Escuela Politécnica Superior

Emulador de Amstrad CPC embebido para publicación automatizada de juegos en Android

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

D. José Luis Luri Bolinski

Tutor/es:

Dr. Francisco José Gallego Durán

Septiembre 2019

Universidad de Alicante

Escuela Politécnica Superior

Departamento de Ciencia de la Computación e Inteligencia Artificial

Grado en Ingeniería Informática

Trabajo Fin de Grado

Emulador de Amstrad CPC embebido para publicación automatizada de juegos en Android

D. José Luis Luri Bolinski

jllb2@alu.ua.es

Tutores

Dr. Francisco José Gallego Durán

Índice de contenido

Índice de figuras

1 Agradecimientos

Quisiera expresar mi más sincero agradecimiento a D. Juan Carlos González Amestoy por toda su dedicación, orientación y apoyo a lo largo de estos años de trabajo.

También reconocer la enorme labor de mi tutor Dr. Francisco José Gallego Durán por haberme guiado, acompañado y ayudado en todo este proyecto.

Y finalmente darle las gracias a mi padre D. José Luis Luri Prieto por todo el esfuerzo que ha tenido que hacer para que pueda escribir estas palabras, por escucharme, aconsejarme y creer en mí.

2 Introducción

2.1 Acerca de los emuladores

Un emulador es un programa de ordenador que proporciona a un sistema —normalmente denominado *sistema anfitrión*— la capacidad de imitar las funciones de otro sistema diferente —conocido como *sistema invitado*—. Esta técnica recrea el hardware y software del sistema original, lo que permite ejecutar aplicaciones, juegos, dispositivos periféricos y otro tipo de componentes para los que en un principio no se encuentra diseñado.

El proceso de creación de un emulador es complejo, requiere de una alta especialización y un profundo conocimiento de la máquina original. Por ende, sólo se suele considerar que un emulador se halla en un estado avanzado de diseño, cuando proporciona un entorno y experiencia fiel al del sistema genuino.

Si observamos el modo en que han evolucionado los emuladores a lo largo de las décadas, podremos ver cómo se han dado una serie de tendencias que han determinado su modo de uso y aceptación entre los usuarios. El emulador fue concebido como una herramienta para crear una plataforma virtual en donde ejecutar un programa determinado. Por ello, a medida que han ido evolucionando, la funcionalidad de los emuladores ha mejorado al añadirse más opciones de configuración, soporte para dispositivos externos, *debuggers*[1], etc.

En virtud de un aumento de opciones técnicas, la usabilidad y accesibilidad al gran público se ha frenado. Los usuarios que quieran ejecutar en su equipo un software diseñado para otra plataforma deben realizar un esfuerzo, comprender su funcionamiento y afrontar una serie de barreras para poder cumplir con su objetivo. En consecuencia, el acceso a aplicaciones y títulos de plataformas, que a día de hoy no pueden ser ejecutados en equipos modernos, ha quedado reducido a un pequeño nicho de personas cuyo interés les ha permitido dedicar tiempo a introducirse en la materia.

A día de hoy y con las herramientas actuales, la dificultad de llevar un software a diferentes plataformas ya ha sido superada. Son muchas las tecnologías que ponen a disposición de los desarrolladores una serie de recursos para tal fin. Por ello, si se hace

[1] Un *debugger* —en castellano: depurador— es un programa usado para probar, depurar y analizar los errores de otros programas.

uso de la herramienta apropiada, el proceso puede ser relativamente sencillo; en otros muchos casos, puede significar tener que rehacer el juego o la aplicación por completo.

Si volvemos 35 años en el tiempo, a la década de los 80, comprobaremos que los recursos de entonces eran limitados, y los juegos y aplicaciones se diseñaban aprovechando al máximo las capacidades del sistema en el que iban a ser ejecutados. Debido a esto, los títulos quedaban anclados a un sistema en particular y, salvo que hubiesen si reprogramados o adaptados, resultaba imposible ejecutarlos sobre otras plataformas.

La capacidad de los procesadores no fue lo suficientemente potente hasta varios años después. Este hecho permitió que se desarrollaran las primeras aplicaciones capaces de emular fielmente el comportamiento de otro sistema. De esta manera, se hacía posible correr el juego de un sistema diferente, emulando su entorno original para el sistema que había sido diseñado.

Volviendo a la actualidad, son muchas las plataformas que cuentan con un sistema de emulación en Android; basta con acceder a la tienda de aplicaciones *Google Play Store*[2] de Google y hacer una búsqueda por su nombre. Sin embargo, su uso requiere de un aprendizaje por parte del usuario, así como de una serie de configuraciones para su puesta a punto. Además, las aplicaciones y juegos se distribuyen por un canal diferente, por lo que queda condicionado el alcance y la posibilidad de llevarlo a un público acostumbrado a tener acceso al contenido en un solo clic de ratón.

2.2 CPCTelera

La herramienta CPCTelera se define como un *framework*[3] que integra un conjunto de herramientas que agilizan el proceso de desarrollo de videojuegos para Amstrad CPC, y cuya filosofía es proveer un entorno fácil de configurar, rápido, usable y bien documentado. Además, pone a disposición de los programadores una serie de APIs[4] y librerías para manejar gráficos, audio, gestión de memoria, entre otras cosas. Mediante

[2] *Google Play Store* es una plataforma de distribución digital de aplicaciones móviles para los dispositivos con sistema operativo Android.

[3] Un *framework* —en castellano: entorno de trabajo— es un conjunto estandarizado de conceptos, prácticas y recursos para la construcción y distribución de aplicaciones.

[4] Una *Application Programming Interface* —en castellano: Interfaz de Programación de Aplicaciones— es un conjunto de subrutinas, funciones y procedimientos que ofrece una biblioteca para ser utilizada por otro software.

un proceso automatizado de compilación cruzada, es capaz de generar versiones de los juegos en formato *CDT*[5] y *DSK*[6] para la plataforma Amstrad.

La Escuela Politécnica de la Universidad de Alicante promueve en diferentes ingenierías el aprendizaje e interés por la programación a bajo nivel a través de esta herramienta. La simplicidad de un sistema de 8 bits permite a los alumnos comprender el funcionamiento más elemental de un computador. A día de hoy, las herramientas más utilizadas ocultan detalles y conceptos en virtud de una mayor productividad, y no por ello dejan de ser conocimientos muy necesarios para la formación integral de un ingeniero.

2.3 Objetivos

Con el fin de llevar la visibilidad de los juegos desarrollados bajo esta plataforma a otro nivel, surge la idea de integrar un nuevo componente en CPCTelera que permita generar proyectos de forma automatizada en plataformas Android.

Mediante un concepto innovador y revolucionario, contrario al de una aplicación en forma de emulador que ejecuta juegos externos, se presenta un formato en donde el juego lleva integrado un emulador embebido que le permita ser ejecutado como si se tratara de una aplicación nativa.

Gracias a que el objeto resultante queda ensamblado en un único producto, éste puede distribuirse por un mismo canal y abstraer al usuario final de la experiencia del proceso de emulación.

Del mismo modo, el propósito de este proyecto es seguir contribuyendo a la comunidad Amstrad internacional, e impactar positivamente sobre el desarrollo formativo de los alumnos de la Universidad de Alicante.

2.4 Motivación

Vivimos en un mundo virtualizado, incierto y cambiante, en donde el sentimiento de nostalgia nos hace sentirnos más seguros. Aquellos objetos y recuerdos que formaron

[5] El formato CPC Digital Tape, conocido por las siglas CDT, es un formato de archivo en el que se almacena la estructura y contenidos de una cinta de casete.

[6] Es un formato estandarizado a través del cual se almacenan la estructura y contenido de un disquete en un archivo digital.

parte de nuestro pasado, a veces se convierten en un refugio que somos capaces de elevar a un símbolo dotado de un aura especial.

Recuerdo mi primer acercamiento a un ordenador, sentado sobre las rodillas de mi tío. En un juego, un héroe pérsico tenía que cruzar un palacio lleno de enemigos y trampas para poder salvar a su princesa. Pero no es el único recuerdo que conservo, puedo contar cientos de momentos e historias. Cada etapa de mi infancia viene enmarcada con una aventura, una melodía o un personaje que por suerte pude compartir con amigos, familiares o disfrutar en soledad.

No recuerdo el momento exacto en el que descubrí mi vocación por la informática. No obstante, puedo reconocer que, a consecuencia de los videojuegos, se fraguó en mí una necesidad por aprender, crear y compartir, y fue entonces cuando me sumergí por completo en el arte de la programación.

Sin embargo, los emuladores siempre me resultaron un gran misterio. Me fascinaba y me sigue fascinando cómo una aplicación puede ser lo más parecido a una máquina del tiempo en donde rememorar aquellos momentos de felicidad. Pero no fue hasta llegar a la Universidad cuando obtuve las aptitudes necesarias para comprender e interiorizar todo el aspecto técnico que hay detrás de ellas y la magia que las hace funcionar.

En definitiva, este proyecto es mi tributo y una manera de materializar mi admiración hacia todas aquellas personas que han contribuido en mayor o menor medida al desarrollo de estas maravillas de la ingeniería.

2.5 Alcance del proyecto

Durante la planificación del proyecto fue necesario concretar con exactitud el alcance y las metas que me proponía cubrir con mi tutor Francisco José Gallego Durán. Los dos grandes hitos que nos propusimos llevar a cabo fueron: empaquetar y llevar los juegos de Amstrad a Android mediante un emulador embebido y, segundo, automatizar todo ese proceso y su integración en CPCTelera.

Diseñar e implementar un emulador desde cero es un proceso extremadamente complejo, y llevarlo a un estado de madurez como para que sea útil puede requerir años de trabajo y dedicación. Teniendo en cuenta el objeto del proyecto y su finalidad, resultaba mucho más interesante y apropiado utilizar algún emulador ya existente, cuya

calidad y funcionalidad estuviesen más que probadas, y que además se adaptase a nuestras necesidades.

Por otro lado, el proceso de automatización nos planteaba un cúmulo de incertidumbres acerca de la viabilidad y desarrollo. En los inicios, ya tenía cierta experiencia con el entorno de desarrollo de Android, y quien lo conozca sabe que está compuesto por varios gigabytes de librerías, herramientas y ficheros de configuración. Realizar un proceso de automatización sobre él y, por otra parte, mantenerlo en el tiempo, podría constituir sin duda un verdadero rompecabezas.

Finalmente, y dando por hecho que resultara posible desarrollar un método de automatización, tampoco contábamos con la seguridad de que el objeto resultante de este proceso pudiese ser distribuido a través de las tiendas de aplicaciones. En cualquier caso, decidimos ponernos manos a la obra y afrontar todas estas dificultades en su debido momento.

3 Estudio del estado del arte

3.1 Introducción

A día de hoy, no son pocas las empresas que continúan dilatando la vida de sus viejos títulos. La empresa SNK, famosa en el pasado por sus máquinas recreativas y videoconsolas Arcade, ha lanzado sus clásicos sobre diferentes plataformas. Tanto PlayStation, Nintendo o la mismísima Xbox han recibido versiones de sus joyas más preciadas y, como no podía ser menos, ahora han dado el salto a las plataformas móviles.

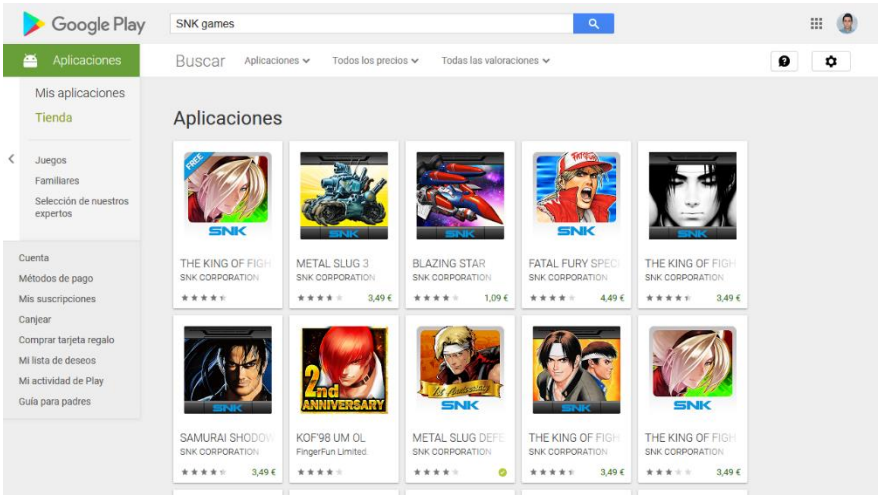

Figura 1 - Búsqueda del término «SNK games» en Google Play Store

Mediante una colaboración con la empresa francesa Dotemu, especializada en *retrogaming*, desarrollaron una tecnología para facilitar dichos *ports*[7]. La propia desarrolladora y distribuidora Dotemu, que se vio envuelta en polémica por usar un emulador gratuito, acabó emitiendo una nota de prensa en la que confirmaba disponer de una licencia especial para hacer uso del emulador *Nebula*.

