horas	Analista y Diseñador	20 €	160 €
Implementación, prueba y corrección de errores.	41 horas	Programador	15 €	615 €
Segundo videojuego: "Terrestrial Blast!".	**212 horas**	-	-	**3545 €**
Definición.	16,5 horas	Analista	20 €	330 €
Especificación.	11 horas	Analista	20 €	220 €
Diseño.	45,5 horas	Diseñador	20 €	910 €
Implementación.	83 horas	Programador	15 €	1.245 €
Diseño Artístico.	10 horas	Diseñ. Artístico	15 €	150 €
Prueba y corrección de errores.	46 horas	Programador	15 €	690 €
Documentación (Memoria, Informe Previo).	68 horas	Director	25 €	1.700 €
Coste total de las recursos humanos	**477 horas**			**8.625 €**

Figura 5.3: Tabla con los costes desglosados de los recursos humanos del proyecto. El coste por hora es el coste en bruto.

Seguidamente se adjunta una tabla (ver Figura 5.4) con el coste de los materiales empleados durante la realización del proyecto. Cabe decir que el software que conforma el entorno de desarrollo de Android es totalmente gratuito. Sin embargo, el emulador de Android para PC funciona realmente mal, siendo imprescindible adquirir, al menos, un terminal para desarrollar la aplicación.

En cuanto al resto del software, la licencia de Windows XP ha sido la que la licencia de que dispone UPC y no se ha contado como coste adicional, pues en caso de no disponer de ella se habría optado por utilizar una variante gratuita de Linux. Y lo mismo ocurre con el resto de aplicaciones utilizadas, como son el procesador de textos, el editor de imágenes y la herramienta para crear la especificación de las aplicaciones. Pues se ha optado por utilizar aplicaciones de pago si se tenía la licencia o aplicaciones gratuitas si no se tenía esta, ya que siempre hay una alternativa gratuita.

Otro coste muy importante es la conexión a Internet, pues toda la documentación oficial acerca de Android y acerca del desarrollo de aplicaciones para la plataforma se encuentra en la nube. Además de innumerables ejemplos y información de gran valor. Por tanto, aunque nos ayudemos también de libros, es imprescindible contar con conexión a la red.

Recurso material	Coste por mes	Número de meses	Coste total
Smartphone Android	-	-	400 €
Ordenador (PC)	-	-	600 €
Conexión a Internet	50 €	6	300 €
		Coste total	**1.300 €**

Figura 5.4: Tabla con los costes materiales del proyecto desglosados.

Concluimos, que el coste total de desarrollar el proyecto, sumando el coste de los recursos humanos y el coste de los materiales, es de **9.925€**.

Capítulo 6 - Conclusiones del proyecto

1. Conclusiones generales

Llegados a este punto de la memoria, a continuación se redactan las conclusiones más importantes extraídas de la realización del proyecto. Los objetivos del proyecto se han cumplido al detalle y a través de ellos podemos extraer conclusiones muy valiosas.

En primer lugar, fruto del estudio y análisis del S.O. Android, podemos concluir que:

- El desarrollo y comercialización de aplicaciones en Android (sean videojuegos o no) es relativamente fácil y barato. Pues el carácter abierto del S.O. junto con las facilidades que Google da al desarrollador facilitan mucho la tarea.

- Aun así, las herramientas de desarrollo están lejos de ser perfectas. Pues durante el desarrollo de los videojuegos se han detectado fallos. Algunos de estos fallos después han sido corregidos en futuras actualizaciones del entorno pero otros aún no tienen solución.

  En concreto, el emulador de Android para PC por el momento ofrece un rendimiento muy malo y es imposible desarrollar un videojuego complejo haciendo las pruebas a través de este emulador. Si no tenemos un dispositivo con ecosistema Android donde ir haciendo las pruebas, no podremos llevar a cabo el desarrollo con éxito.

❖ Por último, la fragmentación entre diferentes dispositivos con diferentes versiones de Android es un problema. Pues si queremos que nuestra aplicación llegue a más de la mitad de usuario de Android no es viable lanzar una aplicación que utilice una versión concreta del S.O. hasta, al menos, un año después de su salida. Y aun así estaremos llegando a poco más de la mitad de usuario.

  Para demostrarlo solo tenemos que observar el porcentaje de usuarios con Android que en Enero de 2012 tienen la versión 2.3 de Android o una de sus variantes, versión que salió en Diciembre del 2010. El porcentaje es de tan solo el 55.5% [18].

  Esto provoca que mejoras en la API de Android no puedan ser utilizadas por el desarrollador hasta pasado un buen tiempo.

Por tanto, Android es un ecosistema que supone una buena plataforma para el desarrollador a día de hoy, pues ya tiene cierta madurez. Pero aún quedan algunos puntos que mejorar, los cuales enriquecerán aún más la experiencia del desarrollador con la plataforma.

En cuanto al desarrollo de videojuegos en concreto, las conclusiones extraídas son las siguientes:

- Se pueden desarrollar juegos relativamente exigentes en cuanto a número de elementos en pantalla y magnitud de los escenarios haciendo uso únicamente del lenguaje Java y de algunas librerías adicionales.

  A priori puede parecer un problema el hecho de que el lenguaje C, en versiones anteriores a la 2.3 de Android, sea un lenguaje que únicamente podamos utilizar a través de funciones declaradas en lenguaje Java. Pues la API de Android está únicamente en Java y, por tanto, debemos trabajar con dos lenguajes si queremos hacer uso de C.

  Este problema se puede apaliar gracias a la librería LibGDX, que oculta la librería de físicas Box2D en lenguaje C, para que podamos utilizarla desde Java. De esta forma podemos utilizar una librería en C sin necesidad de utilizar más que el lenguaje Java en nuestra aplicación.

  Adicionalmente es necesario hacer un uso eficiente de la librería de gráficos OpenGL ES, como el que se ha detallado en el apartado 8 del capítulo 4.

- Cuando desarrollamos videojuegos para dispositivos móviles normalmente hay que priorizar soluciones eficientes en cuanto a tiempo, ante soluciones eficientes en espacio. Y es que, mientras en este tipo de dispositivos la memoria RAM suele ser extensa y es difícil de desbordar, no se puede decir lo mismo de las CPUs, pues estas son limitadas y algoritmos que hagan un uso excesivo de estas acaban afectando muy negativamente al rendimiento del videojuego.

- Otra conclusión importante, es que los videojuegos deben implementar controles adaptados a los dispositivos donde se van a jugar y se debe evitar, por ejemplo, el mapeo de botones en pantalla táctil sin una justificación en cuanto a usabilidad.

  Además los videojuegos, deberán contar una historia, con unos personajes, aunque estos se describan simplemente visualmente, mediante la reproducción de los niveles. Pues esto provocará que el videojuego tenga un mayor atractivo de cara al desarrollador.

- Por último, la conclusión más importante de todas, radica en el hecho de que es preferible desarrollar un videojuego siguiendo una estructura extensible, reutilizable y portable, tal y como se ha hecho con “Terrestrial Blast!”, que llevar a cabo un diseño basado únicamente en la eficiencia.

  Y es que, aunque el desarrollo inicial del núcleo extensible y reutilizable de la aplicación es más extenso y difícil de llevar a cabo, toda esta dificultad y extensión posteriormente se convierte en múltiples ventajas que compensan con creces las desventajas.

  En primer lugar, dividir el desarrollo en dos niveles de abstracción nos permite separar perfectamente entre: el desarrollo de los componentes que pueden componer los diferentes elementos del videojuego de forma genérica (utilizando el lenguaje Java) y la generación de estos elementos mediante la combinación de los componentes desde un nivel de abstracción mayor (utilizando el lenguaje XML).

  Esta separación nos permite evitar errores y reutilizar el código, lo que, a largo plazo, acabo ahorrándonos tiempo de desarrollo. Además, una definición extensible y reutilizable de ambos niveles, facilita enormemente la extensión de estos.

  Por tanto, sacrificando mayor tiempo en la construcción del núcleo (o motor) y tardamos más en ver los primeros resultados, nos ahorramos a cambio mucho más tiempo en la posterior creación de niveles y de elementos de estos niveles (como pueden ser enemigos, objetos con los que interactuar, etc.). De forma que incluso llevar a cabo una segunda parte del videojuego, será mucho más fácil pues podremos partir del mismo núcleo ya construido y extenderlo con mejoras de forma realmente simple.

2. Conclusiones personales

Las conclusiones personales extraídas de la realización del proyecto se pueden resumir en un punto:

- Se han adquirido los conocimientos y experiencia necesarios para poder, en un futuro, desarrollar videojuegos para Android con facilidad. Así como para cualquier otra plataforma móvil, con un período de adaptación a esta relativamente corto.

  Abriendo de esta forma el camino para poder dedicarme al desarrollo de videojuegos en un futuro.

Pero si nos paramos a detallar este punto, en concreto, se ha conseguido:

- Adquirir los conocimientos necesarios para poder desarrollar tanto aplicaciones simples, como videojuegos que muevan gráficos en tiempo real.
  - Se ha aprendido a desarrollar aplicaciones para el sistema operativo Android.
  - Se ha aprendido a utilizar la librería gráfica multiplataforma OpenGL ES 1.X.
  - Se ha aprendido a utilizar el motor de físicas multiplataforma Box 2D.
  - Se han integrado los conocimientos adquiridos durante la carrera con los nuevos conocimientos, adaptándolos a un sistema operativo diferente. Así, conceptos de Ingeniería del Software, de programación, de planificación de proyectos se han mezclado para dar lugar a los dos productos desarrollados durante la realización del proyecto.
- Se ha adquirido la capacidad de acotar un coste en tiempo y dinero para la realización de un videojuego sobre la plataforma Android a través de la experiencia adquirida con los dos productos construidos durante la realización del proyecto de final de carrera.
- Y, por último, se ha aprendido mediante la practica desarrollar un diseño para un videojuego que además de eficiente es extensible, reutilizable y cambiable. Además, de haberse tomado consciencia, con la propia practica de lo importantes que pueden llegar a ser estas propiedades en el motor de un videojuego.

3. Ampliaciones posibles del proyecto

Este proyecto no tiene porqué finalizar aquí, pues existen diferentes ramas por donde ampliarlo que se pasan a detallar a continuación. Las ramas abarcan únicamente al producto "Terrestrial Blast!", pues ha sido el producto más completo desarrollado y el que más vistas al futuro tiene.

En primer lugar, el motor, que como ya se ha dicho es fácilmente extensible en los dos niveles de abstracción que presenta, podría ser ampliado con nuevos niveles *jugables* y nuevos elementos para estos niveles del videojuego. Llegando de esta forma a construir un videojuego totalmente completo y profesional.

En segundo lugar, se podría llevar a cabo una ampliación no tanto a nivel de escenarios, enemigos, etc. sino a nivel de funcionalidades. En concreto se podría implementar un modo online, desarrollo que también se vería facilitado gracias a la estructura extensible, reutilizable y portable del videojuego.

En tercer lugar, incluso se podría crear un videojuego que difiera del actual, partiendo del motor construido, el cual brinda muchas posibilidades diferentes, y ampliándolo de la forma necesaria para conseguir representar el nuevo videojuego planteado.

Por último indicar que, para facilitar estas ampliaciones, el código del videojuego "Terrestrial Blast!" ha sido documentado con detalle. Cada clase tiene una cabecera con una explicación en lenguaje natural, que ayuda a entender qué es lo que allí se gestiona. Lo mismo ocurre con las operaciones, las cuales están dotadas de una cabecera que describe qué responsabilidades se llevan a cabo en ellas, junto con comentarios en el interior que explican cómo se están llevando a cabo dichas responsabilidades.

Mediante esta especificación detallada, no solo se pretende que el videojuego se pueda extender con facilidad, sino que además se intenta dar un soporte para el aprendizaje de otros desarrolladores que quieran llevar a cabo estructuras extensibles, reutilizables y portables en su videojuego.

Y es que, durante el desarrollo ha sido muy útil disponer de otros proyectos a través de los cuales poder mirar ejemplos de cómo implementar determinados elementos del videojuego [24]. Pero, desgraciadamente, los proyectos abiertos para su estudio son muy escasos y suelen estar poco documentados, lo que nos lleva a deambular un buen rato por las clases hasta que entendemos que se está haciendo en ellas.

Con “Terrestrial Blast!” se pretende que, gracias a la especificación detallada y a lo explicado en esta misma memoria sea fácil aprender a través de este ejemplo real de videojuego. Aportando también unos recursos para el aprendizaje en Castellano, ya que actualmente existen pocos recursos para aprender a desarrollar en Android que estén en dicho idioma.

Anexo I – Manual de Usuario de “Terrestrial Blast!”

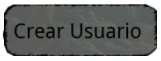
En este anexo se detalla cómo navegar por los diferentes menús de “Terrestrial Blast!” así como el manejo del personaje que controlamos en los diferentes niveles del videojuego. Después de leer esta guía el usuario sabrá todo lo necesario para poder completar el videojuego con éxito.

1. Navegando por los menús

En este primer apartado nos centramos en la navegación por los diferentes menús del videojuego.

Figura 7.1: Pantalla de título del videojuego “Terrestrial Blast!”. Es la primera pantalla que aparecerá a la hora de ejecutar el videojuego.

Cuando ejecutamos el videojuego lo primera pantalla que aparecerá es la pantalla de título del videojuego (ver Figura 7.1), con el titulo y el nombre del autor. Para poder empezar la aventura lo único necesario es tocar sobre cualquier punto de la superficie de la pantalla táctil.

La siguiente pantalla es la de selección de usuario. Aquí podemos llevar a cabo diferentes acciones. Si es la primera vez que jugamos al videojuego deberemos crear nuestro usuario, para ello presionamos sobre la etiqueta de un slot de usuario vacío 

. Esto nos llevará a la pantalla de creación de usuario, donde presionando sobre el campo de texto deberemos insertar nuestro nombre. Cuando tengamos el nombre debidamente introducido presionaremos sobre el botón “Crear Jugador”, lo que nos devolverá a la pantalla de selección de usuario esta vez con la cuenta de usuario recién creada disponible para comenzar a jugar.

En caso de querer eliminar una cuenta de usuario, únicamente deberemos presionar sobre el icono que se encuentra en la parte derecha de cada cuenta de usuario. El sistema nos pedirá que confirmemos la eliminación y, en caso de confirmar, la cuenta de usuario desaparecerá dando lugar a un slot vacio que, en cualquier momento, podrá ser utilizado para crear un nuevo usuario.

Si ya hemos decidido con qué cuenta de usuario queremos iniciar sesión, presionaremos sobre la etiqueta con el nombre de esta cuenta de usuario (por ejemplo David). Dicha acción nos llevará a la pantalla de selección de niveles. En esta pantalla podremos presionar sobre el icono del nivel concreto que queramos jugar (por ejemplo 02) para inicializar la ejecución del nivel. Pero si encontramos que una de los niveles contiene un icono donde se presenta un candado , significará que dicho nivel aún no está desbloqueado y presionar sobre este icono no provocará ninguna acción hasta que el jugador haya completado el nivel anterior a este y, por tanto, pueda jugarse.