Si analizamos alguno de sus juegos más destacados, por ejemplo el *Metal Slug* para Android, podremos ver que el juego se visualiza en una resolución de 4:3 y con un mensaje en la parte superior derecha que insta al usuario a insertar una moneda. Es decir, los usuarios están jugando a la versión original de 1996 para la recreativa *Neo*

[7] Un *port* es un término utilizado en la industria de los videojuegos cuando un juego diseñado para una plataforma es convertido para funcionar en otra plataforma diferente.

Geo MVS Arcade Cabinet de manera transparente en sus ordenadores, consolas o teléfonos móviles.

Figura 2 - Captura de pantalla de «Metal Slug» en un dispositivo Android

Al mismo tiempo, la empresa japonesa Sega, conocida por muchos de sus juegos y videoconsolas, también está aprovechando estas tendencias para lanzar sus clásicos sobre plataformas móviles. De la misma forma que SNK, mediante un emulador embebido, consigue volver a llevar juegos de generaciones pasadas sobre las plataformas actuales.

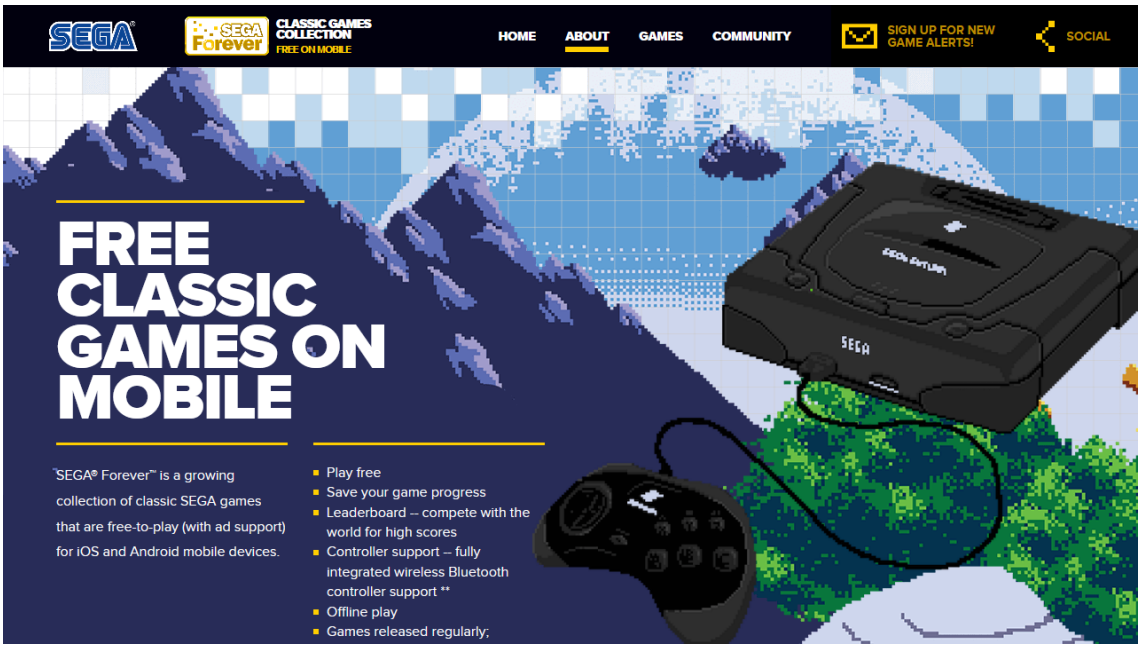

Figura 3 - Captura de pantalla del sitio web dedicad a «SEGA Forever»

Sin embargo, no todas las plataformas cuentan con un producto similar en el mercado. Muchas de las empresas desarrolladoras apuestan por fórmulas más rentables, mientras que otras han desaparecido o reorientado sus negocios, como es el caso de Amstrad. Esta firma, pasados los 90, dejó de centrarse en ordenadores para hacerlo sobre dispositivos de comunicaciones. Dada la situación actual del mercado, no existe una propuesta comercial o comunitaria para llevar títulos de Amstrad CPC a otras plataformas mediante esta fórmula.

Para sacar adelante el proyecto, será necesario estudiar y analizar los componentes que forman este tipo de aplicaciones de forma individualizada. Por un lado, examinaremos el estado actual de los emuladores de Amstrad CPC, el proceso necesario para su ejecución en Android y los mecanismos disponibles para la generación de forma automática de aplicaciones para dispositivos móviles.

3.2 Búsqueda de un emulador de Amstrad CPC

El emulador es el *engine*[8] y núcleo del proyecto. Es el componente que se va a encargar de marcar la diferencia entre un producto robusto y funcional y algo a medio hacer. De esta forma, consideramos importante dedicar tiempo a analizar, probar e investigar las diferentes alternativas, así como hacer alguna que otra prueba de concepto para ver con más detalle las problemáticas que pudiesen surgir.

La búsqueda de un emulador de Amstrad CPC no fue una tarea sencilla. Elegir erróneamente un emulador en una etapa temprana supondría echar por tierra mucho esfuerzo, comprometer los tiempos que nos habíamos marcado y echar un paso atrás en el desarrollo.

Por ello, fue necesario determinar los puntos clave que debían ser considerados a la hora de elegir un emulador:

- **Calidad**: el emulador debe ser de una buena calidad tanto en su ejecución, como en su implementación. El usuario final debe percibir fluidez tanto en la imagen como en el sonido.
- **Rendimiento**: el emulador debe estar optimizado, ya que será ejecutado sobre dispositivos móviles sujetos a limitaciones de potencia y batería.

[8] Un *engine* —en castellano, motor— se refiere a una pieza de software cuyas rutinas permiten el diseño, creación y funcionamiento de otro software.

- **Compatibilidad**: la emulación debe ser precisa para garantizar una alta compatibilidad con el software de la máquina original, y más teniendo en cuenta que será una solución genérica para desarrolladores.
- **Licencia**: debe estar bajo una licencia libre o, en su defecto, bajo una que cuente con la aprobación de sus desarrolladores para ser distribuido legalmente como un emulador embebido.
- **Tecnología**: el emulador debe estar implementado en alguno de los lenguajes de programación soportados oficialmente por Android —actualmente *Java*, *Kotlin* y *C/C++*—.

A continuación, me propuse conocer el estado actual de los emuladores diseñados específicamente para Android. Disponer de un emulador avanzado y diseñado para esta plataforma hubiese sido el punto de partida ideal. Tras hacer una evaluación de los tres principales proyectos más populares, llegué a la conclusión de que ninguno de ellos se encontraba en un punto apropiado de madurez, compatibilidad y mantenibilidad como para ser adoptados en el proyecto. La oferta de emuladores de Amstrad CPC para Android es muy limitada y de una calidad muy cuestionable. Las diferentes alternativas apenas han sido actualizadas en el tiempo y a día de hoy se encuentran en un estado de total abandono. En mi opinión, el hecho de que haya alguien detrás del emulador es fundamental. Tenía muy presente que era cuestión de tiempo que surgiera alguna dificultad y, en un software de este tipo, es muy difícil dar con una solución si no se conoce con detalle su funcionamiento.

Por ello, opté por continuar con la búsqueda de un emulador de mayor calidad en otra plataforma. Asumí que portar el emulador a Android llevaría más trabajo en un primer momento, pero a largo plazo aportaría mayores beneficios. La escena de emulación en PC es totalmente diferente. Normalmente es la principal plataforma en donde se desarrollan los primeros emuladores para cada sistema y en donde se dan más alternativas, calidad y funcionalidad. Analicé los emuladores con mayor reputación y seleccioné aquellos que se ajustaban a los requerimientos que habíamos marcado.

Tras hacer una comparativa entre ellos, opté por empezar a trabajar sobre Arnold. De entre todos los emuladores disponibles, es el único que cuenta con todos los requisitos que nos habíamos propuesto. Es un emulador con más de una década de desarrollo y que además presenta un estado avanzado y reconocido por la comunidad. Se encuentra desarrollado en C, cuenta con un buen rendimiento y sus fuentes están disponibles en Internet bajo una licencia libre que permite su distribución de todas sus formas.

En definitiva, ya tenía un emulador sobre el que trabajar y realizar las primeras investigaciones. El siguiente objetivo era estudiar su código fuente, adaptarlo para su compilación en Android y desarrollar el primer componente necesario para montar una prueba de concepto.

3.3 Comparativa

3.3.1 Emulación actual en Android

3.3.1.1 DroidCPC

DroidCPC es un emulador comercial de Amstrad CPC para dispositivos Android. Se caracteriza por ofrecer un buen rendimiento en todo tipo de dispositivos, además de ser compatible con controladores y teclados externos.

Apenas cuenta con algo más de 5.000 instalaciones y su última actualización fue a finales del año 2016. Entre los comentarios destacados por la comunidad de *Google Play*, parece destacar que tiene problemas de compatibilidad con bastantes juegos.

3.3.1.2 CPCDroid

CPCDroid es un proyecto de software libre que lleva una emulación completa de los CPC 464, 664 y 6129 a dispositivos Android. Está basado en el conocido emulador Caprice32, el cual destaca por su grado de compatibilidad con software CPC, al que se han incorporado funciones de capturación de pantalla, sistema de trucos, reasignación de teclado, cambios de configuración en caliente, etc. Por contra, éste no se distribuye a través de la tienda *Google Play*, y por lo que se puede ver en el sitio web del desarrollador, dejó de mantenerse hace ya algunos años.

3.3.1.3 AndCPC

AndCPC es otro proyecto de software libre basado en el emulador Caprice32 para llevar Amstrad CPC a sistemas Android. A pesar de estar en un estado muy temprano de desarrollo, ofrecía una buena compatibilidad con software CPC. Sin embargo, nunca llegó a implementar un teclado o joystick virtual, por lo que los usuarios se veían obligados a conectar un teclado físico para poder hacer uso de los juegos. El proyecto no llegó a tener una buena aceptación y fue abandonado al cabo de un tiempo.

3.3.2 Emulación actual en PC

3.3.2.1 WinApe

WinApe es un emulador de referencia debido a su estabilidad, rendimiento y compatibilidad con Amstrad CPC, CPC Plus y los modelos de Schneider. Cuenta con más de 20 años de desarrollo y se distribuye gratuitamente para la familia de sistemas operativos de Microsoft.

Por contra, se encuentra desarrollado en el lenguaje de programación Delphi, el cual no está soportado en Android, y a cuyas fuentes se ha decidido restringir el acceso.

3.3.2.2 Arnold

Arnold es un emulador multiplataforma, desarrollado en el lenguaje de programación C; la comunidad le atribuye un buen rendimiento y compatibilidad con Amstrad CPC, CPC Plus, KC Compact, Aleste 520Ex y GX4000. Su desarrollo continúa activo tras más de 15 años, desde que dio a luz su primera versión; su código fuente es accesible desde su sitio web bajo licencia GPL[9].

3.3.2.3 Roland

Roland es un emulador de la serie de ordenadores Amstrad/Schneider CPC 464/664/6128 para sistemas Linux y Windows. Se encuentra desarrollado en C++11 y, como puede comprobarse en su sitio web y repositorios, cuenta con una buena documentación. Por contra, todavía se encuentra en una fase temprana. No hay garantías ni retroalimentación por parte de la comunidad que nos asegure una buena compatibilidad con el software de la máquina. Además, tampoco existe un desarrollo continuo que pueda apuntar mejoras con el tiempo.

3.3.2.4 Retro Virtual Machine

Retro Virtual Machine es un emulador de Amstrad CPC 464/664/6128 y Zx Spectrum 48k/128k/+2A/+3 para plataformas Windows, Linux y Mac. Es sin lugar a dudas, uno de los mejores emuladores desarrollados hasta la fecha, tanto por su aspecto técnico como por el apartado artístico de la aplicación. Su desarrollo se encuentra activo pues su implantador continúa mejorando el emulador e incluso planea la introducción de nuevas

[9] La Licencia Pública General de GNU, conocida en inglés por sus siglas GPL, se refiere a un tipo de licencia de software que garantiza a los usuarios la libertad de usar, estudiar, compartir y modificar el software.

plataformas como MSX. Su desarrollador es Juan Carlos González Amestoy, de Alicante, quien actualmente distribuye su emulador de forma gratuita a través de Internet bajo una licencia Freeware.