Una vez que comienza la ejecución de un nivel, debemos completarlo siguiendo las indicaciones que aparecen en el apartado 2 de este mismo anexo. Cuando completamos el nivel nos aparece una pantalla con tres botones, presionando estos botones podremos indicar si queremos ir al menú de selección de niveles, repetir el nivel jugado o jugar el siguiente nivel. En caso de fracasar el nivel, la pantalla de fracaso nos permite hacer las mismas funciones, menos la de jugar el siguiente nivel, pues al no haber completado el nivel actual, el siguiente puede no estar desbloqueado (ver Figura 7.2).

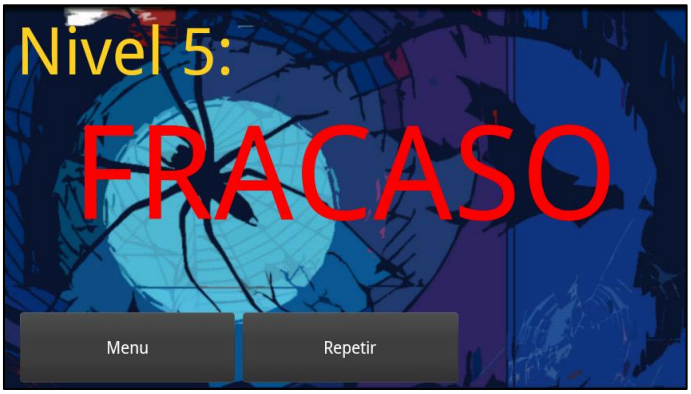


Figura 7.2: Pantalla de Fracaso. Aparecerá en el videojuego “Terrestrial Blast!” cuando el usuario no haya conseguido completar un nivel.

2. Completando los niveles

Cuando jugamos un nivel del videojuego, para controlar el movimiento del personaje principal, hacemos uso del llamado sensor de orientación, combinación de la brújula digital y el sensor de aceleración.

El funcionamiento de este sensor es el que vemos en las Figura 7.3. Si queremos que el personaje se mueva hacia arriba o hacia abajo, rotaremos el dispositivo con el que estamos jugando sobre sí mismo y sobre el eje X, eje que es paralelo al plano del suelo y al del jugador. En cambio, si queremos desplazarnos hacia los lados, rotaremos el dispositivo con el que estamos jugando sobre sí mismo y sobre la eje Z, eje que es perpendicular al plano del jugador y paralelo al plano del suelo.

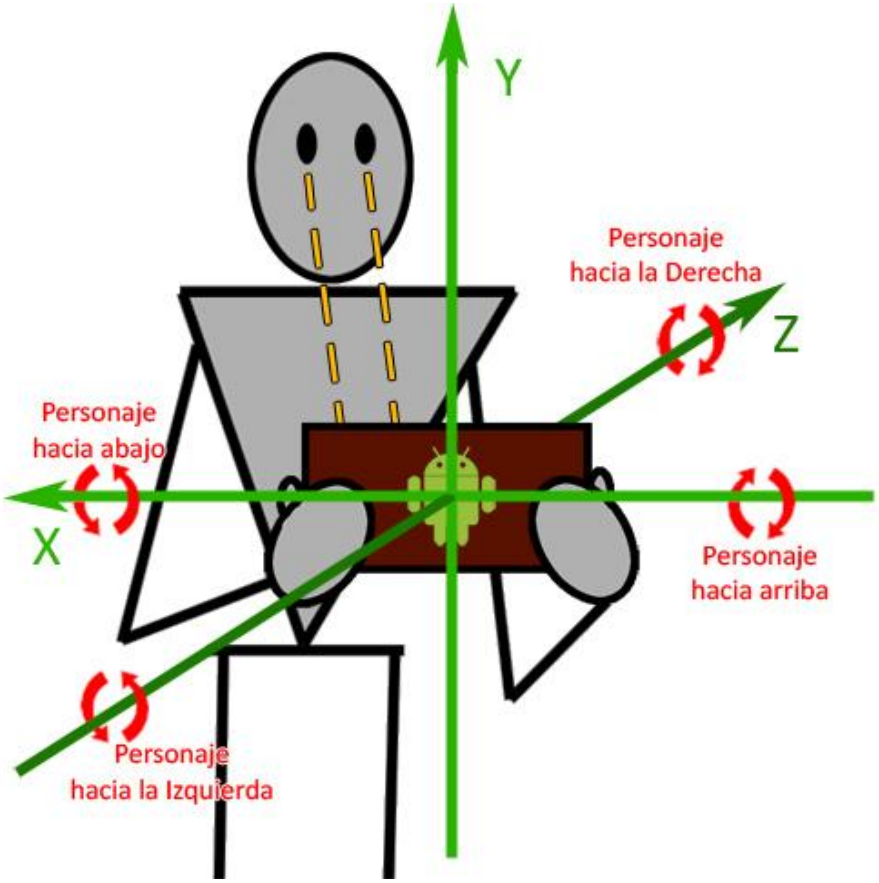


Figura 7.3: Esquema donde se describe cómo podemos controlar al personaje principal del videojuego "Terrestrial Blast!" a través del sensor de orientación. Las flechas rojas indican sobre que eje debemos rotar el dispositivo para llevar a cabo el movimiento indicado también en rojo.

Durante la carga del nivel deberemos mantener el dispositivo con la orientación que constituirá la orientación base en la ejecución de dicho nivel. Es decir, la posición en la que pongamos el dispositivo durante dicha carga será la posición que mantendrá al personaje principal inmóvil; y las rotaciones aplicadas sobre los ejes partiendo de esta posición base son los que provocarán el movimiento del personaje principal.

Nótese que la orientación base del dispositivo es independiente de la dirección de los ejes, pues aunque el centro de coordenadas se corresponde con el centro del dispositivo, la dirección de los ejes se calcula a través de la brújula digital y, por tanto, siempre es con respecto al suelo. Por tanto, si tenemos el dispositivo con la pantalla orientada hacia el cielo, orientada hacia el jugador o en una posición intermedia, el eje Z seguirá siendo perpendicular a nosotros tal y como lo vemos en la Figura 7.3 y la rotación tendrá que ser siempre sobre dicho eje.

A la hora de rotar el dispositivo hay dos niveles de velocidad, tanto para el desplazamiento en horizontal como el desplazamiento en vertical. Si rotamos el dispositivo entre 10 y 15 grados, el movimiento será de nivel 1. En cambio, si igualamos o excedemos los 15 grados el movimiento será de nivel 2 y, por tanto, más rápido.

En cuanto a las habilidades, el personaje principal tiene dos. En primer lugar, tenemos el disparo. Para activarlo deberemos accionar el botón que aparece en la esquina inferior derecha de la pantalla (ver Figura 7.4). Cada vez que accionemos el botón una nueva bola de energía será disparada, sin límite alguno de disparos. Con estas bolas de energía dañaremos y destruiremos a los enemigos y obstáculos.

Figura 7.4: Captura de pantalla de “Terrestrial Blast!” donde se señalan los diferentes elementos del Hud del videojuego.
También se intenta explicar a través de la imagen cómo se arrastran elementos por el escenario. Y, por último, se muestran los bloques de piedra y las marcas de la pared donde dichos bloques deben posicionarse.

La segunda habilidad es la telequinesis, con esta habilidad podemos arrastrar una serie de objetos a través del escenario. Para utilizarla, deberemos presionar sobre el área de la pantalla táctil ocupada por el elemento que queremos arrastrar y, sin despegar el dedo de la pantalla, arrastrarlo hasta el lugar que queramos (ver Figura 7.4). Si movemos rápidamente el dedo la velocidad se propagará al objeto, de forma que no solo se pueden arrastrar objetos sino que también podemos lanzarlos. Cabe decir que no podemos arrastrar cualquier elemento, solo los elementos que tengan dicha propiedad.

Esta habilidad sirve para completar los puzles que encontremos en los diferentes niveles, así como para quitar obstáculos que estén bloqueando algún camino. Igual que ocurría con la habilidad anterior, la telequinesis no tiene ningún tipo de límite de uso.

Ya hemos detallado los movimientos y habilidades del personaje, así como la forma de utilizarlos pero, ¿Cuál es el objetivo que tenemos que cumplir en cada nivel del videojuego “Terrestrial Blast!”?

Pues bien, el objetivo de todo nivel del videojuego es abrir la puerta que se encuentra en cada nivel y cruzarla. Cuando crucemos la puerta el nivel se habrá completado con éxito. Para abrir la puerta debemos completar un puzle oculto en cada escenario. Y es que por el escenario, se encuentran distribuidos unos bloques de piedra, debemos encontrarlos y posicionarlos en su sitio, señalado con una marca en la pared (ver Figura 7.4). En el momento en que todos los bloques de piedra estén en su sitio la puerta se abrirá.

Hay niveles cuya puerta ya está abierta desde el inicio, hay otros que tienen un puzle de una sola pieza (o bloque de piedra) y también hay otros niveles más complejos donde hay que buscar y posicionar varios bloques.

Pero mientras resuelve los puzles del nivel el jugador tiene que ir con cuidado, pues los niveles están plagados de enemigos. El personaje principal parte con 4 vidas a la hora de comenzar cada nivel, lo que significa que solo podrá aguantar cuatro golpes de los enemigos. En caso de sufrir el cuarto golpe el personaje principal es derrotado y el nivel fracasa, así que habrá que comenzar este nivel de nuevo. Eso sí, para dar tregua al personaje principal, cuando este sea golpeado parpadeará durante unos instantes de tiempo, momento en el cual no podrá ser golpeado de nuevo. Este parpadeo se ha implementado con el objetivo de que el jugador tenga tiempo de esconderse en caso de ser golpeado; pero cuidado, porque los enemigos también presentarán esta invulnerabilidad instantes después de ser golpeados.

La vida del personaje principal se mostrará en la esquina superior izquierda de la pantalla del nivel, para que el jugador tenga presente en todo momento cuántas vidas le queda (ver Figura 7.4).

2.1. Enemigos y obstáculos

A la hora de jugar los diferentes niveles encontraremos una serie de enemigos y obstáculos que nos complicarán las cosas, pues completar dichos niveles no será tan fácil como puede parecer. Los enemigos y obstáculos que encontraremos a nuestro paso son los siguientes:

Bloque de hielo: Este tipo de bloque no afecta negativamente a nuestra salud, pero impide el paso. Para deshacernos de estos bloques podemos: o disparar contra ellos para romperlos o arrastrarlos con el dedo (tocando la pantalla táctil) para quitarlos del medio.

Tanque: Se trata de un enemigo hostil, si nos toca nos restará una vida. Los hay de diferentes variantes, algunos se mueven, otros disparan y otros se mueven y disparan. Para deshacernos de ellos deberemos dispararles, con un solo disparo bastará.

Cañón: Otro enemigo hostil, el cual nos quita una vida por cada golpe ya sea directo o a través de uno de sus disparos (ver Figura 7.5). Este enemigo no puede ser abatido, pues son cañones estáticos muy resistentes que forman parte de la propia guarida.

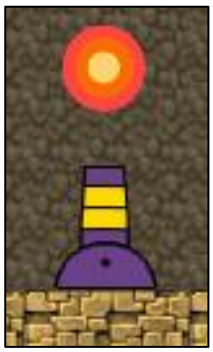


Figura 8.5: Enemigo Cañón de “Terrestrial Blast!”.

Murciélago: En su vertiente más inofensiva, son animales que se mueven de forma alocada por el escenario. Son escurridizos y si nos tocan nos dañarán, pero con simplemente un disparo se pueden abatir.

Eso sí, existe una variante de la especia más grande. Este tipo de murciélago dispara ondas de energía y requiere de dos disparos para ser abatido.

Anexo II – Construyendo niveles para el videojuego “Terrestrial Blast!”

En este anexo se explica cómo crear un nivel nuevo para el videojuego “Terrestrial Blast!” únicamente haciendo uso del lenguaje XML. Pues el juego se ha dotado con un lector de XML que permite leer un formato concreto de fichero e interpretarlo en forma de nivel del videojuego.

1. Introducción

Como ya se ha explicado en el capítulo 4, el videojuego “Terrestrial Blast!” ha sido diseñado con el objetivo de combinar dos cualidades: la eficiencia necesaria para que el videojuego funcione de forma fluida y la capacidad de expansión de su código por tal de que sea fácil añadir o modificar funcionalidades (lo que llamamos alta extensibilidad).

Para conseguir una alta extensibilidad, cada entidad u objeto del videojuego se compone de varios componentes y cada uno de estos componentes se encarga de un aspecto concreto de dicho objeto. Para algunos componentes se han implementado varios subtipos y cada uno de estos realiza su función de una forma diferente. Pero aparte de la variedad de subtipos tenemos que destacar que cada uno de estos está diseñado de forma que pueda ser configurable, dando lugar a una variedad de posibilidades mayor.

Es a través del lenguaje XML que creamos los objetos escogiendo qué componentes queremos que dicho objeto tenga, que subtipo de cada componente queremos y cómo configuramos este subtipo para el objeto concreto dado. Como podemos entrever esto nos da un sinfín de combinaciones y, por tanto, de objetos posibles; todo ello reutilizando los subtipos de componente implementados.

En este apartado trabajaremos un nivel de abstracción por encima del lenguaje Java y nos centraremos en cómo utilizar los componentes implementados en este lenguaje para construir un nivel del videojuego y, por tanto, todos los objetos que componen este nivel. Para ello, como se ha indicado, utilizaremos el lenguaje XML.

Cabe decir que crear un nuevo subtipo de componente, en la mayoría de los casos, es tan fácil como crear una subclase del componente elegido a través del lenguaje Java. Además, si queremos, podemos concebir dicho subtipo de forma genérica permitiendo configurarlo también a través de XML, lo que llevaría a una ampliación de esta guía.

2. Creación de un nuevo nivel para el videojuego

Vamos a ver en detalle cómo podemos crear un nuevo nivel para el videojuego únicamente haciendo uso del lenguaje XML.

Para ello llevaremos a cabo tres pasos:

1. Primeramente definiremos el nivel (explicado con detalle en el apartado 2.1 de este mismo anexo). Los niveles del videojuego se definen cada uno en un fichero XML propio cuyo nombre debe ser "*level<numero_nivel>.xml*" donde *<numero_nivel>* es el número que identificará al nivel que estamos creando. Estos ficheros se introducen todos dentro del directorio "`/assets/levels/`" del proyecto para Eclipse.

2. En segundo lugar, para que la aplicación reconozca el nivel, la constante *NIVELES_TOTALES* (declarada dentro de la clase *InformacionDelSistema,* en el paquete *com.juego.terrestrial.sistema*) es la que tiene la última palabra y deberá contener el número máximo de niveles. Nótese que los niveles que se podrán jugar serán los que tengan asignado los números entre 1 y *NIVELES_TOTALES*, ambos incluidos.

   Por tanto, el valor asignado a la constante debe ser coherente y deben existir todos los niveles que en ella indicamos.

3. Por último, todo nivel debe tener asignado un icono, el cual es el que se presentará en la pantalla de *selección de nivel*. Dicho icono se introducirá dentro del directorio "`/assets/images/`" con el nombre "*level<numero_nivel>.png*" (el formato debe ser obligatoriamente PNG).

Nótese que en caso de que el nivel no quepa en la pantalla de *selección de nivel* se habilitarán unas flechas que nos permitirán retroceder y avanzar a lo largo de las páginas de niveles por tal de seleccionar el nivel que queramos de entre todos los niveles disponibles.