Cuadro comparativo

Emulador	Sistema	Estado	Rendimiento	Compatibilidad	Tecnologías	Licencia
Droid CPC	**Android**	**Prematuro**	**Medio**	**Bajo**	**n. d.**	**Shareware**
CPC Droid	**Android**	**Prematuro**	**Medio**	**Medio**	**C**	**GNU Public License**
And CPC	**Android**	**Prematuro**	**Medio**	**Medio**	**C**	**GNU Public License**
WinApe	**PC**	**Avanzado**	**Alto**	**Alta**	**Delphi**	**Freeware**
Arnold	**PC**	**Avanzado**	**Alto**	**Alta**	**C**	**GNU Public License**
Roland	**PC**	**Prematuro**	**n. d.**	**n. d.**	**C++11**	**GNU Public License**
Retro Virtual Machine	**PC**	**Avanzado**	**Alto**	**Alta**	**C/LUA**	**Freeware**
SugarBox	**PC**	**Prematuro**	**n. d.**	**n. d.**	**n. d.**	**Freeware**

3.4 Primer prototipo con Arnold

El primer objetivo fue conseguir las fuentes de Arnold para analizar su funcionamiento. Su autor las distribuye libremente a través de Internet bajo una licencia de software libre en diferentes foros y páginas web. Por ello pude empezar rápidamente con los preparativos para la investigación.

A continuación, me centré en identificar aquellas dependencias a las que se hallaba sujeto el emulador. Al tratarse de una aplicación multiplataforma para Windows y Linux, era muy probable que estuviese utilizando algún tipo de biblioteca para crear una interfaz gráfica común. Me inquietaba que hubiese dependencias acopladas al motor de emulación que me obligasen a modificar componentes internos de Arnold. Tras analizar la estructura del proyecto y leer parte de sus fuentes, identifique la biblioteca *wxWidgets* para la creación de los elementos gráficos de la interfaz y SDL[10] para el acceso a vídeo, audio y la gestión de entrada de dispositivos.

Seguidamente fui adaptando estas fuentes para su ejecución en Android. Por suerte, la biblioteca SDL contaba con un proyecto oficial para Android. De esta forma, la dependencia con esta biblioteca no plantearía un problema y nos serviría como base del proyecto. Sin embargo, no ocurría lo mismo con *wxWidgets*, instrumento que está diseñado para crear interfaces gráficas en aplicaciones de escritorio Windows, Mac y Linux.

Por lo tanto, concentré mis esfuerzos en extraer el núcleo de Arnold, de manera que sólo dependiese de SDL para funcionar. De esta forma, llevar el emulador a Android supondría una tarea sencilla pues no requeriría de muchos ajustes. El objetivo era extraer aquellas partes fundamentales de Arnold que permitiesen una emulación básica sin tener que modificar en exceso su código fuente. Sin embargo, tras más de un mes de trabajo, me encontraba lejos de obtener una versión funcional. El acoplamiento a las dependencias, la poca documentación disponible y la complejidad para entender y leer el código no me permitían obtener una versión operativa.

Viendo que me empezaba a encontrar en una situación de bloqueo, me propuse hablar con Francisco José Gallego Durán, mi director del trabajo. Tan pronto como le plantee las dificultades a las que estaba haciendo frente, y que me obligaban a introducir excesivas modificaciones en el código fuente de Arnold, él me propuso descartar por el

[10] Las bibliotecas Simple DirectMedia Layer, también conocida SDL, proporcionan funciones básicas para el manejo de vídeo, audio y periféricos a través de OpenGL y Direct3D.

momento este emulador y que estableciésemos contacto con el desarrollador de Retro Virtual Machine.

3.5 Segundo prototipo con Retro Virtual Machine

Después del intento fallido con Arnold, pensamos que una posible colaboración con otro emulador podría ser una mejor alternativa. Nos pusimos en contacto con Juan Carlos González Amestoy, desarrollador de Retro Virtual Machine, al que invitamos a una reunión para darle a conocer el proyecto y plantearle una posible colaboración con su emulador.

Tan pronto como le explicamos los objetivos, los requisitos que nos habíamos marcado y los problemas que nos habíamos encontrado, él se ofreció a participar en el proyecto de manera totalmente desinteresada. Dado que ya teníamos un proyecto base, montado sobre SDL para Android, nos propusimos que por un lado Juan Carlos se encargara de extraer el motor de su emulador para hacerlo correr sobre la biblioteca SDL en PC, mientras que yo, por el otro, lo adaptaría a la versión de Android. Tras más de un mes de trabajo obtuvimos los primeros resultados. Conseguimos hacer correr el emulador y lanzar el intérprete de Basic de Amstrad CPC en un teléfono.

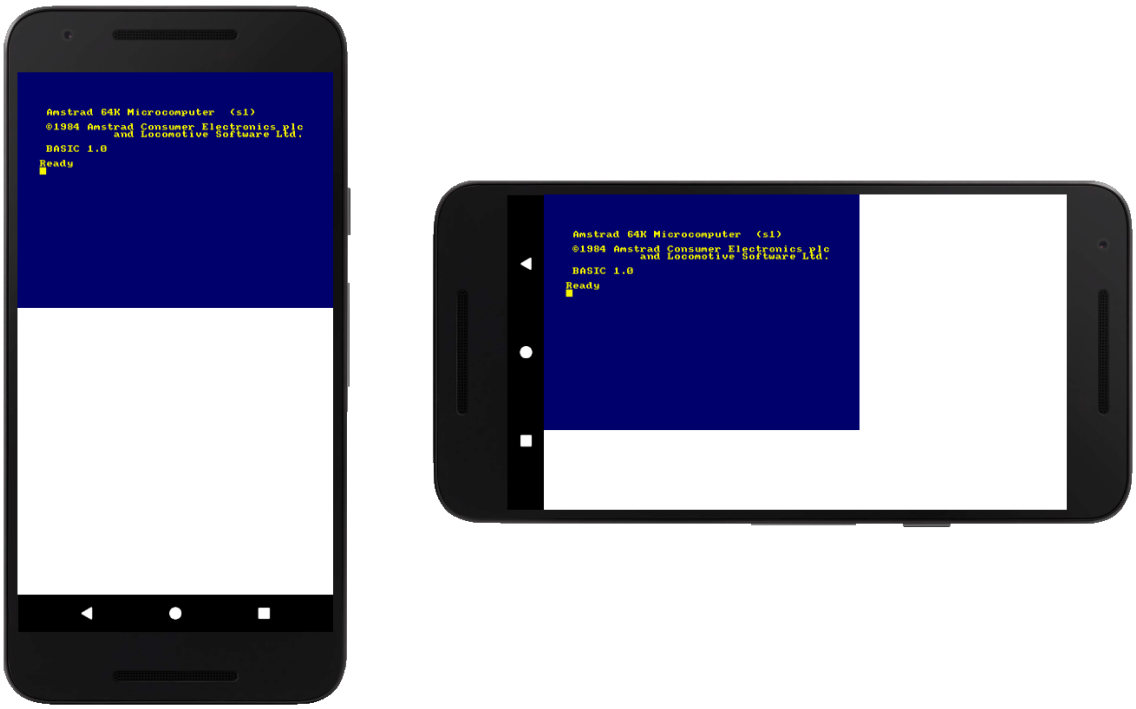

Figura 4 - Captura de pantalla de la primera versión del prototipo

Sin embargo, el prototipo todavía no resultaba del todo funcional. Por un lado, el núcleo del Retro Virtual Machine que estábamos ejecutando no tenía implementados los sistemas de cintas o disquetes con los que cargar un juego. Y por otro, la aplicación no contaba con un mecanismo de entrada con el que interactuar con el emulador.

De esta forma el siguiente objetivo que nos propusimos vendría a solucionar parte del problema. Queríamos añadir un método por el cual pudiésemos lanzar los juegos directamente sobre emulador de manera automática.

3.6 Carga automática de juegos por SNA

Cuando ponemos en marcha un ordenador Amstrad CPC, lo primero en aparecer en una pantalla es un el *prompt*[11] de *Basic*. Un terminal de letras amarillas sobre un fondo azul permite al usuario interactuar con la máquina a través de una serie de comandos y operadores.

Para proceder a ejecutar un juego, basta con introducir el disquete o casete en la ranura correspondiente, lanzar la orden *cat* para listar su contenido y, a continuación, ejecutar la orden *run* junto al nombre del fichero de programa. Seguidamente, un programa denominado l*oader*[12] vuelca la información desde el disquete o casete en la memoria principal. Este proceso requiere de varios minutos de espera hasta completarse y debe repetirse cada vez que se desee ejecutar el juego.

El método descrito anteriormente requiere de intervención del usuario y de tiempos de espera. Sin embargo, uno de los objetivos del proyecto es abstraer al usuario del sistema original y mejorar su experiencia. Dado que estamos en un entorno virtual, es posible almacenar el estado de la máquina emulada en un determinado punto en el tiempo —por ejemplo, en el instante en que el juego ha sido completamente cargado— para luego ser restaurado.

Este estado en forma de fichero se conoce como Snapshot o SNA. Su estructura ha sido estandarizada por la comunidad y son muchos los emuladores que ya han adoptado este formato para almacenar el estado del emulador. Mediante este sistema es posible evitar lanzar el intérprete de Basic y llevar al usuario al instante deseado.

---

[11] El prompt es un carácter o conjunto de caracteres que se muestran en una línea de comandos para indicar que está a la espera de órdenes.

[12] Un *loader* —en castellano, cargador— es un programa encargado de cargar los datos para otro juego o programa.

Cuando se compila un juego sobre CPCTelera, se generan tres ficheros: *.*DSK*, *.*CDT* y *.*SNA*[13]. El primero es relativo al juego en formato disquete, el segundo en formato casete y el último se corresponde al estado mencionado anteriormente.

Con lo cual, tras la implementación de este tipo de carga en Retro Virtual Machine por parte de Juan Carlos y su integración en el prototipo, pudimos llegar a ejecutar los primeros juegos desde este tipo de formato. No obstante, y tras hacer algunas pruebas, nos resultó llamativo que el rendimiento de la aplicación no fuera del todo el deseado.

3.7 Problemas de rendimiento SDL

Como ya se ha comentado en puntos anteriores, la aplicación hace uso de *SDL* para el manejo de vídeo y audio. Los ensayos sobre teléfonos móviles reales daban como resultado un rendimiento muy pobre. La fluidez de la imagen no era tan precisa como debía e incluso se producían una serie de chasquidos en el sonido que demostraban que algo no funcionaba bien.

Dado que no avanzaba en la materia y no encontraba el origen de mis dificultades, me vi obligado a solicitar la ayuda de Juan Carlos para localizar el foco del problema. Gracias a su profundo conocimiento sobre emulación e ímpetu por la optimización, él determinó que nuestro problema estaba siendo causado por la biblioteca *SDL*. La conversión de *Surfaces*[14] a *Textures*[15] que utilizábamos para dibujar los fotogramas del juego estaba dilapidando recursos de manera innecesaria. Y llegamos a la conclusión de que valía la pena prescindir de *SDL* y montar la aplicación totalmente nativa sobre la *API* de Android. La ganancia de rendimiento fue abrumadora y con ella se solucionaron los problemas derivados en imagen y sonido.

3.8 Empaquetando juegos

Llegados a este punto y con una emulación que funcionaba sorprendentemente bien, nos propusimos dar el siguiente paso e investigar acerca de los procesos para la generación automática de aplicaciones.

---

[13] El formato SNA fue definido por primera vez para el emulador CPCEMU y permite almacenar el estado de un sistema en un momento determinado.

[14] Una *Surface* —en castellano, superficie— es una estructura que contiene una colección de píxeles para su posterior combinación con otros conjuntos de datos.

[15] Una *Texture* —en castellano, textura— es un tipo de estructura que contiene datos de forma eficiente para su posterior representación.

Para trabajar sobre el proyecto es necesario un conjunto de herramientas. La principal es Android Studio, el entorno de desarrollo integrado oficial para la creación de aplicaciones para Android, basado en IntelliJ IDEA. Gradle como base del sistema de compilación, NDK de Android para poder implementar partes de la aplicación en código nativo, mediante lenguajes como C y C++, y el SDK de Android para para crear y depurar aplicaciones, así como controlar dispositivos Android que estén conectados.

Uno de los principales objetivos del proyecto es ofrecer una herramienta que permita portar juegos a Android de forma sencilla y totalmente transparente. El proceso de automatización que deseábamos conseguir debía funcionar de forma autónoma y con la menor intervención por parte de los usuarios. Sin embargo, el conjunto de herramientas y librerías ocupa en su totalidad aproximadamente entre 2GB y 3GB de espacio en disco. Su correcta instalación requiere de una serie de descargas, procesos de configuración, resolución de dependencias y demás contingencias dependientes del entorno. Sin lugar a dudas, un proceso excesivamente complejo y difícil de mantener en el tiempo.

Dada la situación, decidimos dar un nuevo enfoque a la generación de las aplicaciones e investigar nuevos métodos alternativos. Con el objetivo de evitar delegar en el usuario el proceso de compilación y eliminar en la medida de lo posible dependencias de terceros, surgió la idea de trabajar sobre una versión previamente compilada de la aplicación. Mediante esta técnica, sólo sería necesario distribuir una aplicación base junto a un mecanismo que se encargara de introducir el juego en su interior. La realidad es que en un primer momento no estábamos muy seguros de si este proceso iba a resultar ser posible. Las aplicaciones una vez compiladas son empaquetadas, formateadas y firmadas, por lo que muy probablemente manipularlas no iba a ser tarea sencilla.