2.1. Formato del fichero XML donde se define un nivel

Una vez vistos todos los pasos necesarios para añadir un nuevo nivel al sistema, vamos a centrarnos en el paso 1, pues es necesario conocer el formato que debe seguir la definición de un nivel en XML si queremos que el nivel que creemos sea interpretado correctamente por el sistema.

En primer lugar, debemos tener en cuenta la sintaxis y la semántica de XML [21]. La sintaxis se puede entrever a través de los ejemplos que aparecerán a continuación. Y en cuanto a la semántica esta aporta dos exigencias principales: que toda etiqueta tenga una apertura y un cierre y que cada fichero XML se vea como un árbol de etiquetas, el cual debe tener una única etiqueta raíz. En el caso del fichero XML donde se define un nivel, la etiqueta raíz será `<definicionnivel>`.

En segundo lugar hay que tener en cuenta las restricciones semánticas que se han impuesto en el desarrollo del videojuego para que la aplicación pueda interpretar correctamente el nivel. Un fichero que define un nivel se pude dividir en dos partes bien diferenciadas: La parte donde se define la música y la parte donde se definen los objetos complejos que componen el nivel.

La parte donde se define la música únicamente deberá aparecer si queremos que durante la reproducción del nivel suene una música de fondo. El formato es el siguiente:

```
<audionivel>
      <musica>
            <pista path="constance.ogg">
                  <accion nom="musicaprincipal"/>
            </pista>
      </musica>
</audionivel>
```

Ejemplo 8.1: Definición de la pista musical de un nivel.

Donde *path* es el nombre del fichero que contiene la pista de audio a reproducir. En concreto, el nombre relativo al directorio "`/assets/music/`", pues la pista de audio deberá estar allí almacenada. Por tanto, el nombre absoluto de la pista de audio será "`/assets/music/<path>`".

En cuenta al nombre de la acción, este es *musicaprincipal* y sirve para indicar que estamos definiendo la pista de audio que contiene la música principal del nivel.

Por ahora, el nivel no brinda la posibilidad de asociar más pistas de audio ya sea para la asignación de músicas a momentos concretos de la reproducción del nivel o para la asignación de sonido a determinados sucesos del propio nivel.

El motivo por el cual no se ha dado esta posibilidad es el hecho de que en los niveles desarrollados esto no ha sido necesario. Pero el código invita a que, si en futuras versiones es necesario, se extienda dando cabida a mas ficheros de audio. Es por este motivo que especificamos la acción a la que la pista da respuesta, a pesar de que solo haya una acción posible; para facilitar la extensión del número de pistas que un nivel puede utilizar.

En cuanto a la parte donde se definen los objetos complejos que conforman el nivel, estos van dentro de la etiqueta `<objetos>` y su orden de definición es importante. Estos objetos, a la hora de construir cada uno de los *renders* que presentarán la evolución en la reproducción de un nivel del videojuego, se pintaran en el orden en que están definidos en nuestro fichero XML. Por ejemplo, si queremos que un personaje pueda solaparse con una puerta en un nivel, probablemente nos interesare que la puerta quede por detrás del personaje cuando este se abalance sobre ella. Para conseguir el resultado deseado en este caso, la puerta deberá ser declarada antes del personaje controlado por el usuario.

Normalmente el primer objeto declarado es el que representa el fondo del escenario, en segundo lugar viene la declaración del escenario, en un tercer nivel posicionamos los objetos que queremos que queden por detrás de los enemigos y del personaje que controlamos (puertas, adornos, etc.) y, por último, posicionamos los objetos que se mueven dinámicamente y tienen una física propia, los cuales interaccionarán entre ellos (enemigos, personaje principal, cajas, etc.).

2.2. Formato de un objeto complejo

Ya hemos visto qué formato tiene la definición de un nivel, pero aun nos queda la parte más importante, y es que un nivel se compone de objetos complejos, vamos a ver, por tanto, cual es el formato a la hora de declarar y definir un objeto complejo.

2.2.1. Declaración de un objeto

A continuación tenemos dos ejemplos de declaración de un objeto. Uno está declarado de forma explícita y el otro de forma implícita:

```
<objeto declaracion="explicita" x="640" y="448" dinamico="falso"
rol="normal">

<objeto declaracion="implicita" x="512" y="768" dinamico="falso"
rol="normal" nom="puerta"/>
```

Ejemplo 8.2: Declaración de un objeto complejo de forma implícita y de forma explícita.

Como podemos observar hay una serie de atributos, la mayoría de los cuales son obligatorios. Vamos a describir cada uno de ellos:

- Por un lado, el atributo `declaracion` especifica el tipo de declaración empleada a la hora de definir el objeto.

  Si la declaración es *explicita* el objeto en su totalidad se declarará en el propio fichero XML donde estamos declarando el nivel del juego.

  En cambio si el objeto se declara de forma *implicita*, su definición se introducirá en un fichero XML individual. Esta opción es muy útil si tenemos un objeto que aparece múltiples veces en uno o más niveles. Definiéndolo en un fichero a parte evitamos repetir esta definición en cada declaración, simplemente tenemos que apuntar a la declaración común. Además, los cambios que hagamos sobre dicha definición afectarán a todas sus declaraciones.

- En segundo lugar, tenemos dos atributos que representan el punto central dentro del nivel (en píxeles) del objeto que estamos declarando. Estos son `x` e `y`.
  Se trata de una información necesaria para cada declaración, pues cada objeto declarado estará en un lugar diferente del escenario, tengan en común la misma definición o no.

  Cabe decir que la posición se declara respecto al eje de coordenadas del espacio de pintado OpenGL. Somos nosotros los que tenemos que asegurarnos de que todos los objetos del nivel se encuentren en un lugar coherente y puedan llegar a aparecer por pantalla.

- En tercer lugar, el atributo `dinamico` sirve para agilizar la reproducción del nivel. Deberá valer *`cierto`* solo si se trata de la definición de un objeto del cual se van a declarar nuevas instancias dinámicamente durante la reproducción del nivel.

  Cabe decir que solo se pueden declarar dinámicamente objetos definidos de forma implícita, es por esto que únicamente tendrá sentido un valor *`cierto`* para este atributo cuando la declaración corresponda con la de un objeto definido implícitamente.

  De esta forma conseguiremos librarnos de la lectura del fichero XML durante la simulación del nivel y esto es muy importante, pues esta lectura es muy lenta. Cuando el nivel se esté cargando, el sistema identificará el atributo y, en caso de tener un valor afirmativo, la definición quedará guardada en memoria principal y no habrá que irla a buscar al fichero XML cuando se cree una instancia del objeto dinámicamente.

  En caso de querer crear una instancia de objeto dinámicamente y no tener la definición de este en memoria principal, solo iremos a buscarla al fichero XML la primera vez, pues después de esta la definición también quedará guardada en memoria principal, por si en el futuro se vuelve a crear una instancia dinámica de este tipo de objeto.

- El último de los atributos obligatorios es el "rol". Todo objeto tiene un rol y este indica qué papel juega el objeto dentro del nivel. Eso sí, solo hay tres tipos de roles deferentes:

  - El rol *`escenario`*: Se utiliza para declarar el objeto que hará la función de escenario dentro del nivel. De este objeto se extraerán las físicas que constituirán las paredes de este nivel, pues deberá tener definida la matriz de tiles que dará forma a dichas paredes, tanto físicamente como visualmente. En cada nivel debe haber un y solo un objeto con el rol *`escenario`*.

  - El rol *`pp`*: Se utiliza para declarar el objeto que hará la función de personaje principal dentro del nivel. El personaje principal es el objeto que nosotros manejamos a través de los controles del dispositivo. Igual que pasa con el rol anterior, debe haber un y solo un objeto con el rol *`pp`* en cada nivel.

  - Por último tenemos el rol *`normal`*: este es el rol que utilizan el resto de objetos que no son ni el escenario ni el personaje principal.

- En el caso concreto en que el objeto declarado se defina implícitamente, es necesaria la inclusión de un nuevo atributo: el atributo “nom”. Este contiene el nombre relativo del fichero XML donde se encuentra la definición del objeto que estamos declarando implícitamente, necesario para saber de dónde debemos extraer esta definición. El nombre que aparecerá no incluirá la extensión “.xml”, puesto que al ser común a todos los nombres la obviamos.

  El nombre aportado es el nombre relativo al directorio “`/assets/objects`” dentro del cual deberán encontrarse todos los ficheros XML que definan a tipos de objeto por sí solos. Entendemos a un tipo de objeto como a la definición de un objeto implícitamente, a través de la cual, luego podremos crear diferentes instancias de dicho tipo de objetos, cada instancia es a lo que también llamamos objeto.

  En el ejemplo, el objeto declarado implícitamente tiene el nombre “puerta” lo que significa que el nombre absoluto del fichero XML donde se encuentra la definición de este tipo de objeto es “`/assets/objects/puerta.xml`”.

2.2.2. Definición de un objeto

Ya hemos visto como declarar una instancia de un tipo de objeto en un nivel. Ahora vamos a ver cómo definir dicho tipo de objeto, pues todo objeto declarado debe tener un tipo y este debe estar declarado en algún lugar.

La diferencia entre definir un tipo de objeto de forma implícita o explícita, radica en que si lo hacemos de la primera forma, deberemos crear un fichero XML únicamente para este tipo de objeto y definir cada uno de los componentes del tipo entre las etiquetas de apertura y cierre `<componentes>`, que conformará la raíz de dicho fichero. En cambio, si definimos un tipo de objeto de forma explícita la definición se añadirá entre la apertura y cierre de la propia etiqueta `<objeto>` que declara la instancia del tipo de objeto (descrita en el apartado anterior).

La desventaja de esta segunda forma de definir tipos de objeto es que tan solo podremos utilizar el tipo en cuestión para la instancia que se está declarando en un momento dado. Por tanto, solo es aconsejable la definición explícita si el tipo de objeto no aparecerá en más ocasiones, ni en este nivel ni en ningún otro.

Pues bien, especificadas las etiquetas dentro de las cuales se va a definir el tipo de objeto según el tipo de declaración, la forma de definirse un objeto es idéntica en ambos casos: deberemos definir cada uno de los componentes que compondrán el tipo de objeto a crear. Como ya se ha explicado en apartados anteriores, un objeto complejo puede tener los siguientes componentes: componente físico, componente lógico, componente de habilidad, componente de comportamiento, componente musical y componente visual.

Un tipo de objeto es libre de implementar los componentes que quiera y dejar de implementar los que también desee, pero hay ciertas dependencias entre ellos que se comentarán a medida que veamos cómo definir cada uno de ellos.

Vamos pues a ver las opciones por defecto que tenemos a la hora de definir cada uno de los tipos de componente citados.

Definición del Componente Físico

El componente físico es el que dota al tipo de objeto que definimos de presencia física dentro del mundo físico. Por tanto, únicamente no tendrá componente físico el objeto que sea un mero adorno visual y no haga mayor función dentro del nivel donde se encuentre.

A continuación tenemos un ejemplo de definición de un componente físico:

```
<fisico nom="CajaFisica" tipo="dinamico" rotacionfija="true"
antigravedad="false">

      <forma densidad="1" anchura="64" altura="58"/>

</fisico>
```

Ejemplo 8.3: Definición del componente físico de un tipo de objeto.

En primer lugar vamos a explicar la función que realizan los atributos declarados en la propia etiqueta que marca el inicio de la definición del componente físico. Algunos de estos atributos serán comunes a todos los componentes pero otros no:

- nom: Este atributo indica el nombre de la subclase de *ComponenteCuerpoFisico* que se va a utilizar para declarar el componente físico en cuestión.

  Todo componente debe especificar el nombre de la clase que se empleará para implementar las funciones que el componente deba llevar a cabo. Por lo general, existirá la posibilidad de emplear la superclase que hace de padre de la jerarquía de implementaciones para el componente en cuestión, pues este constituirá la implementación más simple y limitada. O, por el contrario, podremos emplear las subclases de esta jerarquía, unas implementaciones alternativas y, por norma general, más complejas que la superclase.

  En el caso del componente físico siempre debemos especificar una de las dos subclases disponibles: *CajaFisica* o *CircunferenciaFisica*. Esto es debido a que la superclase es común a todas las implementaciones e implementa la parte que define al cuerpo físico en sí mismo; en cambio, la subclase lo que está definiendo es la forma que el cuerpo físico tiene. Todo cuerpo físico debe tener forma y esto hace que sea imprescindible escoger una de las dos subclases.

La diferencia entre las subclases disponibles está en que *CajaFisica* sirve para representar cuerpos físicos con forma rectangular, mientras que *CircunferenciaFisica* sirve para representa cuerpos físicos con forma circular.

- `tipo`: Este segundo atributo indica que tipo de cuerpo físico se va a declarar. La librería de físicos Box2D interpreta tres tipos de cuerpos físico diferentes y, a través de este atributo, nosotros podemos especificar cuál de los tres tipos de trato recibirá nuestro objeto. Los tipos posibles son: *`dinamico`*, *`estatico`* y *`kinematico`*. La diferencia entre los tres tipos de cuerpos físicos se explica con detalle en el anexo *Box2D*.

- `rotacionfija`: Este atributo nos permite especificar si nuestro objeto puede rotar o no. Si el objeto puede rotar, cuando se mueva dentro del entorno de físicas puede que adquiera una velocidad angular. En cambio, si el objeto no rota, siempre mantendrá su ángulo de inclinación inicial y su velocidad angular siempre será 0.

  Los valores posibles son *`true`* o *`false`*.

  Atributo opcional, en caso de no introducirse el objeto podrá rotar libremente.

- `antigravedad`: En caso de que un cuerpo físico sea anti gravitatorio este no se verá afectado por la gravedad y se podrá mover en cualquier dirección sin resistencia alguna. Por otro lado, el objeto que no tenga anti gravedad se verá afectado por esta y no podrá flotar. Con *`true`* o *`false`* activamos o desactivamos esta propiedad.

  Atributo opcional, en caso de no introducirse el objeto no será anti gravitatorio.

- `angulo`: Especifica el ángulo de giro inicial de la forma física. Por ejemplo, si su valor es 90, la forma física se presentará, por defecto, volteada 90 grados; y si la rotación fija está activada, dicho ángulo se mantendrá durante toda la vida del cuerpo físico en cuestión.

  Atributo opcional, en caso de no introducirse el cuerpo físico se presentará por defecto con un ángulo de rotación de 0 grados.

- `incrementox` e `incrementoy`: Par de valores que indican el incremento que sumado al centro visual del objeto nos proporciona el centro físico de este. Este incremento se define en píxeles.

Cuando el objeto se declara, como hemos visto en el apartado 8.4, se especifica una posición central. Esta posición central define el llamado centro visual del objeto: centro alrededor del cual se pintará el objeto (en caso de que el objeto tenga componente *visual)*. Normalmente el centro físico, alrededor del cual se encontrará el cuerpo físico de un objeto, coincidirá con el centro visual, pero habrá casos en que no queramos que coincidan. Es por eso que tenemos a nuestra disposición estos dos atributos, que se suman al centro visual dando lugar al centro físico, tal y como especifica la siguiente fórmula: (`<objeto x>` + `incrementox`, `<objeto y>` + `incrementoy`).