En consecuencia, tuve que dedicar un tiempo a investigar acerca del formato y estructura de los paquetes de aplicación de Android. Este periodo de análisis me llevó a dar con algunas herramientas de ingeniería inversa que permiten desensamblar estos paquetes y manipularlos para adaptarlos a nuestras necesidades. Mediante ejercicios de prueba y error con diferentes aplicaciones encontré la manera de reemplazar componentes de la aplicación garantizando la integridad del paquete —en un capítulo posterior entraré en más detalle acerca de este proceso—. Tras elaborar un pequeño script que se encargara de reproducir de forma automática las tareas que hasta el momento tenía que hacer de forma manual, pude corroborar que el procedimiento se

realizaba correctamente y que todas las etapas que nos habíamos propuesto eran resueltas y estaban cubiertas por el prototipo.

3.9 Conclusión

La investigación y estudio del estado actual en este campo me ha permitido establecer todos los elementos teóricos y prácticos que sustentan la elaboración de este proyecto. Sin embargo, mediante la construcción del prototipo, he podido probar que el flujo que nos habíamos propuesto es posible, y también me ha permitido identificar riesgos y deficiencias que de otra manera me hubiesen entorpecido en mi trabajo.

En los siguientes capítulos aplicaré todos los conocimientos adquiridos en este proceso para la construcción de un producto final a partir del prototipo. Para su diseño aplicaré técnicas de ingeniería del software que me permitan obtener una aplicación funcional, escalable y sostenible en el tiempo, como eficiente y fiable en diferentes tipos dispositivos Android.

4 Diseño del proyecto

A la hora de diseñar la aplicación he considerado oportuno agrupar de forma lógica cada uno de los componentes. La asignación de responsabilidades y el desacoplamiento entre componentes que no están relacionados permiten aumentar la flexibilidad, la escalabilidad, mantenibilidad y reutilización del código. Además al estructurar adecuadamente sus partes y reducir la cohesión entre ellas, minimiza el riesgo de que la realización de cambios en alguna de las partes tenga impacto en las demás.

Los componentes se agruparán básicamente en tres tipos principales: *activities* —comúnmente conocidos como controladores— en la capa de presentación, *servicios* en la capa de negocio y *repositorios* en la capa de datos. Cada agrupación de componentes tiene un propósito desde el punto de vista de arquitectura y cada componente una única responsabilidad dentro de la lógica de negocio de la aplicación. En los siguientes capítulos trataré con detalle cada uno de los tipos de componentes, sus responsabilidades y cómo han sido implementados.

Figura 5 - Diagrama con los principales componentes de la aplicación

5 Capa de presentación

5.1 Introducción

En el ecosistema Android las aplicaciones están formadas por un conjunto de pantallas que permiten la interacción con el usuario. Dicha funcionalidad está implementada a través de un patrón de arquitectura MVC —modelo, vista y controlador—, en donde se separan los conceptos y responsabilidades para mejorar la cohesión, reutilización de código y su mantenimiento.

Los controladores reciben el nombre de *activities* y son los encargados de responder a los eventos e interactuar con los modelos y servicios. Por otra parte, las vistas se representan a través de un fichero *XML*[16], en donde se define el diseño y estructura visual de la interfaz de usuario. Una vez iniciada la aplicación, ésta puede cambiar entre diferentes *activities* en función de la lógica de negocio, la interacción por parte del usuario o el propio sistema.

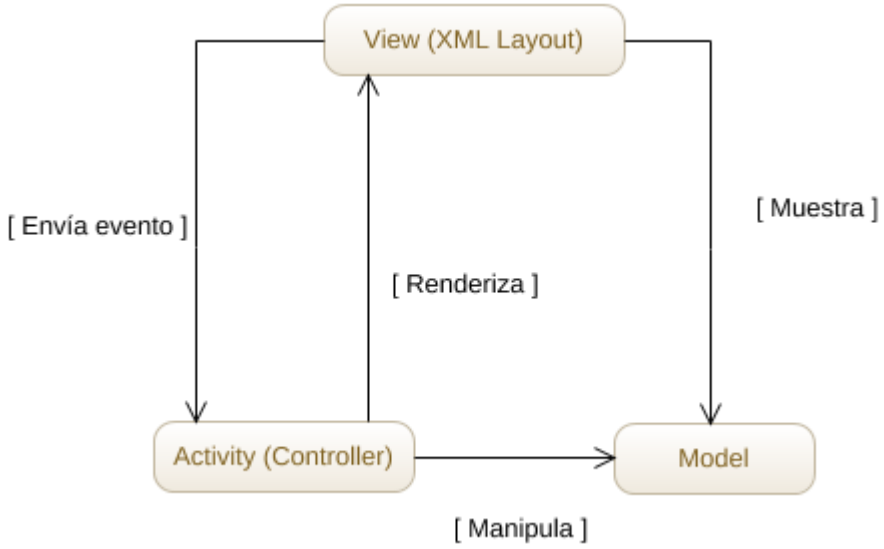

Figura 6 - Diagrama sobre la comunicación entre elementos

[16] El formato eXtensible Markup Language, conocido por sus siglas XML, es un meta-lenguaje que permite estructurar, almacenar y transportar datos de forma legible.

5.2 Layout (Vista)

La apariencia y los componentes de la interfaz de usuario, que forman una pantalla, se consideran un recurso —en inglés, *resource*— denominado *layout*. Mediante ficheros en formato *XML* se define la estructura y diseño de la vista para diferentes resoluciones y modos de pantalla.

Android Studio, que es la principal herramienta con la que se desarrollan aplicaciones en Android, ofrece un editor visual con el que componer la apariencia de nuestras interfaces; del mismo modo que permite exportar de forma automática dichas estructuras a los ficheros antes mencionados.

Es importante tener en cuenta que estos ficheros deben tener el mismo nombre. En primer lugar, porque el nombre del fichero es utilizado como identificador de referencia y, segundo, porque es la manera en que Android determina que pertenecen el mismo *layout*.

Cada fichero debe ser ubicado en unas rutas específicas dentro del proyecto. Existen muchas combinaciones en función de las necesidades de la aplicación pero, para nuestro caso particular, bastará con tener uno para modo retrato que se ubicará en `/res/layout` y el apaisado en `/res/layout_land`.

Por ejemplo, en las siguientes rutas de directorios se encuentran dos ficheros con el mismo nombre. Ambos en combinación describen un único *layout*, identificado como "*my_layout*" y para el que se dispone un modo retrato y apaisado:

- `/res/layout/my_layout.xml (Modo retrato)`
- `/res/layout_land/my_layout.xml (Modo apaisado)`

Posteriormente, desde el proyecto será posible hacer referencia a dicho *layout* a través de un identificador compuesto por el nombre del fichero y prefijo *R.layout:*

```
R.layout.my_layout
```

5.3 Activity (Controlador)

Los controladores en Android reciben el nombre de *activity.* Actúan como un intermediario entre la interfaz de usuario y los servicios, y sus funciones son las de gestionar los eventos producidos en la aplicación e invocar a la lógica de negocio.

Las *activities* tienen un ciclo de vida y una serie de métodos que se invocan en función del sistema y la interacción con el usuario. En la siguiente imagen se detallan los diferentes estados por los que puede pasar una *activity* y los métodos asociados a cada uno de ellos:

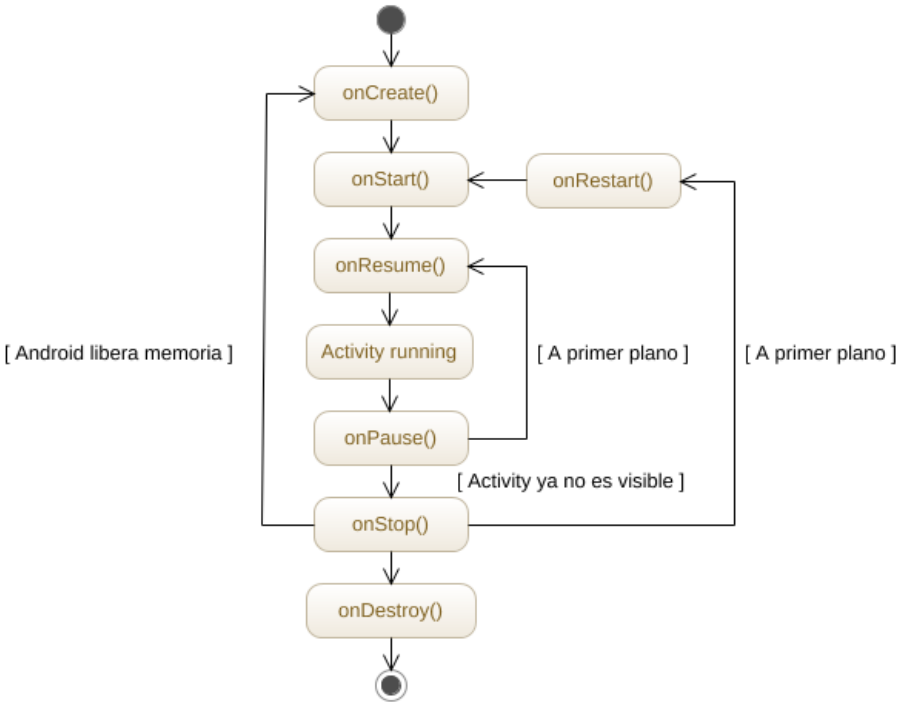

Figura 7 - Diagrama de estados de una Activity

Cuando se ordena crear una pantalla, se invoca el método *onCreate* de la *activity*. El controlador es quien debe especificar qué *layout* se mostrará al usuario haciendo uso del identificador del *layout*.

Retornando al ejemplo del título anterior, una *activity MyActivity* podría realizar una llamada al método *setContentView* junto con el identificador del *layout* para componer dicha pantalla:

```
public class MyActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.my_layout);
  }

}
```

Figura 8 - Activity aplicando una Vista/Layout a través de su identificador

5.4 Interfaz de usuario

La combinación de todas las vistas de comportamiento, derivadas de sus casos de uso, permite obtener una vista arquitectónica de alto nivel de la interfaz de usuario. Mediante el siguiente diagrama de estados, se describe la secuencia de actividades que puede tomar la aplicación en función de los eventos e interacción del usuario.

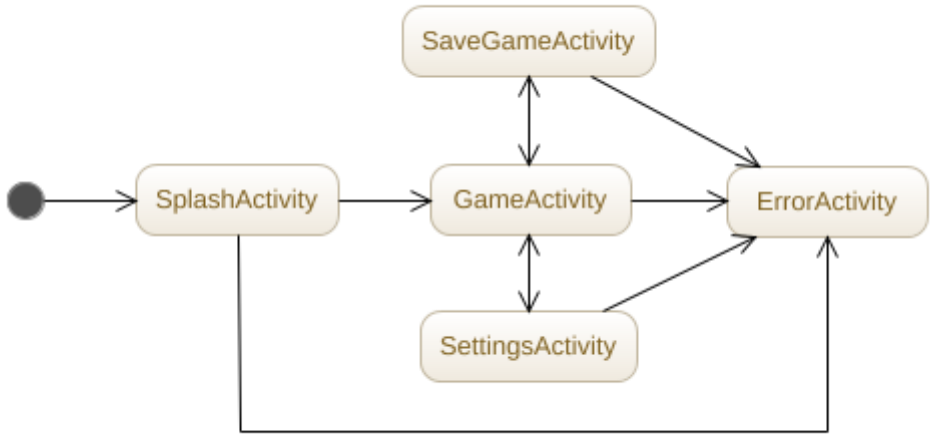

Figura 9 - Diagrama de estados de la interfaz de usuario

Las actividades implementadas son las siguientes:

- **SplashActivity**: es la pantalla de bienvenida que se lanza al iniciar la aplicación, cuya tarea es mostrar una imagen o logotipo durante un tiempo determinado.
- **GameActivity**: vista principal de la aplicación en donde se visualiza la pantalla de juego y los controles para interactuar con él.

- **SaveGameActivity**: es una vista dedicada a la gestión de las partidas guardadas por el usuario. Mediante una lista almacenada en disco, el usuario podrá guardar sus partidas y continuarlas más adelante.
- **SettingsActivity**: es la pantalla de ajustes de usuario en la aplicación, la cual se mantienen entre diferentes sesiones.
- **ErrorActivity**: es una vista conocida comúnmente como pantalla de la muerte o *BSOD*[17]. Su función es presentar de manera clara y descriptiva una situación de error que no pueda ser gestionada por la aplicación.

5.4.1 Jerarquía de clases

En la biblioteca de compatibilidad de Android se ofrecen varias implementaciones de *activities* que no vienen integradas por defecto. Estas versiones son compatibles con versiones anteriores del sistema, proporcionan nuevos elementos de interfaz y nuevas utilidades a las que pueden recurrir las aplicaciones.

Para mantener la compatibilidad con el mayor número de dispositivos y, a su vez, tener acceso a las características visuales de las versiones más recientes de Android, se hace uso de la clase base *AppCompatActivity*. El resto de clases presentes en el diagrama: *AbstractActivity*, *SplashActivity*, *GameActivity*, *SettingsActivity*, *SaveGameActivity* y *ErrorActivity*, son implementaciones propias del proyecto que más adelante se desarrollarán detalladamente.

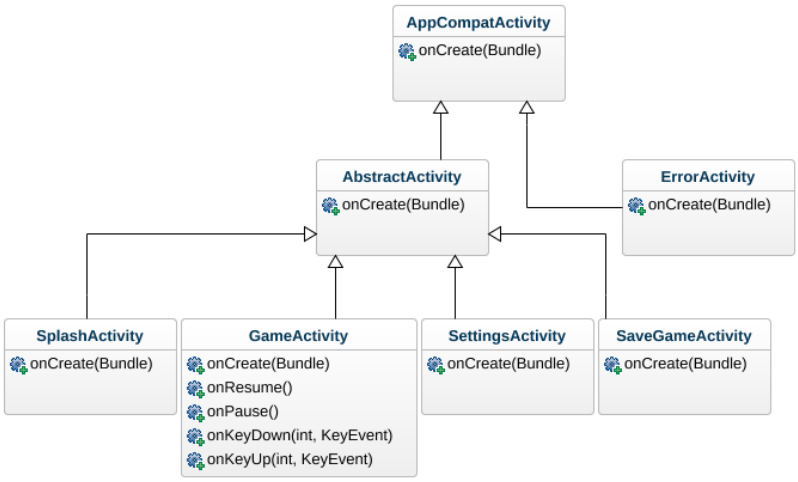

Figura 10 - Diagrama de clases acerca de la jerarquía de Activities

[17] El término *Blue Screen of Death*, también conocido por sus siglas BSOD, hace referencia a una pantalla que se muestra cuando un sistema no puede recuperarse tras un error.

5.4.2 AbstractActivity

Todas las actividades de la aplicación interactúan con diferentes componentes y son susceptibles de lanzar alguna excepción, ya sea porque la configuración no haya sido bien definida, se produzca un error de entrada/salida o no haya suficiente memoria disponible. Las actividades deben manejar aquellas excepciones que se produzcan en su contexto dentro de la lógica de negocio y delegar las que no a una instancia de mayor nivel.

Como ya se ha comentado en apartados anteriores, la propia naturaleza de las actividades en Android hace que puedan ser eliminadas de memoria tras un periodo de inactividad. Esto quiere decir que si la aplicación vuelve a ser restaurada, las *activities* deberán ser capaces de reinicializarse junto a todas sus dependencias.

En relación al proyecto, cabe mencionar que se hace uso del inyector de dependencias *Dagger*, en donde la inicialización e inyección de dependencias de una *activity* se produce en el método *onCreate*. Por lo tanto, si es la primera vez que se inicializa la aplicación o está siendo restaurada tras una liberación de memoria, se deberá preparar nuevamente todas las dependencias para proseguir con la *activity*. En ese caso, la *activity* está expuesta a lanzar una excepción en caso de darse una situación imprevista durante todo dicho proceso.

Para resolver esta problemática y a su vez centralizar en un solo lugar la gestión de *Exceptions*[18] entre actividades, se ha creado la clase *AbstractActivity*, la cual, además de contener lógica común entre actividades, instancia a nivel de *Thread*[19] un *ExceptionHandler* para trasladar el mensaje de error hacia la *activity ErrorActivity* y para su posterior presentación al usuario de una manera legible y amigable.

Con el fin de demostrar cómo se ha implementado, se adjunta a continuación un extracto de la implementación real de la clase *AbstractActivity*:

[18] Una *Exception* —en inglés, excepción— es la representación de un problema que ocurre durante la ejecución de un programa.

[19] Un *Thread* —en castellano, hilo— es una secuencia de instrucciones que forman una tarea y puede ser ejecutada al mismo tiempo que otra tarea.

```java
public abstract class AbstractActivity extends AppCompatActivity {

  @Override
  protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    new ExceptionHandler(this);
  }

  private static class ExceptionHandler implements
    Thread.UncaughtExceptionHandler {

    private final Activity context;

    ExceptionHandler(Activity context) {
      this.context = context;
      Thread.setDefaultUncaughtExceptionHandler(this);
    }

    @Override
    public void uncaughtException(final Exception exception) {
      Intent intent = new Intent(context, ErrorActivity.class);
      intent.putExtra(ERROR_DESCRIPTION, exception.getMessage());
      context.startActivity(intent);
      android.os.Process.killProcess(android.os.Process.myPid());
      System.exit(0);
    }
  }

  // Líneas suprimidas

}
```

Figura 11 - Extracto de la clase AbstractActivity para el manejo de excepciones

5.4.3 SplashActivity

La pantalla de Splash se lanza al iniciar la aplicación y permite mostrar una imagen o logotipo durante un tiempo determinado. Es un recurso estético con el que personalizar el diseño de la aplicación y destacar una marca o logotipo.

Para que pueda ser ajustada a las necesidades de cada desarrollador, la *activity* recibe una serie de parámetros que determinan su comportamiento y apariencia en tiempo de ejecución.

A continuación, se detallan los parámetros que pueden ser ajustados:

Parámetro	Valores	Descripción
enabled	Boolean: true / false	Activar o desactivar el splash. Valor por defecto: true.
background	String: #RRGGBB	Color de fondo en expresión RGB hexadecimal. Valor por defecto: #000000.
delay	Integer: milisegundos	Tiempo de duración de la pantalla en milisegundos. Valor por defecto: 3000.
orientation	String: portrait / landscape / auto	Orientación de la pantalla del splash. Valor por defecto: auto.

Figura 12 - Tabla de parámetros aceptados para la sección Splash

Mediante un fichero de configuración en formato *JSON*[20], que se tratará con más detalle en los siguientes capítulos, es posible personalizar y definir los aspectos más comunes de este tipo de pantallas.

```
{
  "splash": {
    "enabled": true,
    "background": "#eaeffd",
    "delay": 5000,
    "orientation": "auto"
  }
  # El resto de líneas han sido suprimidas
}
```

Figura 13 - Extracto del fichero de configuración de la aplicación

Para definir la vista, Android Studio provee una herramienta que permite diseñar de forma visual la composición de la misma. A continuación, la estructura de objetos se exporta a un fichero externo *XML,* el cual puede cargarse en tiempo de ejecución en el método *onCreate* de la *activity*.

[20] El formato *JavaScript Object Notation*, conocido por las siglas JSON, es una sintaxis para el almacenamiento e intercambio de datos.

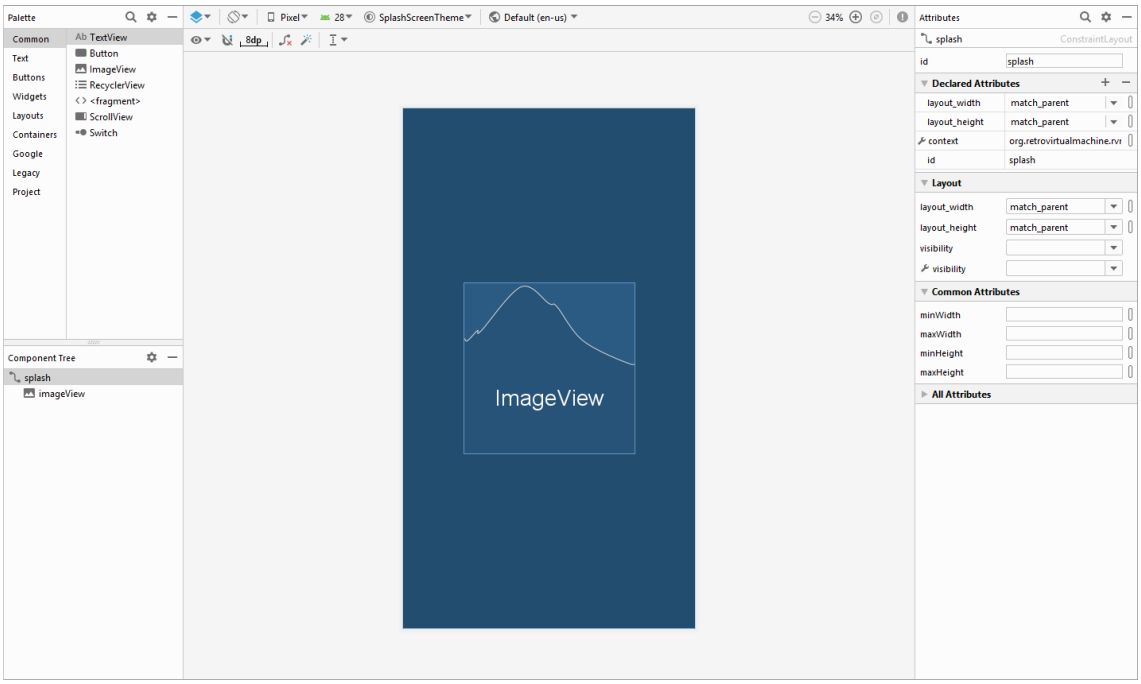

Figura 14 - Captura de Android Studio editando el layout «splash»

La imagen utilizada debe referenciarse como un recurso de imagen dentro de la aplicación. Es decir, por motivos de fragmentación entre tantos tipos de dispositivos, un recurso de imagen puede tener asociados varios ficheros de imagen por cada densidad de pantalla. Luego será la propia aplicación quien utilice una imagen u otra según las características del dispositivo donde esté siendo ejecutada.

Por ello, y como se detallará más adelante, será necesario aprovisionar la aplicación con al menos cuatro imágenes en los siguientes directorios para las diferentes densidades de pantallas definidas por Android:

- `/res/drawable-mdpi/splash.png`
- `/res/drawable-hdpi/splash.png`
- `/res/drawable-xhdpi/splash.png`
- `/res/drawable-xxhdpi/splash.png`

A continuación, para definir el controlar de la vista, es necesario crear una clase *SplashActivity* en la que se defina programáticamente su comportamiento.

Con el fin de demostrar cómo se ha implementado, se adjunta a continuación un extracto de la implementación real de la clase *SplashActivity*:

```
public class SplashActivity extends AbstractActivity {

  @Inject
  protected ActivityHelper activityHelper;
  @Inject
  protected Config config;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    AndroidInjection.inject(this);

    setContentView(R.layout.activity_splash);

    if (!config.getSplash().isEnabled()) {
      showGameActivity();
    } else {
      activityHelper.setOrientation(this, config.getSplash()
          .getOrientation());
      applyBackgroundColor(config.getSplash().getBackground());
      showGameActivityWithDelay(config.getSplash().getDelay());
    }
  }

  // El resto de líneas han sido suprimidas

}
```

Figura 15 - Extracto de la clase SplashActivity

Finalmente, al ejecutar la actividad en un dispositivo real, éste es el resultado:

Figura 16 - Resultado final ejecutado sobre un Nexus 5X

5.4.4 GameActivity

5.4.4.1 Introducción

Los controles de un juego suelen variar en función de la temática, el género o la propia creatividad del desarrollador. Una de las funcionalidades que considerábamos indispensables para que la aplicación cubriese diferentes casos de uso, debía ser la de incluir un medio para poder seleccionar y configurar distintas distribuciones de entrada.

En las plataformas móviles se han generalizado varios tipos de controles, ya sean mandos de juego táctiles, sistemas de un solo toque en la pantalla, control del juego a través del acelerómetro, etc.; por ello, la solución debía ser lo suficientemente flexible como para poder soportar diferentes controles de juego, pantalla en modo retrato o modo apaisado, y a su vez mostrarse capaz de recibir nuevas distribuciones en un futuro.

5.4.4.2 Sistema dinámico de Layouts

Aunque el sistema de *activities* que provee Android SDK es muy potente, no se ajusta por completo a las necesidades del proyecto. Para este caso, es necesario que, en función de un parámetro de configuración, la *activity* sea capaz de decidir qué *layout* ha de seleccionar y qué lógica aplicar.

Por ejemplo, un *layout* que presente un mando de juegos virtual requiere de una serie de componentes visuales y una lógica que controle las pulsaciones táctiles, mientras que, por otro lado, un *layout* que funcione a través del acelerómetro no requiere de ningún componente visual pero sí de una lógica espacial.

Por lo tanto, son implementaciones con una estructura y un comportamiento concreto de un concepto abstracto. Todas ellas necesitan ajustar parámetros y componentes de forma particular en la *activity* y exponer de forma común los elementos que la componen.

```
public interface Layout {

  void setup(Activity activity);

  EmulatorView getEmulatorView();

  List<View> getVirtualPadViews();

}
```

Figura 17 - Interfaz de un objeto Layout

Con vistas a cumplir con el principio *SOLID: Single Responsability*[21], la tarea de decidir qué implementación usar ha sido extraída de las *activities* y delegada en una factoría a través del patrón de diseño *Factory Pattern*.