Así que, para conseguir que el centro visual difiera del centro físico únicamente tenemos que darle valor a estos incrementos. Y puede ser un valor positivo o negativo, según nos convenga. En caso de no definir-los (o se definen los dos o no se define ninguno), el valor por defecto para ambos será 0.

Una vez definidos los atributos que todo componente físico debe o puede tener, es hora de definir los atributos específicos de la forma escogida para este componente. Y es que, como ya se ha explicado, a través del atributo `nom` hemos escogido una forma concreta para el cuerpo físico que estamos definiendo; cada forma necesitará unos atributos específicos y estos se declaran en la etiqueta `<forma>`, etiqueta que a su vez estará declarada dentro de la etiqueta `<fisico>` (en el ejemplo 8.3 podemos apreciar el formato que estamos describiendo).

- Si elegimos una forma rectangular especificando la subclase *CajaFísica*, los atributos necesarios serán los siguientes:
    - `densidad`: Atributo obligatorio que indica la densidad del cuerpo físico que estamos definiendo. A través de la densidad, de la forma y de las dimensiones del cuerpo se extrae su masa.
    - `anchura` y `altura`: Estos atributos son obligatorios y contienen las dimensiones en pixeles del rectángulo físico que estamos definiendo.
    - `friccion`: Atributo opcional que nos permite especificar la fricción del cuerpo que estamos definiendo, dentro del mundo físico gestionado por la librería de físicas Box2D. Para más información acerca del funcionamiento de la fricción en Box2D leer el anexo *Box2D*.
    - `sensor`: En caso de definirse y tomar el valor *`true`* el componente físico no colisionará con el resto de componentes, físicamente será transparente. Eso sí, la librería Box2D nos permitirá consultar datos del cuerpo físico, como por ejemplo el solapamiento con otros cuerpos; pues el cuerpo estará dentro del mundo físico.

Este atributo es útil para objetos que no queremos que colisionen contra los objetos físicos que se mueven dinámicamente por el escenario, pero que sí que queremos que en caso de solaparse con estos produzcan algún efecto o que tengan componente lógico (el cual exige la existencia de un componente físico). Un ejemplo de uso de este atributo puede ser a la hora de definir una puerta, queremos que el personaje pueda posicionarse encima de ella, y también que se nos avise cuando este personaje se haya solapado, para dar por concluido el nivel.

Es un atributo opcional y en caso de no definirse tomará el valor *false*, lo que significa que el cuerpo físico colisionará con el resto de cuerpos que no sean sensores.

- Si, por el contrario, elegimos una forma circular, especificando la subclase *CircunferenciaFisica*, los atributos disponibles serán prácticamente los mismos. Continuaremos pudiendo definir la `densidad`, la `friccion` y dando a nuestro cuerpo la propiedad de `sensor`. Pero en este caso, en lugar de especificar las dimensiones a través de los atributos `anchura` y `altura`, únicamente tendremos el atributo `radio`, un atributo que contendrá el radio de la circunferencia que estamos definiendo en píxeles y que también será obligatorio.

Definición del Componente Lógico

En segundo lugar, vamos a centrarnos en la definición del componente lógico. Cabe decir que para que un objeto pueda tener componente lógico es obligatorio que disponga de un componente físico también, puesto que gracias a la librería de físicas Box2D podemos detectar las colisiones que nos ayudarán a mantener la lógica del objeto consistente con la evolución del nivel.

Como se ha explicado con anterioridad el componente lógico es el que mantiene la lógica del objeto, es decir el estado de este dentro del mundo lógico. Dentro de este mundo cada objeto posee un número de vidas, unos objetivos que debe cumplir para lograr la satisfacción, junto con otros atributos que vamos a ver con detalle a continuación.

Se presenta el ejemplo de definición de un componente lógico que utilizaremos para explicar los atributos que este componente requiere:

```
<logico nom="PuertaNivel" arrastrable="false" grupoataque="pasivo"
grupodefensa="inmune" numvidas="1" tipo="puerta" subtipo="0">

</logico>
```

Ejemplo 8.4: Definición de la parte común de componente lógico para un tipo de objeto.

Los atributos comunes a todo componente lógico se introducen en la etiqueta `<logico>` y son los siguientes:

- `nom`: Como ocurre con el componente físico y como va a ocurrir con todos los componentes que tenemos que definir, el nombre de la clase que vamos a utilizar para implementar el componente que estamos definiendo es imprescindible.

  En este caso la diferencia principal entre escoger una clase u otra radica en los objetivos que debe cumplir un componente para conseguir la satisfacción.

  Puede haber componentes lógicos cuyos objetivos estén satisfechos ya desde el inicio, se trata de los componentes que implementan la superclase, *ComponenteLogico*.

  En cambio hay componentes lógicos que tienen unos objetivos concretos a cumplir y hasta que no los cumplan no pasarán a estar satisfechos. Estos pueden implementar la subclase *PuertaNivel*, que solo estará satisfecha cuando todas las lógicas del tipo `puzzle` del nivel lo estén y, además, el personaje principal se solape con la puerta en cuestión. O pueden definir la subclase *PuzzleDePiezas,* que implementa una matriz con huecos donde encajar piezas de puzles y solo pasará a estar satisfecha cuando cada pieza este en el lugar que le corresponde.

  Pero ¿Qué significa que un objeto está satisfecho y porqué tiene importancia la satisfacción de los objetos?

  Todo objeto dotado con un componente lógico tiene unos objetivos que cumplir y una vez cumplidos pasará a estar satisfecho. Como ya se ha explicado anteriormente, puede ser que los objetivos de un componente lógico sean simplemente crearse, de forma que dichos componentes estén satisfechos desde el inicio del nivel, o puede que este componente tenga que satisfacer los objetivos más complejos comentados con en el párrafo anterior.

  En todo caso, es importante la satisfacción de los componentes lógicos ya que un nivel del videojuego se habrá completado en un momento dado si y solo si todos los componentes lógicos de todos los objetos que permanecen vivos en el nivel están satisfechos. Por tanto, se puede decir que el objetivo de un jugador en este videojuego es satisfacer todos los objetos que componen un nivel.

- `arrastrable`: Si este atributo tiene el valor `true` significa que la habilidad *arrastrar*, la habilidad que se ejecuta cuando el jugador pulsa un lugar del nivel con el objetivo de mover un elemento del escenario, podrá utilizarse para mover el objeto en cuestión. Es decir, estaremos definiendo un objeto vulnerable a la habilidad *arrastrar* y que, por tanto, se podrá ver afectado por dicha habilidad.

- `grupoataque`: Todo componente lógico debe tener un grupo de ataque, según al grupo al que pertenezca dañará a unos u otros grupos de defensa.

  Es a través del grupo de ataque y del grupo de defensa que se define a quién daña y por quién puede ser dañado un objeto del nivel.

  Existen cuatro grupos de ataque posibles: los objetos que pertenezcan al grupo de ataque *`enemigo`* dañarán al grupo de defensa *`aliado`* y *`explosivo`*; los que pertenezcan al grupo de ataque *`aliado`* dañarán únicamente a *`explosivo`*; los que pertenezcan al grupo de ataque *`explosivo`* dañarán a *`aliado`* y *`enemigo`*; y, por último, los que pertenezcan al grupo de ataque *`pasivo`* no dañarán a nadie.

- `grupodefensa`: De igual forma que un componente lógico debe tener un grupo de ataque, todo componente lógico debe tener también un grupo de defensa. Según el grupo de defensa al que pertenezca un componente lógico, este será dañado por un grupo de atacantes u otro. Existen tres grupos de defensa: los del grupo de defensa *`enemigo`* solo son dañados por atacantes pertenecientes al grupos de ataque *`explosivo`*; los del grupo de defensa *`aliado`* serán dañados por los grupos de ataque *`explosivo`* y *`enemigo`*; los del tipo *`explosivo`* son dañados por atacantes del tipo *`aliado`*, *`enemigo`*, *`explosivo`* y por objetos sin lógica, es decir, por todos los objetos menos por los que pertenezcan al grupo de ataque *`pasivo`*; y, en último lugar, tenemos al grupo de defensa *`inmune`* que no es dañado por ningún otro objeto del nivel.

  Es importante tener en cuenta que un objeto no tiene por qué tener el mismo grupo de ataque que de defensa, los dos grupos son independientes y tienen un significado diferente. Por ejemplo, un componente lógico que pertenezca al grupo de ataque *`enemigo`* no tiene por qué tener un grupo de defensa *`enemigo`*.

  Por último, es importante tener en cuenta que un objeto que no tenga lógica daña a *`explosivo`* y no es dañado por nadie. Por tanto, si queremos que un objeto no dañe absolutamente a nadie tendremos que dotarle de lógica y asignarle el grupo de ataque *`pasivo`*.

- `numvidas`: Todo objeto con lógica debe tener un número de vidas. Cada vez que el objeto tope con otro cuyo grupo de ataque pueda dañar a su grupo de defensa, se le restará una vida. Si al restar la vida el objeto se queda con un total de 0 vidas, este muere y desaparece del nivel; entonces su satisfacción dejaría de importar de cara a la completitud del nivel. En caso de que sea el personaje principal el que se queda sin vidas, la reproducción del nivel termina y se muestra la pantalla de fracaso.

  Hay que tener en cuenta que, para que el ataque no sea tan abusivo y un objeto dañado tenga tiempo de ponerse a cubierto, cada vez que se daña a un objeto este entra en estado *debilitado*. El estado *debilitado* dura tantos frames como se especifique en la constante *NUM_FRAMES_DEBILITADO* de la clase *ConfigOComplejo* y durante estos frames el objeto será inmune a todo ataque.

- `tipo`: El tipo al que pertenece un componente lógico es útil para que, cuando utilizamos las subclases *PuertaNivel* y *PuzzleDePiezas*, estas puedan consultar su satisfacción de forma eficiente. Es por eso que *PuertaNivel* consultará solo las lógicas cuyo `tipo` sea *`puzzle`* pues son los puzles los que, junto con el solapamiento entre la puerta y el personaje principal, van a decidir si la lógica está satisfecha o no. Por otro lado, *PuzzleDePiezas* exige que todos los objetos que quieran hacer el papel de pieza tengan asignado el tipo *`pieza`*.

- `subtipo`: Se trata de otro nivel de agrupación más para el componente lógico, un número natural que indica a que subtipo dentro del tipo dado pertenece una lógica. Por el momento, el único tipo que requiere subtipo es *`pieza`*, es a través del subtipo que especificamos de qué pieza estamos hablando. Lo que permite que, si lo deseamos, las diferentes piezas de un puzle no sean intercambiables, puesto que cada subtipo de pieza deberá posicionarse en la posición que le pertenezca.

  El entero que representa al subtipo debe ser mayor o igual a 0.

a. Lógica implementada mediante PuzzleDePiezas

En caso de que la lógica se declare de tipo *PuzzleDePiezas* se deberá añadir una información específica a la definición del componente. Se trata de una matriz de enteros donde, en cada posición, se indicará el subtipo que deberá tener la pieza de puzle que se posicione en dicha posición.

A continuación tenemos un ejemplo de especificación de la matriz que define un *PuzzleDePiezas*:

```
<matriz  numfilas="1" numcolums="1" anchuracasilla="10"
alturacasilla="10">

<row Position="0">

      <column Position="0" Value="0"></column>

      </row>

</matriz>
```

Ejemplo 8.5: Definición de la parte de Componente Lógico específica en el caso de que elijamos la subclase *PuzzleDePiezas* para implementar este componente.

La etiqueta `<matriz>` especifica la declaración de la matriz que conforma el puzle, esta etiqueta debe ir entre la apertura y el cierre de la etiqueta que indica la definición del componente lógico `<logico>`. Los atributos que `<matriz>` debe tener son los siguientes:

- `numfilas` y `numcolums`: Se trata del número de filas y de columnas que tendrá la matriz que representará al puzle de piezas.
- `anchuracasilla` y `alturacasilla`: Se trata de la anchura y la altura de cada una de las casillas del puzle (en píxeles).

  El área donde deberá posicionarse cada pieza de puzle queda definida, por tanto, por: el centro del objeto dado en la etiqueta `<objeto>`, que será el centro de la matriz lógica que estamos definiendo; y estas dimensiones, que especifican cómo de grande será el área de cada casilla.

Una vez definidos los atributos de la etiqueta `<matriz>`, es hora de rellenar la matriz. Para este cometido, dentro de la etiqueta `<matriz>`, se creará una etiqueta `<row>` por cada fila existente en dicha matriz. Esta etiqueta requiere el atributo `Position` que contendrá el número de fila que estamos definiendo. Su rango es de 0 a N-1 (donde N es el número de filas de la matriz), teniendo en cuenta que la fila 0 es la fila que se encuentra más arriba en la matriz y la fila N-1 la fila más baja de esta.

Entre la apertura y cierre de la etiqueta `<row>` se deberá crear una etiqueta `<column>` por cada columna que tenga cada fila de la matriz (nótese que todas las filas tienen el mismo número de columnas).

Cada etiqueta `<column>` contendrá el atributo `Position` que, una vez más, indicará la posición que le corresponde a la columna que estamos definiendo, entre 0 y M-1 (donde M es el número de columnas de la matriz). La columna número 0 es la que se encuentra más a la izquierda de la matriz y M-1 la columna que se encuentra más a la derecha.

Además, también deberemos añadir el atributo `Value`, que contendrá el subtipo de pieza que se corresponde con la posición formada por la columna que estamos definiendo y la fila dentro de la cual estamos definiendo dicha columna. El -1 es un subtipo especial reservado para indicar que en una posición concreta no es necesario que vaya ninguna pieza de puzle.

Definición del Componente de Habilidad

El componente de habilidad es el componente que gestiona las habilidades que un objeto puede ejecutar. Actualmente el videojuego tiene dos habilidades implementadas: la habilidad de disparo, que nos permite disparar en cuatro direcciones, y la habilidad de arrastrado, que nos permite arrastrar objetos cuya lógica defina que se pueden arrastrar (para más información acerca de cómo definir esta propiedad ir al apartado *Componente Lógico*).

El ejemplo que utilizaremos para explicar el formato de la definición de un componente de habilidad es el siguiente:

```
<habilidad nom="ComponenteHabilidad" distlatx="44" distverty="90"
incvertx="0" inclaty="3" tiempodisparo="0" tipodisparo="bolita"></habilidad>
```

Ejemplo 8.6: Definición de un componente de habilidad para un tipo de objeto.

Y los atributos que este componente exige son:

- `nom`: En este caso, por el momento, no hay alternativas posibles, así que el componente de habilidad siempre estará implementado por la clase *ComponenteHabilidad*.

- `distlatx` y `distverty`: Estos atributos definen a qué distancia estará el centro del disparo, que se creará cuando ejecutemos la habilidad disparo, del centro del objeto que estamos definiendo. Cabe decir que el disparo no es más que otro objeto que se declara implícita y dinámicamente durante la ejecución del nivel de juego.

  En concreto, `distlatx` es la distancia utilizada cuando el disparo va hacia la derecha o hacia la izquierda mientras que `distverty` es la distancia utilizada cuando el disparo va hacia arriba o hacia abajo.

  Por último, es importante tener en cuenta que ambos atributos, como distancias que son, deben ser positivos. Es el sistema el que decidirá si deben sumarse o restarse al centro del objeto emisor del disparo, según la dirección de dicho disparo.