La clase *LayoutFactory* tiene la responsabilidad de crear objetos *layout* sin exponer los detalles de implementación a las *activities*, poniendo en práctica el principio *SOLID: Liskov Substitution*[22].

A través de un parámetro dado por la configuración del desarrollador, se instanciará la implementación apropiada.

```
public class LayoutFactory {

  public Layout makeLayout(@Nonnull LayoutType layoutType) {
    switch (layoutType) {
      case GAME_PAD:
        return new GamePadLayout(...);
      case ONE_TOUCH:
        return new OneTouchLayout(...);
      case TWO_BUTTONS:
        return new TwoButtonsLayout(...);
      default:
        throw new UnsupportedOperationException();
    }
  }

}
```

Figura 18 - Extracto de la clase LayoutFactory

[21] El principio *SOLID de Single Responsability* establece que un objeto debe tener una única responsabilidad o una única razón para ser cambiada.

[22] El principio *SOLID de Liskov Substitution* establece que cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

Con el siguiente diseño, añadir nuevas implementaciones es tan sencillo como añadir una nueva clase a la factoría. Quedando, como resultado, un sistema de alta escalabilidad y mantenibilidad.

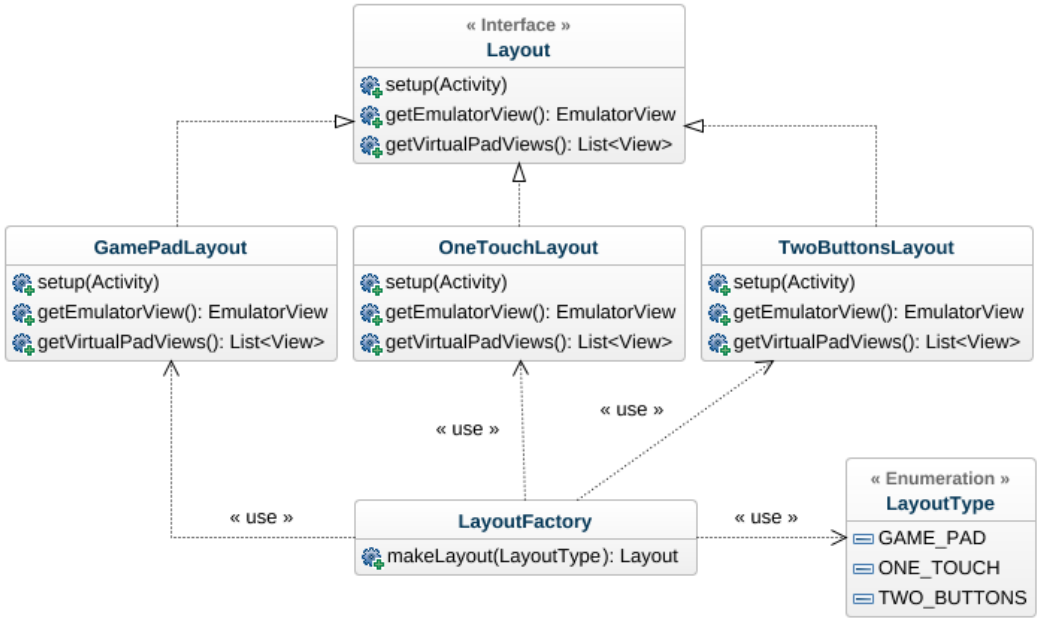

Figura 19 - Diagrama de clases para la creación dinámica de Layouts

A continuación, en el módulo de *ActivityModule* del Inyector de Dependencias Dagger, se define cómo ha de proveerse la dependencia *layout* en tiempo de ejecución; lo que cumple el principio *SOLID: Dependency Inversion*[23].

```
@Module
class ActivityModule {

    @Provides
    Layout provideLayout(Config config, LayoutFactory factory) {
        return factory.makeLayout(config.getLayout().getType());
    }

}
```

Figura 20 - Método encargado de proveer el Layout concreto

Finalmente, en la clase *GameActivity*, que continúa siendo la encargada de gestionar los eventos durante el ciclo de vida del juego, se inyectará el objeto *layout* apropiado y

[23] El principio *SOLID Dependency Inversion* establece que las clases de alto nivel no deberían depender de las clases de bajo nivel y las abstracciones no deberían depender de los detalles.

se invocará al método *setup*, para que aplique la estructura y lógica correspondiente definida por el desarrollador.

```
public class GameActivity extends AbstractActivity {

  @Inject
  protected Layout layout;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    AndroidInjection.inject(this);

    layout.setup(this);
  }

 // El resto de líneas han sido suprimidas

}
```

Figura 21 - Extracto de la clase GameActivity para el uso dinámico de Layouts

5.4.4.3 Ajuste del layout

En cuanto al ajuste de la distribución de pantalla a las necesidades de cada juego, el desarrollador tiene disponible una serie de parámetros para ajustar el *layout* desde el fichero de configuración.

A continuación, se detallan los parámetros que pueden ser ajustados:

Parámetro	Valores	Descripción
type	String: *gamepad* / *onetouch* / *twobuttons*	Selección del tipo de layout. Valor por defecto: gamepad.
orientation	String: *portrait* / *landscape* / *auto*	Orientación de la pantalla de juego. Valor por defecto: auto.

Figura 22 - Tabla de parámetros aceptados para la sección Layout

Seguidamente se presenta un ejemplo de configuración

```
{
  "layout": {
    "type": "gamepad",
    "orientation": "auto"
  }
  # El resto de líneas han sido suprimidas
}
```

Figura 23 - Extracto de la sección layout en el fichero de configuración

5.4.4.4 Mando virtual

Un clásico dentro de los controles para videojuegos es el mando compuesto por una cruceta de cuatro direcciones, cuatro botones de acción «**A**», «**B**», «**X**» e «**Y**», y dos botones auxiliares de «**SELECT**» y «**START**».

Figura 24 - Ejemplo del layout «gamepad» en un juego real

Para hacer uso de este *layout* es necesario especificar el tipo «**gamepad**» en el fichero de configuración. Las teclas del controlador virtual pueden ser mapeadas a una tecla de Amstrad CPC utilizando las claves reservadas para tal fin. Por otra parte, también es posible ocultar alguno de los botones mediante la palabra reservada «**none**».

A continuación se presenta un ejemplo de configuración

```
{
  "layout": {
    "type": "gamepad"
  },
  "keymap": {
    "up": "joy_up",
    "down": "joy_down",
    "left": "joy_left",
    "right": "joy_right",
    "a": "joy_fire1",
    "b": "none", // Botón desactivado
    "x": "none", // Botón desactivado
    "y": "joy_fire2",
    "start": "escape",
    "select": "none"// Botón desactivado
  }
  // El resto de líneas han sido suprimidas
}
```

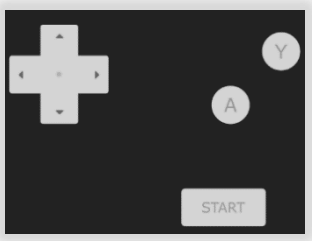

Figura 25 - Ejemplo de configuración del layout «gamepad»

Dentro de la definición *keymap* las claves definen los controles de la aplicación mientras que los valores la tecla de Amstrad CPC asociada. Todas las teclas disponibles pueden ser utilizadas conforman la siguiente tabla:

Alpha	Num	Char	Fn	Command	Joy
key_a	key_1	return	f1	clr	joy_up
key_b	key_2	escape	f2	cursor_up	joy_down
key_c	key_3	delete	f3	cursor_down	joy_left
key_d	key_4	tab	f4	cursor_left	joy_right
key_e	key_5	space	f5	cursor_right	joy_fire1
key_f	key_6	minus (-)	f6	enter_num	joy_fire2
key_g	key_7	caret (^)	f7	f_dot	
key_h	key_8	at (@)	f8	ctrl	
key_i	key_9	left_bracket ([)	f9	shift	
key_j	key_0	right_bracket (])	f0	copy	
key_k		semicolon (;)			
key_l		colon (:)			
key_m		backslash (\)			
key_n		comma (,)			

key_o		period (.)			
key_p		slash (/)			
key_q		caps_lock			
key_r					
key_s					
key_t					
key_u					
key_v					
key_w					
key_x					
key_y					
key_z					

Figura 26 - Tabla de claves reservadas para Amstrad CPC

Y finalmente, la implementación de la distribución se encuentra definida en la clase *GamePadLayout*:

```java
public class GamePadLayout implements Layout {

  private Config config;
  private EmulatorService emulatorService;
  private EmulatorView emulatorView;
  private List<View> virtualPadViews;

  public GamePadLayout(Config config, EmulatorService
      emulatorService) {
    this.config = config;
    this.emulatorService = emulatorService;
    this.virtualPadViews = new ArrayList<>();
  }

  @Override
  public void setup(Activity activity) {
    activity.setContentView(R.layout.activity_gamepad);

    emulatorView = activity.findViewById(R.id.emulator);
    GamePadView gamePadView = activity.findViewById(R.id.gamepad);
    GamePadButtonView aView = activity.findViewById(R.id.a);
    GamePadButtonView bView = activity.findViewById(R.id.b);
    GamePadButtonView xView = activity.findViewById(R.id.x);
    GamePadButtonView yView = activity.findViewById(R.id.y);
    GamePadButtonView startView =
        activity.findViewById(R.id.start);
    GamePadButtonView selectView =
        activity.findViewById(R.id.select);

    setupGamePad(gamePadView);
    setupButton(aView, config.getKeyMap().getA());
    setupButton(bView, config.getKeyMap().getB());
    setupButton(xView, config.getKeyMap().getX());
    setupButton(yView, config.getKeyMap().getY());
    setupButton(startView, config.getKeyMap().getStart());
    setupButton(selectView, config.getKeyMap().getSelect());
  }

  @Override
  public EmulatorView getEmulatorView() {
     return emulatorView;
  }

  @Override
  public List<View> getVirtualPadViews() {
    return virtualPadViews;
  }

  // El resto de líneas han sido suprimidas

}
```

Figura 27 - Extracto de la clase GamePadLayout

5.4.4.5 Un toque

Con la llegada de los teléfonos inteligentes y sus pantallas táctiles, son muchos los juegos que han optado por mecánicas sencillas a través de este tipo de interfaz de una sola pulsación. La pantalla se comporta como un botón **«A»** a través del cual el usuario realiza una determinada acción.

Figura 28 - Ejemplo del layout «onetouch» en un juego real

Para aplicar este *layout* basta con especificar el tipo «**onetouch**» en el fichero de configuración y vincular la tecla **«A»**, equivalente a la pulsación de la pantalla, a una tecla de Amstrad CPC.

```
{
  "layout": {
    "type": "onetouch"
  },
  "keymap": {
    "a": "joy_fire1"
  }
  // El resto de líneas han sido suprimidas
}
```

Figura 29 - Ejemplo de configuración del layout «onetouch»

En cuanto a la implementación, toda la lógica necesaria ha quedado encapsulada en la clase *OneTouchLayout*, tal y como se puede ver en el siguiente extracto:

```
public class OneTouchLayout implements Layout {

  private Config config;
  private EmulatorService emulatorService;
  private EmulatorView emulatorView;

  public OneTouchLayout(Config config, EmulatorService
      emulatorService) {
    this.config = config;
    this.emulatorService = emulatorService;
  }

  @Override
  public void setup(Activity activity) {
    activity.setContentView(R.layout.activity_onetouch);

    emulatorView = activity.findViewById(R.id.emulator);

    View parentView = activity.findViewById(R.id.one_touch);
    addListener(parentView, config.getKeyMap().getA());
  }

  @Override
  public EmulatorView getEmulatorView() {
    return emulatorView;
  }

  @Override
  public List<View> getVirtualPadViews() {
    return Collections.emptyList();
  }

  private void addListener(View parentView, final Key key) {
    parentView.setOnTouchListener(new View.OnTouchListener() {
      @Override
       public boolean onTouch(View view, MotionEvent event) {
         switch (event.getAction()) {
           case MotionEvent.ACTION_DOWN:
             emulatorService.keyDown(key);
             view.performHapticFeedback(
               HapticFeedbackConstants.VIRTUAL_KEY
             );
             break;

           case MotionEvent.ACTION_UP:
             emulatorService.keyUp(key);
             view.performClick();
             break;
          }
        return true;
      }
   });

}
```

Figura 30 - Extracto de la clase OneTouchLayout

5.4.4.6 Dos botones

Otra distribución bastante común en los juegos para dispositivos móviles es la compuesta por dos botones de acción.