- `incvertx` y `inclaty`: Se trata de atributos que definen unos incrementos que se suman a las coordenadas del centro del disparo para que este quede correctamente posicionado respecto a su foco de emisión. El objetivo es ajustar la posición del disparo para que visualmente sea correcta con respecto al objeto emisor de dicho disparo.

  Cuando el disparo salga lateralmente se sumará `inclaty` a la coordenada Y del centro del objeto emisor del disparo, dando lugar a la coordenada Y del centro del objeto disparado. Por otro lado, cuando el disparo salga verticalmente se sumara `incvertx` a la X del centro del objeto emisor para calcular la X del centro del disparo.

  Esta vez no estamos tratando con distancias, por tanto, los incrementos pueden ser positivos o negativos según el lado al que queramos desviar el centro del disparo con respecto al centro del emisor del disparo.

  A continuación, se presenta una imagen donde se puede apreciar la función de los cuatro últimos atributos descritos, de una forma más intuitiva:

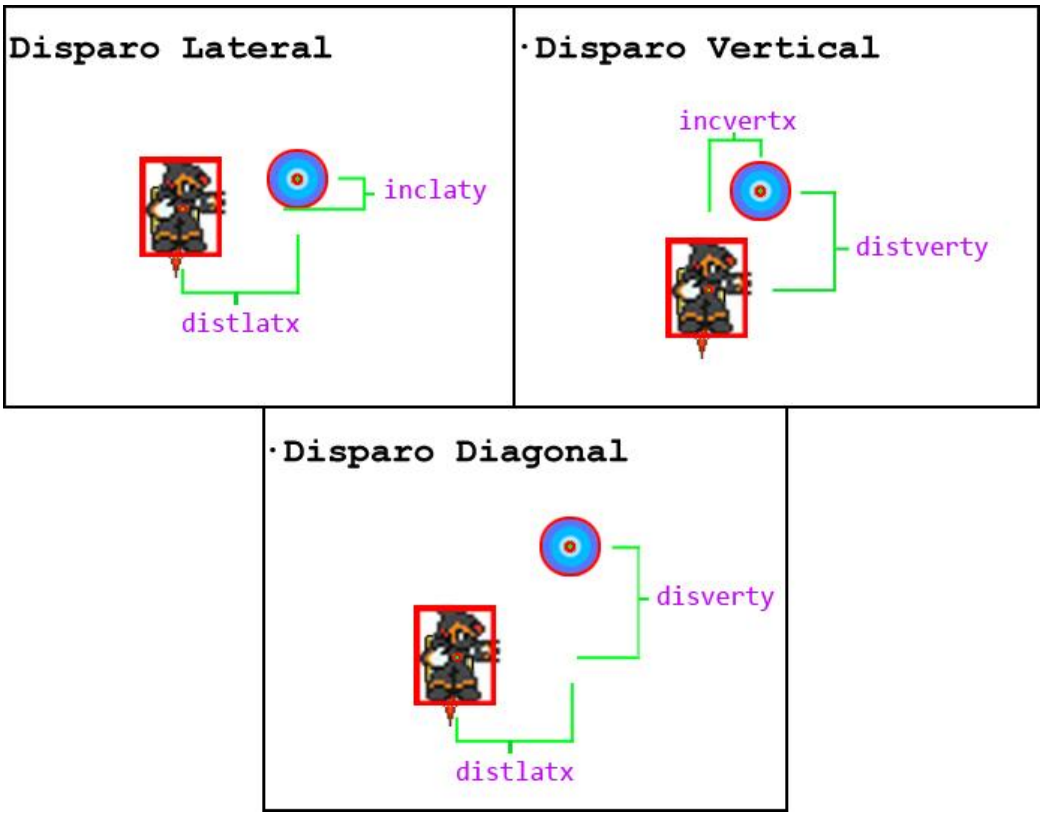


Figura 8.1: Esquema con los atributos utilizados para calcular el centro del objeto disparo a partir del centro del objeto que dispara. Distinguimos entre los diferentes tipos de disparo, pues cada tipo utiliza unos u otros atributos.
En rojo podemos ver el borde del área física ocupada por los diferentes objetos que entran en conflicto, así como el centro de cada uno de estos objetos físicos.

- `tiempodisparo`: A través de este atributo introducimos el número de frames que queremos que transcurran entre que se indica el deseo de ejecutar la habilidad disparo y se ejecuta dicho disparo. Este retardo existe para permitir el uso de animaciones de disparo por parte del objeto emisor: si utilizamos una animación para esta habilidad querremos que el disparo se cree en un momento dado de dicha animación.

  En caso de que no vayamos a introducir una animación de disparo es preferible que este valor sea 0 para dar una respuesta el máximo de rápida posible.

- `tipodisparo`: A través de este atributo indicamos el tipo de disparo que utilizaremos a la hora de ejecutar la habilidad disparo.

  Como ya se ha explicado anteriormente, un disparo no deja de ser otro objeto con declaración implícita. Por tanto, lo que realmente estamos indicando a través del nombre que asociamos con este atributo, es que el disparo es una instancia del tipo de objeto definido de forma implícita en el fichero con nombre absoluto "`/assets/objects/<nombre>.xml`".

En el ejemplo, el hecho de indicar el nombre “bolita” implica que debe de existir un fichero con nombre absoluto “`/assets/objects/bolita.xml`” dentro del sistema de ficheros de la aplicación.

Vemos que no existe ningún atributo que haga referencia a la habilidad de arrastrado. Esto es debido a que la habilidad de arrastrado no necesita de una configuración, pues siempre tiene el mismo comportamiento.

Por último, debemos tener en cuenta que sin la existencia de un Componente Físico, la habilidad de disparo no se puede llevar a cabo, pues se necesita tener el centro físico del objeto emisor para poder calcular el centro físico del objeto emitido. Así que, por este motivo, se exige que todo objeto con Componente de Habilidad también tenga un Componente Físico definido.

Definición del Componente de Comportamiento

El componente de comportamiento define como va a comportarse el objeto en cuestión. Para permitir un comportamiento personalizable con más opciones, reutilizando clases de la aplicación dividimos el comportamiento en dos subcomponentes: el comportamiento móvil y el comportamiento de habilidad.

A la hora de definir un componente de comportamiento podemos definir solo uno de los subcomponentes o ambos. Los dos son independientes y, por tanto, la clase que implemente uno de los subcomponentes no condicionará a la clase que implemente el otro.

Se presenta ahora un ejemplo de definición de un componente de comportamiento:

```
<comportamiento nom="ComponenteComportamiento">

      <compmovil nom="MovimientoArtificial" movporvel="false"
      fuerzalateral1="2" fuerzalateral2="4" fuerzaarriba1="2"
      fuerzaarriba2="5" fuerzaabajo1="2" fuerzaabajo2="5"
      limitelateral1="2" limitelateral2="5" limitearriba1="-1"
      limitearriba2="-1" limiteabajo1="-1" limiteabajo2="-1">

      </compmovil>

      <comphabilidad nom="HabilidadesArtificiales"
      disparocambiable="true" direccciondisplat="izquierda"
      direccciondispvert="arriba" direccciondisp="izquierda">

      </comphabilidad>

</comportamiento>
```

Ejemplo 8.7: Definición de un componente de comportamiento para un tipo de objeto.

Como vemos, en primer lugar se define la etiqueta `<comportamiento>` que únicamente tiene el atributo `nom`. Actualmente solo hay una clase disponible que implemente este componente y, por tanto, el atributo toma el valor *ComponenteComportamiento*. Esto es debido a que no ha sido necesario implementar más variedad de clases.

Después definimos los dos subcomponentes (o uno, según nos convenga). Nos centraremos primero en la definición del comportamiento móvil:

a. Definición del subcomponente Comportamiento Móvil

Dicho subcomponente se define a través de la etiqueta `<compmovil>`, que contiene los siguientes atributos:

- `nom`: Nombre de la clase utilizada para implementar el subcomponente. Por el momento existen dos:
    - *MovimientoArtificial*, la cual implementa una inteligencia artificial, simple pero altamente configurable.

      El funcionamiento es el siguiente: se crea un vector llamado vector de acciones donde, en cada posición, introducimos una acción y una duración para esta acción. A la hora de ejecutar la IA lo que se hace es ejecutar cada una de las acciones durante el tiempo indicado como duración de cada acción, en orden de aparición en el vector. Y cuando se ha ejecutado la última acción del vector volvemos a ejecutar la primera acción de este (generando un bucle).

      Es a través del XML que introducimos el vector de acciones, lo que da lugar a muchas combinaciones de acciones posibles.

    - *MovimientoUsuario,* la cual implementa unos controles para que el usuario dirija al objeto definido. Por el momento solo hay un tipo de control, así que es aconsejable que solo un objeto en cada nivel emplee este comportamiento y que sea el objeto que asume el rol de personaje principal. Se puede encontrar más información sobre los controles implementados para el movimiento en el anexo I.

- `movporvel`: El valor de este atributo debe ser `true` o `false`. En caso de valer `false` especificará que el objeto se mueve aplicando una fuerza constante, lo que provocará un efecto de aceleración y de frenada realista (pues la velocidad va aumentando o descendiendo poco a poco). Si, por el contrario, toma el valor `true` especificará que el objeto se mueve aplicando una velocidad constante, lo que se traduce en que el objeto no mostrará efectos de frenada y aceleración (útil en según qué casos).

- `fuerzalateral1`, `fuerzalateral2`, `fuerzaarriba1`, `fuerzaarriba2`, `fuerzaabajo1`, `fuerzaabajo2`: Estos seis atributos indican la fuerza con la que se moverá el objeto cuando lleve a cabo un movimiento, pero solo en el caso en que este deshabilitado el atributo `movporvel`.

  Distinguimos entre tres direcciones sobre las cuales definimos fuerzas diferentes e independientes: la dirección lateral (incluye tanto el movimiento hacia la derecha como el movimiento hacia la izquierda), la dirección hacia arriba y la dirección hacia abajo.

  Además, para cada una de las direcciones tenemos dos niveles de fuerza. Esto nos va a permitir: combinar hasta dos velocidades cuando configuremos el movimiento artificial; o, en caso de utilizar *MovimientoUsuario*, que el usuario pueda utilizar dos niveles de velocidad, consiguiendo así un control más fiel a las intenciones del usuario.

  Llamamos nivel 1 al nivel donde se debería asignar la fuerza menor y nivel 2 al nivel donde se debería asignar la fuerza mayor.

  Por último, las unidades de la fuerza son Néwtones y, como ya se ha explicado anteriormente, la fuerza se aplicará de forma constante durante todo el rato que dure el movimiento.

- `limitelateral1`, `limitelateral2`, `limitearriba1`, `limitearriba2`, `limiteabajo1`, `limiteabajo2`: Este atributo tendrá dos significados diferentes según el valor que tome el atributo `movporvel` :

  - Si `movporvel` vale *`true`*: A través de estos atributos indicamos las velocidades constantes que se aplicarán para cada dirección y cada nivel de movimiento. Pues es el caso en que nos movemos aplicando una velocidad constante.

  - Si, por el contrario, `movporvel` vale *`false`*: A través de estos atributos indicamos las velocidades límites, para las diferentes direcciones y los diferentes niveles de movimiento. Es decir, las velocidades que podremos alcanzar, como máximo, mediante la aplicación de la fuerza constante introducida en los atributos anteriores. Una vez alcanzada la velocidad límite para la dirección y el nivel que se esté aplicando el objeto se mantendrá a esta velocidad en lugar de seguir aumentándola.

  Estos atributos se introducen en metros por segundo.

Componente Móvil implementado mediante MovimientoArtificial

Una vez introducidos los atributos comunes a todo comportamiento móvil, en caso de haber especificado la clase *MovimientoArtificial* como clase que implementa al subcomponente, se deberá definir la inteligencia artificial a utilizar.

Es entre la apertura y el cierre de la etiqueta `<compmovil>` donde introduciremos la definición de esta IA. Vamos a ver, por tanto, un ejemplo de definición de una inteligencia artificial para el tipo de subcomponente indicado:

```
<defiasimple numposiciones="2" esinfluenciable="false">

      <defposicion Position="0" accion="izquierda" duracion="70"
      nivelenfasis="2">

      </defposicion>

      <defposicion Position="1" accion="derecha" duracion="70"
      nivelenfasis="2">

      </defposicion>

</defiasimple>
```

Ejemplo 8.8: Definición de la inteligencia artificial del subcomponente Comportamiento Móvil cuando este se implementa mediante la clase *MovimientoArtifical*.

Como podemos observar toda la IA va dentro de la etiqueta `<defiasimple>`. Esta etiqueta contiene los atributos:

- `numposiciones`: Atributo que contendrá el número de acciones que van a constituir la inteligencia artificial.
- Y `esinfluenciable`: Un atributo que, en caso de valer *`true`*, indica que un objeto externo puede venir y exigir que nos movamos hacia una dirección concreta y, en ese caso, nosotros tendremos que hacer lo que nos pide. Este atributo deberá valer *`true`* si, por ejemplo, estamos definiendo un objeto que tiene el rol de proyectil y que según la dirección de disparo que especifique el objeto emisor tendrá que moverse hacia un lado o hacia otro.

  En caso de tomar el valer *`false`*, un objeto externo no podrá influir en la dirección a la que nos movemos.

Cuando ya hemos definido los parámetros básicos es hora de definir el vector de acciones. Definimos tantas etiquetas `<defposicion>` como posiciones tendrá el vector de acciones (número que ya hemos especificado en `numposiciones`). Como se puede entrever, cada etiqueta representará una acción del vector de acciones y por ello deberá llevar los siguientes parámetros:

- `Position`: Contendrá la posición del vector de acciones que estamos definiendo (entre 0 y `numposiciones` - 1).

- `accion`: Nombre de la acción que se encuentra en la posición del vector especificada a través del atributo anterior. Como estamos definiendo el comportamiento móvil, la acción puede ser cualquiera de las ocho direcciones de movimiento que el sistema interpreta: *`izquierda`*, *`arribaizq`*, *`abajoizq`*, *`derecha`*, *`arribader`*, *`abajoder`*, *`arriba`*, *`abajo`* o *`estadobase`*.

  Donde *`estadobase`* es la acción de mantenerse inmóvil.

- `duracion`: Número de frames que se estará aplicando esta acción.

- `nivelenfasis`: Atributo que indica el nivel de énfasis con que se aplica la acción de movimiento. Puede tomar el valor *`1`* o *`2`* y, como ya se ha explicado anteriormente, repercutirá sobre la fuerza o velocidad aplicada, pues, según el nivel escogido, se utilizará un u otro atributo de entre los definidos en la etiqueta `<compmovil>`. A mayor nivel mayor fuerza o velocidad.

  En todo momento se está hablando de fuerza o velocidad, sin especificar sobre cuál de los dos es el afectado, porque es el otro atributo de `<compmovil>`, llamado `movporvel`, el que decide si el objeto se mueve mediante la aplicación de una fuerza constante o mediante la aplicación de una velocidad constante.

Por último, hay que tener en cuenta que solo tiene sentido la definición de un comportamiento móvil si el objeto que estamos definiendo tiene Componente Físico. Ya que si el objeto no tiene representante en el mundo físico, las ordenes de movimiento no podrán llevarse a cabo, pues no hay ningún cuerpo físico al que aplicarle una fuerza o una velocidad.

b. Definición del subcomponente Comportamiento de Habilidad

Ahora que ya hemos visto como definir el movimiento que llevará a cabo el objeto, es hora de definir el comportamiento de habilidad. ¿Qué habilidades llevará a cabo el objeto que estamos definiendo y cuando las ejecutará?

Pero antes hay que dejar claro que solo tiene sentido la definición de este subcomponente, si el objeto que estamos definiendo tiene Componente de Habilidad, porque, en caso contrario, las habilidades que el comportamiento de habilidad pretenda utilizar no podrán ejecutarse.

Como podemos ver en el ejemplo 8.7, es la etiqueta `<comphabilidad>` la que define el comportamiento ante el manejo de habilidades. Los atributos que debemos especificar en esta etiqueta son los siguientes:

- `nom`: En este caso las subclases disponibles son *HabilidadesArtificiales* y *HabilidadesUsuario*.
    - *HabilidadesArtificiales* implica la definición de una simple pero personalizable inteligencia artificial que dirija las habilidades del objeto. El mecanismo es idéntico al que hemos visto en *MovimientoArtificial* salvo por el hecho de que, en este caso, las acciones a introducir dentro del vector de acciones están relacionadas con el uso de las habilidades.
    - En cuanto a *HabilidadesUsuario*, como ocurría con *MovimientoUsuario*, su uso está reservado para el personaje principal, ya que es el usuario el que manejará las habilidades de todos los objetos que implementen este tipo de comportamiento de habilidad.

      En el anexo I se dan más detalles acerca del control implementado para controlar la ejecución de habilidades.
- `disparocambiable`: Atributo que, en caso de valer *`true`*, obliga a que el disparo salga emitido en la trayectoria en la que el objeto se está moviendo en un momento dado, o en la trayectoria en la que se ha movido en último lugar.

  Aun así, hay que tener en cuenta que existen versiones alternativas de la habilidad disparo que se limitan a un solo eje, como puede ser un *disparo lateral*. En este caso cogeríamos la última dirección lateral tomada, en lugar de la última dirección real que puede haber sido una que implique otro eje el cual no nos interesa en este momento.

  Si el atributo toma el valor *`false`*, en cambio, el disparo irá siempre en la dirección por defecto, definida en los atributos que se explicarán a continuación.

- `direcciondisplat`, `direcciondispvert`, `direcciondisp`: Se trata de los atributos que contendrán las direcciones en las que se desplazarán por defecto los tres tipos de disparo existentes.

  Distinguimos entre *disparo lateral* y *disparo vertical* si nos conviene que el disparo lleve una trayectoria en uno solo de los ejes. En cambio *disparo*, a secas, será la acción que permitirá que el disparo se emitido se mueva en cualquiera de las ocho direcciones posibles.

  Es importante tener en cuenta que el valor por defecto puede pasar a ser un valor persistente que dure toda la vida del objeto que estamos definiendo si `disparocambiable` vale *`false`*.

  Los valores posibles para `direcciondisplat` son: "*`izquierda`*" o "*`derecha`*". Los valores posibles para `direcciondispvert` son: "*`arriba`*" o "*`abajo`*". Y, por último, los valores posibles para `direcciondisp` son: *`izquierda`*, *`derecha`*, *`arriba`*, *`abajo`*, *`arribaizq`*, *`abajoizq`*, *`arribader`* o *`abajoder`*.

Componente de Habilidad implementado mediante HabilidadesArtificiales

Una vez introducidos los atributos que la etiqueta`<comphabilidad>` requiere, nos tenemos que preguntar si la subclase que hemos escogido para que implemente nuestro comportamiento de habilidad requiere información adicional específica o no. La respuesta solo será afirmativa si hemos decidido utilizar la subclase *HabilidadesArtificiales*, pues en este caso, debemos añadir la inteligencia artificial concreta que vamos a utilizar, entre la apertura y cierre de la etiqueta `<comphabilidad>`:

```
<defiasimple numposiciones="1">
      <defposicion Position="0" accion="displateral" duracion="65">
      </defposicion>
</defiasimple>
```

Ejemplo 8.9: Definición de la inteligencia artificial del subcomponente Comportamiento de Habilidad cuando este se implementa mediante la clase *HabilidadesArtificiales*.

De la misma forma que ocurría con la IA de *MovimientoArtificial*, tenemos que definir la etiqueta `<defiasimple>`, aunque en este caso el único atributo que esta incluirá será `numposiciones`. Como antes, este atributo indicará el número de posiciones que tendrá el vector de acciones que vamos a definir.

Seguidamente, dentro de la etiqueta `<defiasimple>` debemos definir una etiqueta `<defposicion>` por cada acción del vector de acciones.

Los atributos `Position` y `duración` de esta etiqueta vuelven a tener el mismo significado que tenían en la IA de *MovimientoArtificial*, por este motivo no es necesario volver a explicarlos. No ocurre lo mismo con el atributo `accion`, que aunque sigue siendo el atributo que contiene la acción asociada con la posición del vector que se esté definiendo, en esta ocasión tiene unos valores diferentes, pues las acciones de movimiento y de habilidad no son las mismas. Por tanto, en este caso, el atributo `accion` puede tomar los siguientes valores: *`displateral`*, *`dispvertical`*, *`disparo`* y *`espera`*. Se trata de los tres tipos de disparo ya vistos con antelación y una acción llamada *`espera`* que permite definir una posición del vector en la que no se lleva a cabo la ejecución de ninguna habilidad y simplemente se espera a que transcurran los frames de duración indicados.

Podemos ver como la habilidad de *arrastrado* no se puede utilizar en esta IA, pues no se admiten acciones que indiquen la ejecución de esta habilidad. Esto es debido a que no era necesaria para los enemigos implementados en el juego. Si queremos utilizarla se debería definir una subclase nueva con una IA más completa, que permitiera su uso.

Por último, hay que tener en cuenta que, a diferencia de lo que ocurría con la IA de *MovimientoArtificial*, en esta IA en el momento de llevar a cabo la acción de una posición del vector de acciones, esta acción se ejecuta una vez y el resto del tiempo hasta completar la duración de la acción es una simple espera. En MovimientoArtificial, en cambio, el movimiento indicado como acción se llevaba a cabo durante todos los *frames* que dicha acción ocupaba.

Definición del Componente Musical

Ahora vamos a describir cómo podemos definir el componente que gestiona los sonidos de un objeto. Para ello presentamos el siguiente ejemplo:

```
<musical nom="ComponenteMusical">
	<sonido path="explosion.ogg" tipo="0">
		<estadoactivacion nom="muriendo"/>
	</sonido>
	<sonido path="movimientocanon.ogg" tipo="-1">
		<estadoactivacion nom="derecha"/>
		<estadodesactivacion nom="izquierda"/>
	</sonido>
	<sonido path="movimientocanon.ogg" tipo="-1">
		<estadoactivacion nom="izquierda"/>
		<estadodesactivacion nom="derecha"/>
	</sonido>
	<sonido path="disparo.ogg" tipo="0">
		<estadoactivacion nom="disparo"/>
	</sonido>
</musical>
```

Ejemplo 3.10: Definición de un componente musical para un tipo de objeto.

El mecanismo básico de este componente es el siguiente:

La aplicación tiene una serie de estados predefinidos, los cuales se identifican por una palabra. Estos estados se activan y se envían a los componentes *visual* y *musical* cuando se dan las condiciones apropiadas. En la definición del Componente Musical, lo que hacemos es asociar sonidos a cada uno de estos estados, para que cuando los estados se den, los sonidos asociados a estos comiencen a reproducirse o cesen su reproducción.

Cabe decir que también se podría ampliar el abanico de estados disponibles a los que podemos asociar sonidos. Pero para ello debería tocarse el código java, pues los estados se activan internamente.

Una vez explicado el funcionamiento del componente musical es hora de que veamos el formato a seguir para definir este componente. Como vemos, es la etiqueta `<musical>` la que marca el inicio de la definición del componente musical. Esta etiqueta únicamente requiere el atributo `nom`, que en este caso, siempre tomara el valor *ComponenteMusical*, pues no hay más que una implementación del componente.

Dentro de la etiqueta `<musical>`, para cada sonido tenemos una etiqueta `<sonido>`. Esta etiqueta requiere dos atributos:

- `path`: Donde se indica el nombre relativo del fichero de sonido. Es necesario incluir la extensión del fichero en el nombre, ya que se admiten varios tipos de formatos (todos los que admite Android). El nombre indicado para el fichero es relativo al directorio "`/assets/sounds/`", ya que es en este directorio donde deben ubicarse todos los sonidos. Por tanto, el nombre absoluto del fichero de sonido será "`/assets/sounds/<nombre_con_extension>`".

- `tipo`: Entero que indica el tipo al que pertenece el sonido que estamos definiendo, siendo los tipos disponibles los que se presentan en la siguiente tabla:

Numero Identificador	Nombre del tipo	Descripción
0	Normal	Se trata de un tipo de sonido que se reproduce una sola vez de principio a fin.
-1	Cíclico	Se trata de un tipo de sonido que se reproduce cíclicamente de principio a fin. Cuando llegamos al fin del sonido volvemos a reproducirlo desde el inicio y así hasta que se indique el cese de su reproducción.

Figura 8.2: Tabla con los tipos de sonido que pueden declararse dentro de un componente musical.

Una vez definida la etiqueta `<sonido>`, entre su apertura y cierre deberá aparecer una etiqueta `<estadoactivacion>` por cada estado que vaya a activar dicho sonido; y, en caso de tratarse de un sonido cíclico, una etiqueta `<estadodesactivacion>` por cada estado que vaya a desactivar la reproducción del sonido.

Un sonido normal no puede tener estados de desactivación puesto que siempre se reproduce de principio a fin. En cambio, todo sonido cíclico debe tener al menos un estado de desactivación, a menos que queramos que se reproduzca cíclicamente hasta que el objeto muera, momento en el cual todos sus sonidos se pausan y se eliminan.

Las etiquetas `<estadoactivacion>` y `<estadodesactivacion>` tienen un único atributo llamado `nom`, se trata del nombre del estado al que representan. Como ya se ha explicado en este mismo apartado, los estados se activan cuando se dan las condiciones que definen al propio estado y se envían tanto al *Componente Musical* como al *Componente Visual*. Los estados que ambos componentes pueden recibir durante la ejecución del nivel y, por tanto, los valores que este atributo puede tomar, son los siguientes:

Nombre estado visual y musical	**Momento de activación**
izquierda	Se activa cuando el objeto comienza a desplazarse hacia la izquierda.
derecha	Se activa cuando el objeto comienza a desplazarse hacia la derecha.
arriba	Se activa cuando el objeto comienza a desplazarse hacia arriba.
abajo	Se activa cuando el objeto comienza a desplazarse hacia abajo.
estadobase	Se activa cuando el objeto se mantiene inmóvil y no se está desplazando hacia ninguna dirección.
arribaizp	Se activa cuando el objeto comienza a desplazarse diagonalmente hacia arriba y a la izquierda.
abajoizp	Se activa cuando el objeto comienza a desplazarse diagonalmente hacia abajo y a la izquierda.
arribader	Se activa cuando el objeto comienza a desplazarse diagonalmente hacia arriba y a la derecha.
abajoder	Se activa cuando el objeto comienza a desplazarse diagonalmente hacia abajo y a la derecha.
disparo	Se activa cuando se activa la habilidad disparo.
arrastrar	Se activa cuando se ha dado una nueva orden sobre la habilidad arrastrar, ya sea agarrar en un punto concreto o arrastrar hacia un punto concreto lo que ya tenemos agarrado.
soltar	Se activa cuando se desactiva la habilidad arrastrar.

estadopulsado	Algunas lógicas lo activan cuando están un paso más cerca de la satisfacción. Por ejemplo: *PuertaNivel* cuando la puerta se abre.
estadonopulsado	Algunas lógicas lo activan cuando están un paso más lejos de la satisfacción. Por ejemplo: *PuertaNivel* cuando la puerta se cierra.
satisfecho	Se activa cuando la lógica pasa a estar satisfecha.
debilitado	Se activa cuando el objeto es golpeado y se le resta una vida pero no se queda con 0 vidas.
muriendo	Se activa cuando el objeto es golpeado y se le resta una vida, quedándose con 0, con lo cual va a morir inminentemente.

Figura 8.3: Tabla con los estados que tanto el *Componente Visual* como el *Componente Musical* recibirán cuando se den las condiciones de activación pertinentes.

Por último, hay que tener en cuenta que cada estado podrá tener como máximo un sonido asociado. De esta forma evitamos tener excesivos sonidos que, lejos de dar mejor ambientación al juego y facilitar que el usuario entienda qué está ocurriendo en el transcurso del nivel provocaría el efecto contrario.

Definición del Componente Visual

Se trata del componente que gestiona la parte visual de un objeto, es decir, la parte que vemos por pantalla. Es uno de los más importantes y, por tanto, uno de los más complejos de definir.

Tal y como se ha hecho con el resto de componentes, a continuación se presenta un ejemplo de la definición de un *Componente Visual*:

```
<pintable nom="GestorDeSprites" anchura="64" altura="64" angulo="0"
            anchuratext="0.125" alturatext="0.25">
</pintable>
```

Ejemplo 8.11: Definición de un Componente Visual para un tipo de objeto.

En esta ocasión, es la etiqueta `<pintable>` la que marca el inicio de la definición del componente en cuestión. Los atributos que esta etiqueta requiere son los siguientes:

- `nom`: En este caso existen varias alternativas para implementar el *Componente Visual*, se elegirá una u otra según el cometido que queramos que lleve a cabo el componente que se está definiendo:
    - En primer lugar tenemos la superclase *ComponentePintable*. Esta será la opción escogida cuando tengamos un objeto que visualmente sea estático, es decir, que siempre presente la misma textura sin alteraciones. Eso sí, si el objeto se mueve, siempre puede haber un desplazamiento o giro de la textura, según la posición y ángulo que ocupe el objeto en cuestión.
    - En según lugar tenemos la opción que probablemente se utilice en la mayoría de los casos, se trata de la subclase *GestionDeSprites*. Esta subclase gestiona animaciones que serán reproducidas cuando recibamos los estados a los que dichas animaciones se han asociado.
    - En último lugar, tenemos la subclase *MatrizDeTiles*. Esta clase será escogida cuando queramos presentar una matriz donde cada posición sea una textura independiente. Útil para pintar puzles e imprescindible para pintar la matriz de tiles que representará el escenario del nivel.
- `anchura` y `altura`: A través de estos atributos indicamos la anchura y la altura del área dentro de la cual se pintará la textura, que siempre ocupará la totalidad de esta área. Es importante tener claro que estas dimensiones son independientes de las dimensiones de la física del objeto, siendo el área visual del objeto independiente del área física y al revés.
- `angulo`: Atributo que especifica el ángulo base del *Componente Visual*. El componente se pintara desde el inicio del nivel con este ángulo de rotación. Tal y como ocurre con las dimensiones, el ángulo del Componente Físico y el ángulo del Componente Visual son independientes.

  Dicho ángulo podrá cambiar si el objeto se mueve y además puede rotar (propiedad configurable desde el Componente Físico), en caso contrario, el Componente Visual permanecerá permanentemente en el ángulo indicado.