Figura 31 - Ejemplo del layout «twobuttons» en un juego real

Para establecer esta distribución de pantalla es necesario especificar el tipo «**twobuttons**» en el fichero de configuración y vincular las teclas **«A» y «B»** como se muestra a continuación:

```
{
  "layout": {
    "type": "twobuttons"
  },
  "keymap": {
    "a": "joy_fire1",
    "b": "joy_fire2"
  }
  // El resto de líneas han sido suprimidas
}
```

Figura 32 - Ejemplo de configuración del layout «twobuttons»

Para la implementación de esta distribución se ha definido la clase *TwoButtonsLayout*:

```
public class TwoButtonsLayout implements Layout {

  private Config config;
  private EmulatorService emulatorService;
  private EmulatorView emulatorView;
  private List<View> virtualPadViews;

  public TwoButtonsLayout(Config config, EmulatorService
      emulatorService) {
    this.config = config;
    this.emulatorService = emulatorService;
    this.virtualPadViews = new ArrayList<>();
  }

  @Override
  public void setup(Activity activity) {
    activity.setContentView(R.layout.activity_twobuttons);

    emulatorView = activity.findViewById(R.id.emulator);
    GamePadButtonView aView = activity.findViewById(R.id.a);
    GamePadButtonView bView = activity.findViewById(R.id.b);

    setupButton(aView, config.getKeyMap().getA());
    setupButton(bView, config.getKeyMap().getB());
  }

  @Override
  public EmulatorView getEmulatorView() {
    return emulatorView;
  }

  @Override
  public List<View> getVirtualPadViews() {
    return virtualPadViews;
  }

  private void setupButton(final GamePadButtonView
      gamePadButtonView, final Key key) {
    gamePadButtonView.setOnClickListener(new
        GamePadButtonView.OnClickListener() {
          @Override
          public void onPress() {
            emulatorService.keyDown(key);
          }

          @Override
          public void onRelease() {
            emulatorService.keyUp(key);
          }
        });
       virtualPadViews.add(gamePadButtonView);
    }
  }

}
```

Figura 33 - Extracto de la clase TwoButtonsLayout

5.4.4.7 Dispositivos de entrada

Como ya se ha visto en el título anterior, los eventos producidos por interacción con el panel táctil se gestionan a nivel de *layout*. Cada objeto que compone nuestra interfaz puede implementar uno o varios *listener*s para dotarlos de un comportamiento específico en función del evento que se haya producido. Por el contrario, los eventos recibidos por un botón físico en el propio dispositivo, un teclado, o un controlador Bluetooth, deben ser gestionados a nivel de *activity*. En el caso que nos concierne, cuando un botón es presionado, los métodos *onKeyDown* y *onKeyUp* de la *activity* son invocados junto con el identificador del dispositivo y un código de tecla.

Para gestionar este tipo de eventos, los cuales pueden variar en función del tipo y modelo del dispositivo, se ha creado un servicio denominado *DeviceService*. En el capítulo dedicado a la capa de negocio se entrará en más detalles acerca de su funcionamiento e implementación. A nivel de *activity*, se interactúa con *DeviceService* para obtener la tecla mapeada entre el dispositivo y Amstrad; y a continuación, ésta es enviada al emulador a través del *EmulatorService*:

```
public class GameActivity extends AbstractActivity {

  @Inject
  protected DeviceService deviceService;
  @Inject
  protected EmulatorService emulatorService;

  @Override
  public boolean onKeyDown(int keyCode, KeyEvent event) {
    Key key = deviceService.getMappedKey(event.getDeviceId(),
        keyCode);
    emulatorService.keyDown(key);
    return false;
  }

  @Override
  public boolean onKeyUp(int keyCode, KeyEvent event) {
    Key key = deviceService.getMappedKey(event.getDeviceId(),
        keyCode);
    emulatorService.keyUp(key);
    return false;
  }

  // El resto de líneas han sido suprimidas

}
```

Figura 34 - Extracto de la clase GameActivity sobre eventos de teclado

5.4.4.8 Suspensión y restauración

Cuando una *activity* se encuentra en primer plano y el usuario bloquea el teléfono, recibe una llamada o presiona una tecla que envía la aplicación a un segundo plano; se dice que la *activity* entra en un estado suspensión denominado *Pause*. Del mismo modo, cuando la aplicación vuelve a un primer plano y se restaura la *activity*, se produce otro cambio de estado denominado *Resume*.

Para el funcionamiento de la aplicación son necesarios varios hilos de ejecución. Por un lado, tenemos al menos un hilo encargado de la *activity* y, por otro, uno a cargo de la instancia del emulador. Por lo tanto, cuando la *activity* pasa de un estado a otro, es necesario notificar al emulador de que debe detener o continuar con su ejecución.

Cuando una *activity* pasa por alguno de los estados, los métodos *onPause* y *onResume* son invocados respectivamente. Es por ello que a través del servicio *EmulatorService* se hace necesario controlar el estado del emulador tal como se muestra en el siguiente extracto:

```
public class GameActivity extends AbstractActivity {

  @Inject
  protected EmulatorService emulatorService;

  @Override
  protected void onResume() {
    super.onResume();
    applyUserRules();
    emulatorService.resume();
  }

  @Override
  protected void onPause() {
    super.onPause();
    emulatorService.pause();
  }

  // El resto de líneas han sido suprimidas

}
```

Figura 35 - Extracto de la clase GameActivity para la gestión de estados

5.4.5 SaveGameActivity

La pantalla presenta un archivo con los diferentes estados de juego almacenados por el usuario. La vista se compone de una lista de viñetas que agrupan una captura de pantalla, un pequeño memorándum y un par de botones para la carga o borrado de la partida. El usuario puede navegar por el listado, deslizando el dedo por la pantalla a través del eje vertical a modo de scroll.

En la parte inferior derecha se ubica un botón flotante en forma de disquete. Dicho botón es el encargado de invocar el procedimiento que toma la instantánea y persiste la partida en memoria. El número de partidas que pueden ser almacenadas queda limitado al espacio disponible en la memoria del teléfono. En el capítulo sobre persistencia de datos se explicará con más detalle acerca del diseño e implementación de este sistema de guardado.

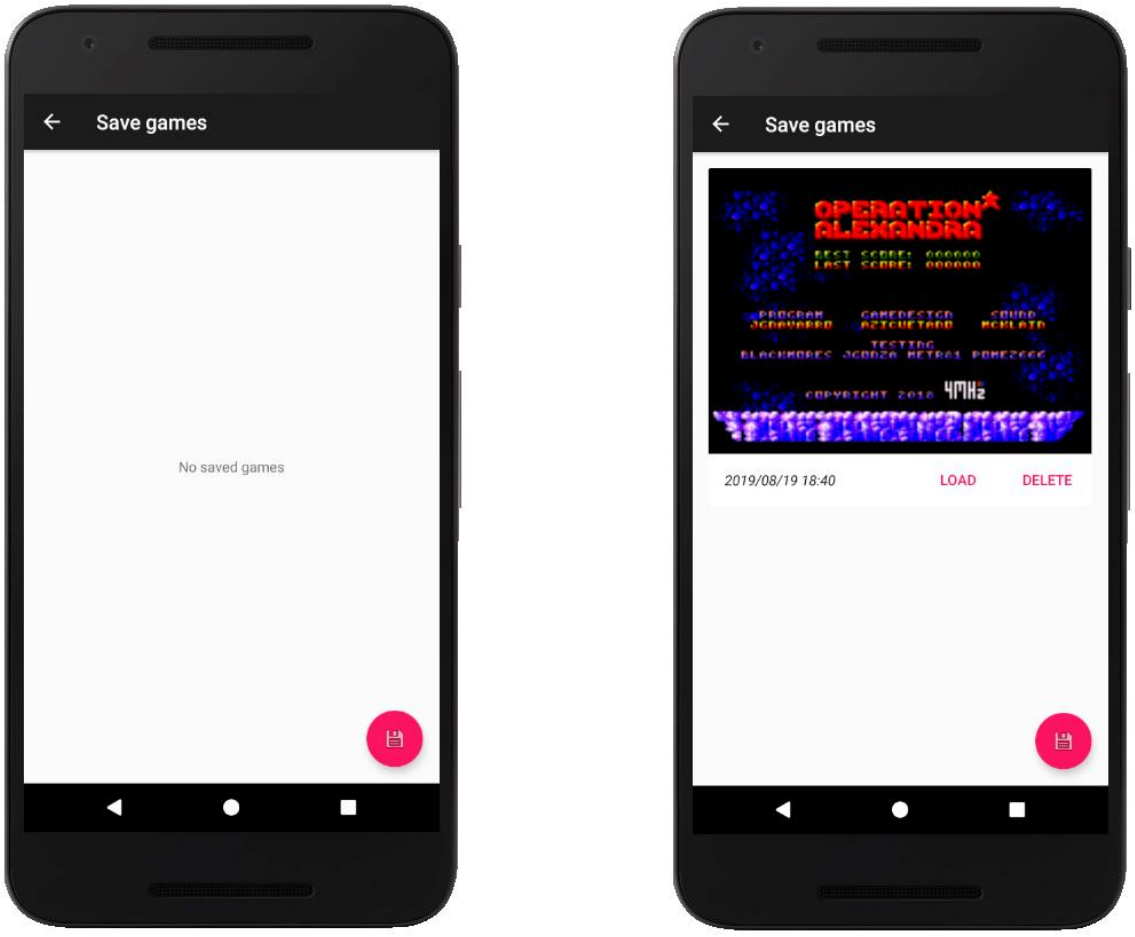

Figura 36 - Capturas de la activity SaveGames en diferentes estados

5.4.6 SettingsActivity

La pantalla dedicada a las preferencias permite configurar los ajustes de control y video de la aplicación. Aunque actualmente las opciones que ofrece son un tanto escasas, su diseño está preparado para poder escalar rápidamente y de forma muy sencilla.

Su implementación se encuentra basada en el framework de preferencias de Android. Cada preferencia de ajuste se define en una clase individual e independiente para su gestión y validación. A continuación, mediante un fichero *XML,* se estructura la interfaz y se ajustan los valores que darán lugar a la composición de la pantalla.

La propia *API* que ofrece Android se encarga de persistir las preferencias en la memoria del teléfono y mantener cada uno de los componentes al valor asociado seleccionado.

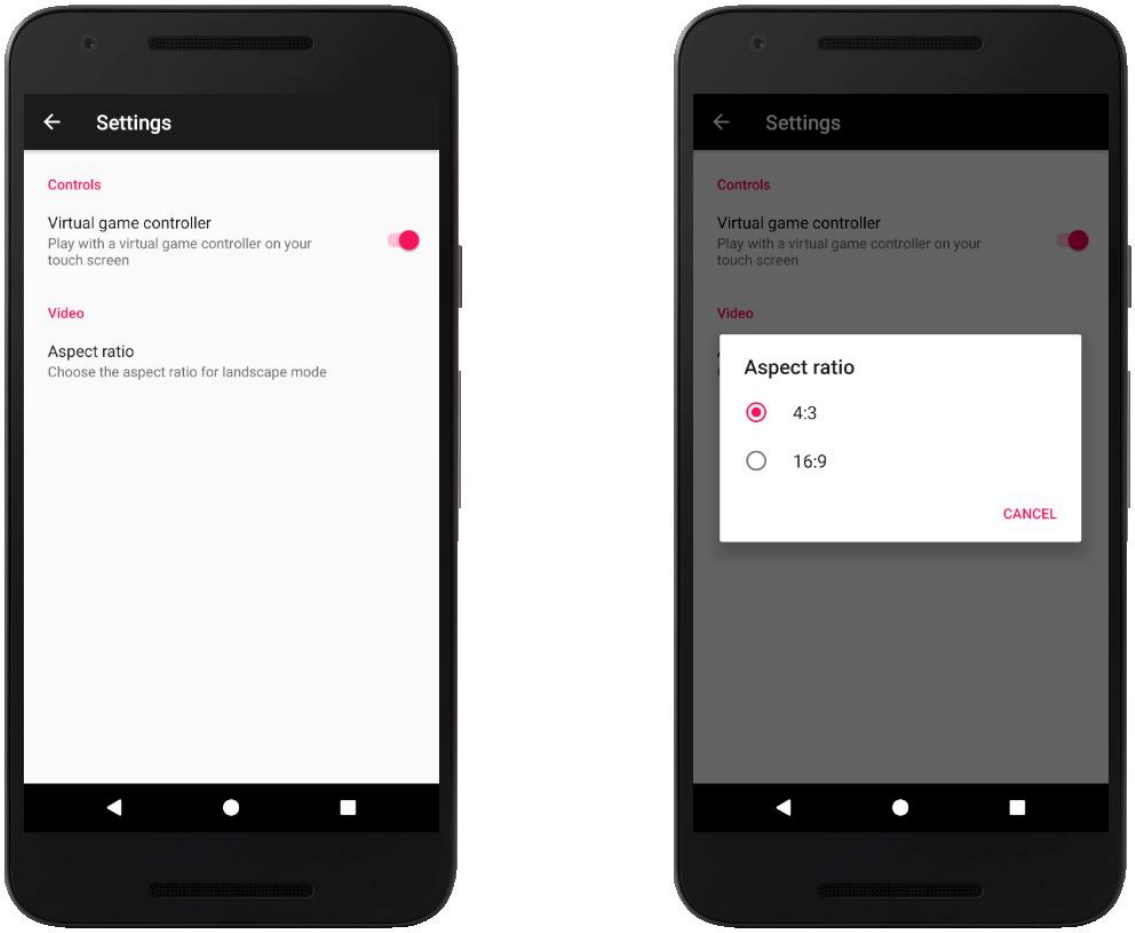

Figura 37 - Captura de pantalla de SettingsActivity

Para la implementación de la *activity* sólo es necesario seleccionar el *layout* y activar la barra superior de acción:

```
public class SettingsActivity extends AbstractActivity {

  @Inject
  protected ActivityHelper activityHelper;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    AndroidInjection.inject(this);
    setContentView(R.layout.activity_settings);
    activityHelper.setSupportActionBar(this);
  }

}
```