- `anchuratext` y `alturatext`: Se trata de las dimensiones de cada una de las texturas dentro del fichero de texturas de donde se extraerán. Como se pueden utilizar varios ficheros de texturas para el pintado de un *Componente Visual*, todos deberán compartir las dimensiones indicadas a través de estos atributos.

  El fichero de texturas debe interpretarse como una matriz de posiciones donde cada posición es una textura diferente y todas las posiciones tienen el mismo tamaño. Además, se exige que estas dimensiones se den en forma de fracción respecto a la totalidad del fichero de texturas. Por ejemplo, en un fichero donde hay cuatro texturas, los valores para los dos atributos serán: `anchuratext="0.25"` y `alturatext="0.25"`.

Independientemente de la clase que escojamos para implementar el componente, entre la apertura y el cierre de la etiqueta `<pintable>` deberá declararse la etiqueta `<recursos>`. Seguidamente vemos un ejemplo de esta declaración:

```
<recursos filatext="0" columnatext="0">
<recurso path="ppChar.png" />
</recursos>
```

Ejemplo 8.12: Definición de los recursos que un *Componente Visual* utilizará para pintar al objeto que representa.

Como podemos ver, la etiqueta `<recursos>` tiene dos atributos: `filatext` y `columnatext`. Estos contienen la fila y la columna que identifican a la posición del fichero de texturas declarado en primer lugar, donde se encuentra la textura que se pintará por defecto en el área visual que el objeto ocupe dentro del nivel. Para saber qué fila y qué columna le corresponde a cada textura del primer fichero, debemos ver el fichero de texturas como si fuera una matriz; en la Figura 8.4 se muestra un ejemplo gráfico.

Según la clase que implemente al componente estos dos atributos tendrán más importancia o menos. En caso de utilizar la clase *ComponentePintable* la textura por defecto será la que se vaya a pintar durante toda la ejecución del nivel.

Una vez definida la etiqueta `<recursos>`, dentro de esta debe introducirse una etiqueta `<recurso>` por cada uno de los ficheros de texturas que vayamos a utilizar durante el transcurso del nivel para pintar el objeto que estamos definiendo.

Es obligatoria la declaración de al menos un fichero de texturas, pues todo *Componente Visual* hará uso de al menos una textura. Además, como ya se ha explicado, el fichero de texturas que declaremos en primer lugar será el fichero del que se extraiga la textura indicada por defecto.

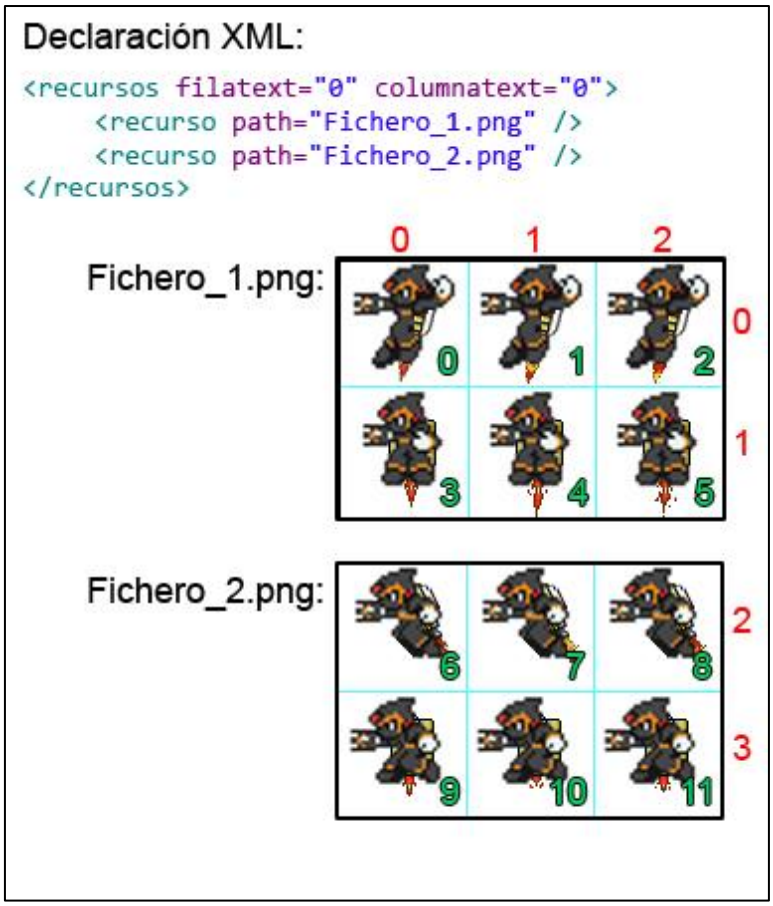


Figura 8.4: Esquema que muestra cómo identificar dentro del fichero XML a las diferentes texturas de los diferentes ficheros de texturas declarados. Se presentan dos formas de identificar a cada textura y cada atributo requerirá una u otra: En rojo el método que nos permite identificar a una textura a través de una fila y una columna; y en verde el método que nos permite identificar a una textura a través de un único entero.

Una vez definida la parte común a cualquier Componente Visual, vamos a ver cuáles son las etiquetas que debemos añadir entre la apertura y el cierre de la etiqueta `<pintable>` en caso de que escojamos una u otra clase para implementar el componente en cuestión (es decir, según el valor del atributo `nom` de la etiqueta `<pintable>`):

En primer lugar, si escogemos la superclase *ComponentePintable* no hace falta información adicional.

a. Componente Visual implementado mediante MatrizDeTiles

Si, en cambio, escogemos la subclase *MatrizDeTiles* tendremos que definir la matriz a pintar. El siguiente ejemplo ayudará a comprender cómo se debe realizar dicha definición:

```
<matriz  numfilas="2" numcolums="2" tilesanchfichero="8"
tilesaltofichero="8">

<row Position="0">

      <column Position="0" Value="0"></column>

      <column Position="1" Value="0"></column>

</row>

<row Position="1">

      <column Position="0" Value="0"></column>

      <column Position="1" Value="-1"></column>

</row>

</matriz>
```

Ejemplo 8.13: Definición de la matriz de tiles que todo *Componente Visual* implementado mediante la clase *MatrizDeTiles* debe definir.

La matriz del *Componente Visual* se define de forma parecida a la matriz específica del Componente Lógico que se implementaba mediante la clase *PuzzleDePiezas*. Seguidamente vamos a explicar detalladamente que formato tiene esta definición.

Introducimos la etiqueta `<matriz>` dentro de la etiqueta `<pintable>` y definimos los siguientes atributos:

Primero debemos introducir los dos atributos, `numfilas` y `numcolums`, que respectivamente especifican el número de filas y de columnas que tendrá la matriz de tiles.

Seguidamente introducimos los atributos `tilesanchfichero` y `tilesaltofichero`. Estos atributos nos proporcionan una información redundante pero útil, se trata del número de columnas y de filas en que se divide el fichero de texturas que emplearemos para pintar la matriz. Como ya hemos visto anteriormente, un fichero de texturas se debe ver como una matriz en la que cada posición contiene una textura.

Decimos que es información redundante porque a través de la altura y anchura de cada textura (atributos que ya se han introducido con anterioridad) podemos extraer también el número de texturas que hay a lo largo y a lo ancho del fichero. Pero aun así es útil de cara a la persona que define la matriz, ya que así mantendrá la imagen mental del fichero de texturas como una matriz de filas y columnas y le será más fácil asociar, a continuación, cada tile de la matriz de tiles con una textura del fichero de texturas.

A la hora de realizar la asociación citada, entre la matriz de tiles y la matriz del fichero de texturas, nos referimos a las texturas del fichero de texturas mediante enteros que identifican la posición donde cada una de estas texturas se encuentra. La posición 0 de dicho fichero corresponderá con la textura que este en la posición que se encuentra en la esquina superior-izquierda del fichero y la textura M (siendo M el número de texturas que se encuentran en el fichero restándole una unidad, por tanto: `tilesanchfichero` * `tilesaltofichero` - 1) será la que se encuentre en la esquina inferior-derecha del fichero de texturas. Para ver más claramente como se identifican las diferentes texturas de un fichero a través de un único entero ir a la Figura 8.4.

Una vez definida la etiqueta `<matriz>`, introduciremos la etiqueta `<row>` con el atributo `Position`, que especificará el número de fila que estamos definiendo, para cada fila de la matriz.

Y dentro de cada fila, para cada columna introduciremos la etiqueta `<column>` con dos atributos: el atributo `Position`, con la posición que le corresponde a esta columna; y el atributo `Value`, con el número de la textura que deberá ser introducida en la posición de la matriz de tiles que estamos definiendo (será la posición identificada por la columna que se está definiendo y la fila dentro de la cual estamos definiendo esta columna).

Si queremos que una posición de la matriz de tiles permanezca vacía y no se le asocie ninguna textura, el valor `-1` está reservado para este cometido.

Para ver con más detalle cómo se declara una matriz es aconsejable que también se repase la explicación introducida dentro del apartado que habla del Componente Lógico, en concreto la parte en que se indica cómo definir la parte especifica de un *PuzzleDePiezas*.

Una vez visto cómo se definen los atributos específicos de un *Componente Visual* que se implementa mediante la clase *MatrizDeTile, e*s importante hacer una reflexión acerca de este tipo de componente. Y es que en todo nivel del videojuego *Terrestrial Blast!,* el objeto que asume el rol de `escenario` es obligatorio que defina un Componente Visual implementado por esta clase, pues en este videojuego los escenarios se definen mediante la técnica de la matriz de tiles.

Además, el objeto *escenario* no debe implementar un componente físico, pues es el sistema el que al avistar el rol en cuestión recoge la matriz de tiles y crea las físicas del escenario de la mejor forma posible. Evitando así la existencia de un gran número de elementos físicos en el mundo físico junto con algunas limitaciones que tiene la librería de físicas Box2D. Para más información leer el apartado acerca de Box2D.

Pero no acaban aquí las peculiaridades de este tipo de Componente Visual, pues existen tres restricciones que MatrizDeTiles impone sobre las propiedades de todo objeto que la implementa:

- La primera restricción es que el objeto no puede rotar (debe tener una física con rotación fija y el ángulo visual base debe ser 0). Esta decisión se lleva a cabo por dos motivos: en primer lugar, rotar una matriz no ha resultado necesario, debido a los usos que se les da a las matrices del videojuego, y, en segundo lugar utilizar la rotación reduciría la velocidad de procesado del videojuego drásticamente.
- La segunda restricción impone que tan solo podemos utilizar un fichero de texturas para pintar toda la matriz, pues, a la hora de pintar, OpenGL no permite utilizar más de un fichero de texturas a la vez y, por motivos de eficiencia, toda la matriz se pinta a través de una sola instrucción OpenGL.
- La tercera y última restricción es que la matriz no puede desplazarse, debe ser estática y no tener comportamiento móvil. El motivo de esta restricción es que no es necesario que las matrices empleadas en los niveles del videojuego se puedan mover, y al igual que la rotación, una translación implica un tiempo excesivo que reduciría la velocidad de procesado del videojuego.

Por último, quizás sea necesario especificar que, en caso de utilizar *MatrizDeTiles*, el punto central donde se encontrará el objeto a la hora de cargar el nivel, introducido en la etiqueta `<objeto>`, será el centro de la matriz.

Componente Visual implementado mediante GestorDeSprites

Vamos a ver ahora cuáles son los atributos que debemos definir en caso de utilizar la implementación para el Componente Visual que ofrece la clase *GestorDeSprites*. Como ya se ha avanzado antes, este tipo está reservado para objetos que quieran presentar un aspecto visual dinámico, que varíe con el tiempo, ya que esto no se puede hacer ni con *ComponentePintable* ni con *MatrizDeTiles*.

La clase *GestorDeSprites* nos permite crear animaciones a través de vectores de texturas. Las texturas que conforman un vector se reproducen secuencialmente generando la sensación de movimiento.

Estos vectores de texturas, también llamados movimientos, activarán o cesarán su reproducción cuando se reciban los estados visuales asociados como órdenes de cese

o reproducción del movimiento en cuestión. Para saber más acerca de qué son los estados visuales y qué papel llevan a cabo, es aconsejable leer el apartado *Componente Musical* de este mismo anexo.

A continuación se presenta un ejemplo de definición de los movimientos que conforman un *GestorDeSprites*:

```
<movimientos framesxsprite="5" invertido="false">
<mov tipo="-1" estado="estadobase">
      <sprite ValueRow="1" ValueColumn="3"/>
      <sprite ValueRow="1" ValueColumn="4"/>
      <sprite ValueRow="1" ValueColumn="5"/>
</mov>
<mov tipo="-1" estado="estadobaseinvertido">
      <sprite ValueRow="1" ValueColumn="0"/>
      <sprite ValueRow="1" ValueColumn="1"/>
      <sprite ValueRow="1" ValueColumn="2"/>
</mov>
<mov tipo="-1" estado="derecha">
      <sprite ValueRow="2" ValueColumn="3"/>
      <sprite ValueRow="2" ValueColumn="4"/>
      <sprite ValueRow="2" ValueColumn="5"/>
</mov>
<mov tipo="0" estado="muriendo">
      <sprite ValueRow="0" ValueColumn="6"/>
      <sprite ValueRow="1" ValueColumn="6"/>
      <sprite ValueRow="2" ValueColumn="6"/>
      <sprite ValueRow="3" ValueColumn="6"/>
</mov>
</movimientos>
```

Ejemplo 8.14: Definición de los movimientos que conforman un Gestor de Sprites. La definición de estos es obligatoria solo cuando el Componente Visual se implementa mediante la clase *GestorDeSprites*.

Como vemos, las diferentes animaciones que componen el *Gestor de Sprites* se declaran en el interior de la etiqueta `<movimientos>`, que a su vez se encuentra en el interior de la etiqueta `<pintable>`.

Dicha etiqueta tiene dos atributos:

- Una animación no es más que un conjunto de imágenes visualizadas una detrás de otra, generando así la sensación de movimiento. A través del atributo `framesxsprite` indicamos el tiempo, en frames, que se mantendrá cada una de estas imágenes por pantalla.

  El nombre del atributo se debe a que en el ámbito de los videojuegos, estas imágenes que conforman una animación se pueden llamar sprites.

  Por último, explicar que se utiliza un número de frames por sprite común para todas las animaciones de un *Gestor de Sprites* puesto que así se exige un nivel de detallismo similar en todas las animaciones de un mismo objeto. Y, de paso, simplificamos la definición de las animaciones, no teniendo que especificar el tiempo individualmente para cada animación.

- Por otra parte, el atributo `invertido` indicará lo siguiente:

  - En caso de valer *`true`*: Al iniciar la ejecución del nivel se presentará al objeto como si su último movimiento hubiese sido hacia la izquierda o hacia una de las dos diagonales que se encuentran hacia la izquierda.

  - En caso de valer *`false`*: Al iniciar la ejecución del nivel se presentará al objeto como si su último movimiento hubiese sido hacia la derecha o hacia alguna de las diagonales que se encuentran a la derecha.

Vamos a ver a continuación, porque es importante saber en todo momento si el último movimiento de un objeto ha sido (o está siendo actualmente) hacia la derecha o hacia la izquierda.

El componente Visual, además de los estados normales y corrientes, comunes al componente musical y Visual, interpreta una versión alternativa para algunos de estos estados, se trata de su versión “invertida”. La necesidad de tener una versión invertida de algunos estados radica en el siguiente hecho: como un objeto se pueda estar moviendo hacia la izquierda o hacia la derecha, cuando este objeto alcance uno de los estados que necesita versión invertida, dependiendo de hacia donde haya sido su último movimiento, estará “mirando” hacia la derecha o hacia la izquierda y esto implica que la animación de dicho estado deberá mostrar al objeto mirando hacia esa misma dirección.

Un ejemplo muy sencillo que permite ilustrar la explicación es el siguiente: si queremos asociar una animación al estado *disparo* querremos que cuando el personaje este mirando hacia la derecha, la animación presente al objeto disparando en esa dirección; y, por el contrario, cuando este mismo objeto este mirando hacia la izquierda querremos que la animación presenta al objeto disparando en esta otra dirección (ver Figura 8.5).

Figura 8.5: Ejemplo de uso de los estados invertidos. En ambas columnas, el personaje se encuentra en reposo. La diferencia radica en el último movimiento que hizo y que le ha condicionado estar ahora en la versión invertida o no invertida de *estadobase*, según el caso.

A continuación se exponen los estados que provocarán un cambio en el valor del atributo `invertido` durante la ejecución del nivel, cambiando de esta forma el lado hacia el que estará "mirando" el objeto en cuestión (pues el valor asignado durante la definición del objeto solo es el valor por defecto):

- `invertido` tendrá el valor *true*: En caso de que el último movimiento llevado a cabo, o el movimiento que se esté llevando a cabo en un momento dado, sea un movimiento hacia las direcciones representadas mediante los estados: *izquierda*, *arribaizq* o *abajoizq*.

- `invertido` tomará el valor *false*: En caso de que el último movimiento o el movimiento que el objeto esté llevando a cabo en un momento dado, sea un movimiento hacia una de las direcciones que se representan mediante los estados: *derecha*, *arribader* o *abajoder*.

Vamos a presentar ahora los estados visuales invertidos que un *GestorDeSprites* puede interpretar. Son versiones invertidas de los estados visuales y musicales definidos en la tabla de la Figura 8.3 (recordemos que estos eran estados que recibe, tanto un Componente Musical, como un Componente Visual). Para identificar el estado original con el que se asocia cada estado invertido hay que tener en cuenta que el nombre de los estados invertidos está compuesto de la siguiente forma: "<nombre_estado_original>*invertido*".

Nombre estado visual invertido	
estadobaseinvertido	*estadopulsadoinvertido*
arribainvertido	*estadonopulsadoinvertido*
abajoinvertido	*satisfechoinvertido*
disparoinvertido	*debilitadoinvertido*
arrastrarinvertido	*muriendoinvertido*
soltarinvertido	

Figura 8.6: Tabla con la versión invertida de algunos de los estados musicales y visuales de la tabla 2. Se trata de unos estados que solo interpreta el *Componente Visual* cuando se implementa mediante *GestordeSprites*.

Ya hemos visto cómo funciona la variable `invertido`, a que se debe su existencia y los nuevos estados invertidos que *GestorDeSprite* interpreta, pero ¿Cómo funciona exactamente el mecanismo?

Pues bien, como ya se ha explicado anteriormente, el mecanismo de *GestorDeSprite* se basa en asociar animaciones a estados visuales concretos, de forma que cuando el Componente Visual recibe un estado (cualquiera de los presentados en la tabla de la Figura 8.3) se busca una animación que dé respuesta a este estado en cuestión. Pero como GestorDeSprites también interpreta estados invertidos, en caso de que `invertido` tenga asignado el valor *true*, antes de buscar la existencia de una animación asociada con el estado recibido, se busca la existe una animación que dé respuesta a la versión invertida del estado en cuestión. En caso de encontrarse una animación definida para la versión invertida, esta se reproducirá, pero en caso de que el estado invertido no tenga animación asociada, se buscará una animación que dé respuesta, ahora sí, al estado normal recibido (quizás a este objeto no le era necesario distinguir entre animación invertida y no invertida para este estado).

Pues bien, después de haber hecho un inciso para explicar el mecanismo de funcionamiento de los estados invertidos y de GestorDeSprite en general, vamos a ver como se definen las animaciones que compondrán al Gestor:

Para cada animación se debe definir una etiqueta `<mov>` dentro de la etiqueta `<movimientos>`. Esta nueva etiqueta tendrá los dos atributos siguientes:

- `tipo`: Atributo que debe contener el tipo al que pertenecerá la animación que vamos a definir. En la siguiente tabla se describen los tres tipos de animación existentes:

Número Identificador del tipo	Nombre del tipo	Descripción
0	Normal	Animación que se reproduce de principio a fin. Una vez finalizada se salta a la reproducción de la animación *estadobase* o *estadobaseinvertido* según el caso.
-1	Cíclico	Animación que se reproduce una y otra vez de forma cíclica hasta que llega un nuevo estado que exije la reproducción de una animación diferente.
1	Pausa Final	Animación que se reproduce de principio a fin y cuando acaba se queda reproduciendo el último sprite hasta que llega un nuevo estado que exige la reproducción de una animación diferente.

Figura 8.7: Tabla que contiene los tres tipos de animaciones que el sistema permite definir. Su diferencia reside en la forma de reproducirse.

- `estado`: Atributo que debe contener el estado que activará la animación que se va a definir. Nótese que cada estado solo puede tener una animación asociada.
  Además, debamos recordar que los únicos estados que el Gestor de Sprites recibirá e interpretará son los de la tabla de la Figura 8.3 y los de la tabla de la Figura 8.6.

Conviene dejar claro que en un momento dado, un objeto solo puede estar reproduciendo una animación. Por este motivo, cuando se recibe un estado que tiene una animación asociada y que no es el estado cuya animación se está reproduciendo, se pausa la reproducción de la animación que en ese momento este en marcha; ya sea de tipo normal, cíclica o de pausa final; para permitir la reproducción de la nueva animación. De hecho, esta es la única forma de que las animaciones cíclicas y de pausa final cesen su reproducción.

En el caso de recibir un estado que no tenga ninguna animación asociada que le dé respuesta simplemente se hace como si dicho estado nunca hubiese llegado.

También es importante remarcar que hay un requisito importante a tener en cuenta cuando vamos a definir un Gestor de Sprites: el estado *estadobase* siempre debe tener una animación asociada. Esto es debido a que este estado asume un rol especial, pues es el estado que pasa a reproducirse cuando una animación del tipo normal completa su reproducción.

Una vez que ya hemos detallado los atributos de la etiqueta <mov> es hora de ver como se define la animación que dará respuesta al estado al que estamos apuntando.

Como ya se ha explicado, toda animación se compone de una serie de imágenes o sprites, pues bien, entre las etiquetas de apertura y cierre <mov> definiremos una etiqueta <sprite> por cada una de las imágenes que compondrán la animación que dicho <mov> está definiendo. Las etiquetas <sprite> tienen que declararse en el orden en el que los sprites que contienen se mostrarán a la hora de reproducir la animación. A continuación tenemos un ejemplo:

```
<mov tipo="0" estado="disparo">
<sprite ValueRow="0" ValueColumn="6"/>
<sprite ValueRow="1" ValueColumn="6"/>
<sprite ValueRow="2" ValueColumn="6"/>
<sprite ValueRow="3" ValueColumn="6"/>
</mov>
```

Ejemplo 8.15: Definición de una de las animaciones de las que se compone un Gestor de Sprites.

La animación del ejemplo se reproduciría de la siguiente forma: en primer lugar se visualizaría el primer *sprite* declarado, en segundo lugar el segundo *sprite*; y así hasta llegar al último. Una vez reproducidos todas las imágenes se comenzaría a reproducir la animación que diese respuesta a *estadobase* o a *estadobaseinvertido*, según el caso, ya que la animación es de tipo normal.

Los dos atributos que añadimos a cada etiqueta <sprite> son la fila y la columna que definen la posición del fichero de texturas donde se encuentra la textura que queremos asociar con este *sprite*. Para entenderlo debemos ver el fichero como una matriz de texturas, tal y como se ha explicado en los párrafos donde se habla acerca de *MatrizDeTiles*.

En caso de utilizar varios ficheros de texturas, para saber cuál es la fila que tenemos que indicar a la hora de asociar una textura que no se encuentre dentro del primer fichero, debemos acoplamos las matrices (ficheros de texturas) en orden de declaración. De forma que, por ejemplo, la fila 0 del segundo recurso declarado, será la fila número <Número de filas por fichero>. Recordemos que las filas van de 0 a <Numero de filas totales> - 1. Ver Figura 8.4 para entender mejor el concepto.

Por último, acerca de GestorDeSprites, hay que decir que, aunque en principio toda animación en reproducción se dejará de reproducir en post de un nuevo estado con animación asociada recibido, existe la posibilidad de definir estados cuya animación no se pueda llegar a parar. Claro está que a estos estados solo se les podría asociar una animación de tipo normal, ya que, en caso contrario, sus animaciones nunca dejarían de reproducirse.

En todo caso para el desarrollo del videojuego no ha sido necesaria la definición de ningún estado que requiera de esta propiedad, así que queda implementada como una alternativa viable a la hora de crear futuros estados. Para crear un estado que sea interpretado por Gestor de Sprites como un estado cuya animación no puede ser reproducida, este debería contener un nombre que siguiese el siguiente formato "<Nombre_estado>*+*" y su versión invertida tendría un nombre con el formato "<Nombre_estado>*invertido+*". Claro está, la definición de un nuevo estado pasa por tener que tocar la implementación Java y, por tanto, se sale de los objetivos de esta anexo.

3. Valoraciones finales

Hasta aquí la explicación completa de cómo definir niveles y objetos para el videojuego "Terrestrial Blast!" a través, únicamente, del lenguaje de programación XML.

Concluimos, de lo explicado en este anexo, que "Terrestrial Blast!" ha sido diseñado pensando en facilitar la labor de creación de nuevos objetos y de nuevos niveles. Siendo las herramientas que se deben utilizar para estas creaciones relativamente fáciles de utilizar; pues se hace uso del lenguaje XML, que es claro y simple.

Además, como hemos visto, la división de los objetos en componentes independientes permite que la definición de estos objetos se pueda valer de una combinación cualquiera de componentes. Las múltiples clases definidas como alternativa para implementar cada uno de estos componentes y el hecho de que estas se hayan definido de forma genérica permitiendo su configuración a través de atributos, hace que el número de posibilidades que tiene el editor XML sea bastante grande.

Tampoco deberíamos olvidar la relativa facilidad con la que podemos añadir nuevas implementaciones para cada componente. Este nivel de ampliación del videojuego no se ha tratado a fondo en este anexo por ser un nivel de mayor complejidad. Pero siempre es bueno que, si queremos crear un objeto con unas cualidades muy concretas que no se pueden definir a través de los componentes implementados por defecto, se puedan crear nuevas implementaciones de estos componentes, pudiendo así definir exactamente el objeto que queremos.

Por último, solo queda animar a que el lector que realmente se haya interesado por el mecanismo de construcción de niveles se plantee la creación de un nivel. Para ello es muy aconsejable ayudarse de los niveles y objetos ya definidos para el propio videojuego, pues sirven como ejemplo completo y ayudarán a acabar de comprender y consolidar lo explicado en este apartado.

Bibliografía

[1] Gartner. *Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011; Smartphone Sales Increased 42 Percent* [Online] Gartner official website <http://www.gartner.com/it/page.jsp?id=1848514> [Consult: 10-01-12]

[2] Google. *Services* [Online] Android developers <http://developer.android.com/guide/topics/fundamentals/services.html> [Consulta: 10-01-12]

[3] Google. *Content Providers* [Online] Android developers <http://developer.android.com/guide/topics/providers/content-providers.html> [Consult: 10-01-12]

[4] Google. *Activities* [Online] Android developers <http://developer.android.com/guide/topics/fundamentals/activities.html> [Consult: 10-01-12]

[5] Google. *User Interface* [Online] Android developers <http://developer.android.com/guide/topics/ui/index.html> [Consult: 10-01-12]

[6] Marvin Solomon, Marvin; Zimand, Marius. *Java for C++ Programmers* [Online] Marius Zimand website <http://triton.towson.edu/~mzimand/os/Lect2-java-tutorial.html> [Consult: 10-01-12]

[7] Several authors. *Rendering (computer graphics)* [Online] Wikipedia <http://en.wikipedia.org/wiki/Rendering_(computer_graphics)> [Consult: 10-01-12]

[8] Palevich, Jack. *Introducing GLSurfaceView* [Online] Android developers blog <http://android-developers.blogspot.com/2009/04/introducing-glsurfaceview.html> [Consult: 10-01-12]

[9] Google. *Android Market* [Online] Android Market <https://market.android.com/?hl=es> [Consult: 10-01-12]

[10] Catto, Erin. *Box2D v2.2.0 User Manual* [Online] Box2D official website <http://box2d.org/manual.pdf> [Consult: 10-01-12]

[11] LibGDX team. *What is libGDX?* [Online] LibGDX official website <http://libgdx.badlogicgames.com/> [Consult: 10-01-12]

[12] Shreiner, Dave. *OpenGL Programming Guide: The Official Guide to Learning* OpenGL. 7th ed. Upper Saddle River, NJ : Addison-Wesley, 2010

[13] Bergman, Per-Erik. *OpenGL ES Tutorial for Android* [Online] Jayway team blog <http://blog.jayway.com/tag/opengl-es/> [Consult: 10-01-12]

[14] Novo Rodríguez, Javier; et al. *¿Qué son los shaders?* [En línea] Página oficial de S.A.B.I.A. <http://sabia.tic.udc.es/gc/teoria/TrabajoHLSLs/shaders.htm> [Consulta: 10-01-12]

[15] Song Ho, Ahn. *OpenGL Vertex Buffer Object (VBO)* [Online] Song Ho Ahn website < http://www.songho.ca/opengl/gl_vbo.html> [Consult: 10-01-12]

[16] Oracle. *Nested Classes* [Online] Oracle documents website <http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html> [Consult: 10-01-12]

[17] *Royalty-Free Music* [Online] incompetech <http://incompetech.com/m/c/royalty-free/> [Consult: 10-01-12]

[18] Google. *Platform Versions* [Online] Android developers <http://developer.android.com/resources/dashboard/platform-versions.html> [Consult: 1-12-11]

[19] Larman, Craig. *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado.* 2ª ed. Madrid [etc.]: Prentice Hall, 2003.

[20] Pruett, Chris. *Aggregate Objects via Components* [Online] Replica Island blog <http://replicaisland.blogspot.com/2009/09/aggregate-objects-via-components.html> [Consult: 10-01-12]

[21] Several authors. *Extensible Markup Language* [Online] Wikipedia <http://es.wikipedia.org/wiki/Extensible_Markup_Language> [Consult: 10-01-12]

[22] Bray, Tim. *Traceview War Story* [Online] Android developers blog <http://android-developers.blogspot.com/2010/10/traceview-war-story.html> [Consult: 10-01-12]

[23] Pruett, Chris. *Google I/O 2010 - Writing real-time games for Android redux* [Online] The Google Code Channel (YouTube) <http://www.youtube.com/watch?v=7-62tRHLcHk&feature=relmfu> [Consult: 10-01-12]

[24] Pruett, Chris. Replica Island [Online] Google Code <http://code.google.com/p/replicaisland/> [Consult: 10-01-12]