Figura 38 - Extracto de la clase SettingsActivity para la pantalla de ajustes

En el fichero `res/xml/settings.xml` se define la estructura de la pantalla y el origen de los diferentes valores:

```
<PreferenceScreen>

  <PreferenceCategory
      android:key="settings_controls"
      android:title="@string/settings_controls_title">

    <EnableVirtualPadPreference
        android:key="@enable_virtual_game_controller"
        android:title="@enable_virtual_game_controller_title"
        android:summary="@enable_virtual_game_controller_summary"/>

  </PreferenceCategory>

  <PreferenceCategory
      android:key="settings_video"
      android:title="@string/settings_video_title">

    <AspectRatioPreference
        android:key="aspect_ratio"
        android:title="@aspect_ratio_title"
        android:summary="@aspect_ratio_summary"
        android:defaultValue="4:3"
        android:entries="@aspect_ratio_entries"
        android:entryValues="@aspect_ratio_entries_values"/>

    </PreferenceCategory>

</PreferenceScreen>
```

Figura 39 - Extracto del fichero settings.xml para la composición de ajustes

En este caso, la *PrefrenceScreen* se encuentra dividida en dos *PreferenceCategory,* las cuales a su vez contienen, cada una de ellas, una preferencia *EnableVirtualPadPreference* y *AspectRatioPrefrence*. Cada preferencia se encuentra implementada por una clase *Java* con el mismo nombre, cuya clase base determina el tipo de comportamiento. Cuando se produce un cambio de preferencia, se dispara un *listener* que se encarga de tomar el nuevo valor y almacenarlo en la memoria del teléfono a través de la *API SharedPreference* de Android.

```
public class EnableVirtualPadPreference extends SwitchPreference {

  public static final String PREFERENCE_ID = "enableVirtualPad";
  public static final boolean DEFAULT_VALUE = true;

  public EnableVirtualPadPreference(Context context,
      AttributeSet attrs) {
    super(context, attrs);

    setDefaultValue(DEFAULT_VALUE);

    setOnPreferenceChangeListener(new OnPreferenceChangeListener(){
        @Override
        public boolean onPreferenceChange(Preference preference,
            Object newValue) {
          getSharedPreferences().edit()
              .putBoolean(PREFERENCE_ID, (boolean)newValue)
              .apply();
          return true;
        }
      });
  }

}
```

Figura 40 - Extracto de la clase EnableVirtualPadPreference

```
public class AspectRatioPreference extends ListPreference {

  public static final String PREFERENCE_ID = "aspectRatio";
  public static final String DEFAULT_VALUE = "4:3";

  public AspectRatioPreference(final Context context, AttributeSet
      attrs) {
    super(context, attrs);

    setDefaultValue(DEFAULT_VALUE);

    setOnPreferenceChangeListener(new OnPreferenceChangeListener(){
        @Override
        public boolean onPreferenceChange(Preference preference,
            Object newValue) {
          getSharedPreferences().edit()
              .putString(PREFERENCE_ID, (String)newValue)
              .apply();
          return true;
        }
      });
  }
}
```

Figura 41 - Extracto de la clase AspectRatioPreference

5.4.7 ErrorActivity

La pantalla de error se encarga de presentar de una forma comprensible aquellas situaciones en donde la aplicación es incapaz de recuperarse tras un error dentro del juego o a cause de una mala configuración del desarrollador.

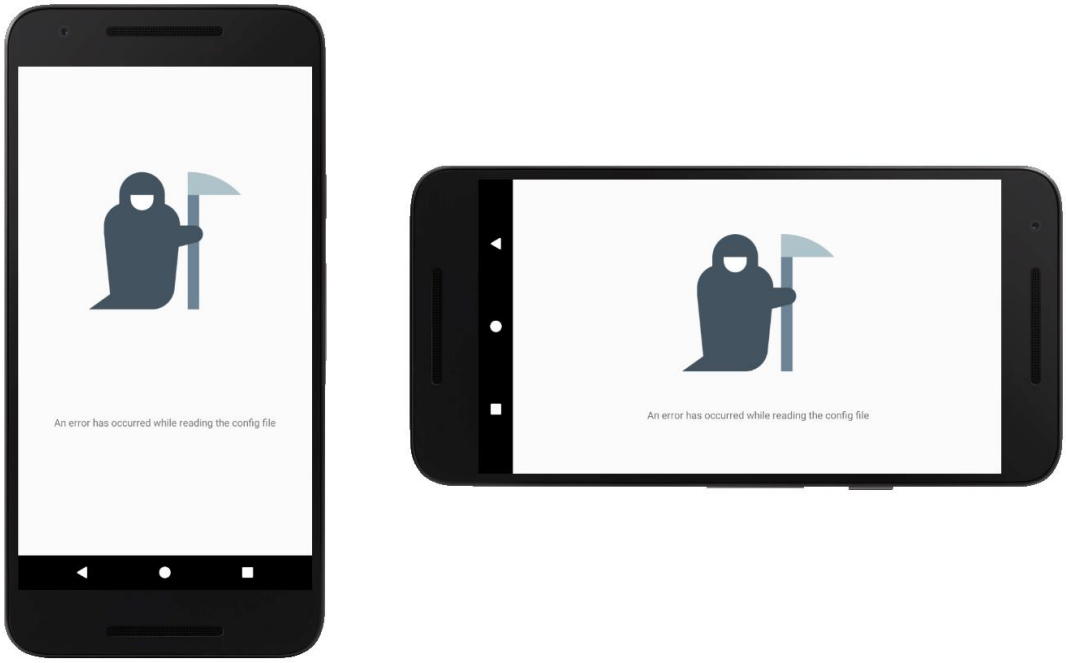

Figura 42 - Ejemplo de ErrorActivity por un error en el fichero de configuración

Su función es la de imprimir por pantalla un mensaje localizado en el contexto de la aplicación:

```
public class ErrorActivity extends AppCompatActivity {

  public static final String ERROR_DESCRIPTION = "description";

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_error);

    String desc = getIntent().getStringExtra(ERROR_DESCRIPTION)
    if (desc != null) {
      TextView textView = findViewById(R.id.text_description);
      textView.setText(desc);
    }
  }

}
```

Figura 43 - Extracto de la clase ErrorActivity para la gestión de errores

6 Capa de negocio

6.1 Emulación

La gestión del motor de emulación es delegado a un servicio denominado *EmulatorService*. La lógica de emulación queda totalmente encapsulada dentro de este componente, con lo que las *activities* que requieran interactuar de alguna manera con la emulación sólo tendrán que invocar al método correspondiente.

Las llamadas desde el servicio hacia *Retro Virtual Machine* se realizan sobre la *Java Native Interface*, la cual permite que un programa escrito en *Java* pueda interactuar con programas escritos en *C*, *C++* o *ensamblador*.

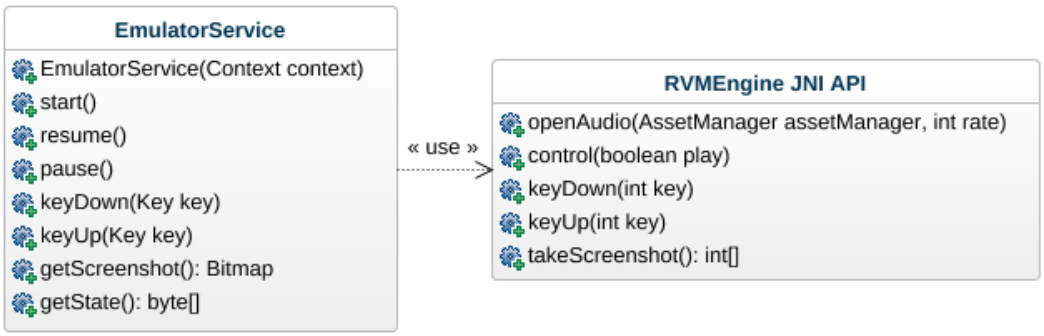

Figura 44 - Diagrama de clases de EmulatorService

6.2 Dispositivos

El componente *DeviceService* encapsula la lógica para la gestión de dispositivos y provee una interfaz de aplicación con la que interactuar con ellos. Aunque actualmente las funcionalidades con dispositivos son muy reducidas, la aplicación es capaz de gestionar las entradas provenientes desde un teclado o un mando de juegos.

Para ellos se ha introducido el concepto de *DeviceAdapter*. Como su propio nombre indica, se trata de un adaptador que traduce el *KeyCode* —el número asignado a la tecla en un dispositivo— a su equivalente en Amstrad CPC.

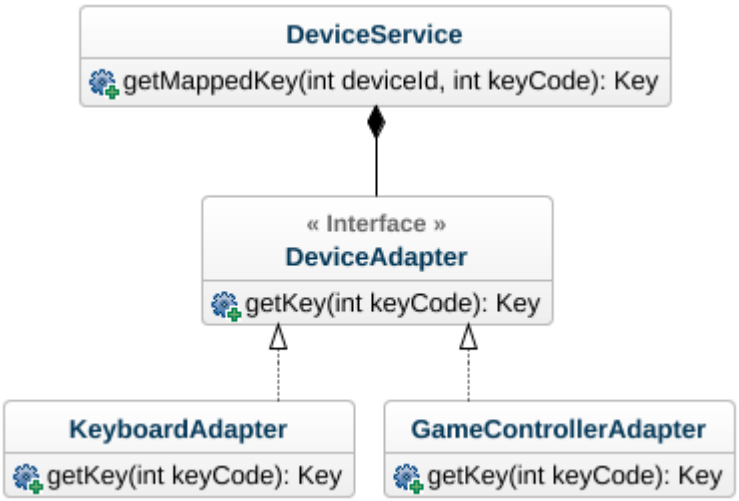

Figura 45 - Diagrama de clases de DeviceService

Las *activities* son las encargadas de gestionar la entrada de pulsaciones desde dispositivos externos. A través del servicio *DeviceService* se expone el método *getMappedKey*, el cual utiliza el *identificador de dispositivo* para seleccionar el adaptador apropiado y aplicar el mapeado de teclas.

```
public class DeviceService {

  @Nullable
  public Key getMappedKey(int deviceId, int keyCode) {
    DeviceAdapter adapter = getDeviceAdapter(deviceId);

    if (adapter != null) {
      return adapter.getKey(keyCode);
    }

    return null;
  }

  private DeviceAdapter getDeviceAdapter(int deviceId) {
    int sources = InputDevice.getDevice(deviceId).getSources();
    if (isGameController(sources)) {
      return gameControllerAdapter;
    } else if (isKeyboard(sources)) {
      return keyboardAdapter;
    } else {
      return null;
    }
  }

  // El resto de líneas han sido suprimidas

}
```

Figura 46 - Extracto de la clase DeviceService sobre selección de adaptador

La aplicación se comunica a través de la *API* de Android con dispositivos externos conectados por cable o vía inalámbrica. El mapeado de teclas, es decir, la asociación entre las teclas y/o botones de dispositivo externo a una tecla de la máquina Amstrad CPC, se realiza por defecto.

Figura 47 - Asociación de teclas del mando de PS4 por defecto

```
public class GameControllerAdapter implements DeviceAdapter {

  private final SparseArray<Key> defaultKeys = new SparseArray<>();

  public GameControllerAdapter(Config config) {
    configDefaultKeys(config.getKeyMap());
  }

  private void configDefaultKeys(Config.KeyMap keymap) {
    defaultKeys.put(KeyEvent.KEYCODE_DPAD_UP, keymap.getUp());
    defaultKeys.put(KeyEvent.KEYCODE_DPAD_LEFT, keymap.getLeft());
    defaultKeys.put(KeyEvent.KEYCODE_DPAD_RIGHT,
        keymap.getRight());
    defaultKeys.put(KeyEvent.KEYCODE_DPAD_DOWN, keymap.getDown());
    defaultKeys.put(KeyEvent.KEYCODE_BUTTON_A, keymap.getA());
    defaultKeys.put(KeyEvent.KEYCODE_BUTTON_B, keymap.getB());
    defaultKeys.put(KeyEvent.KEYCODE_BUTTON_X, keymap.getX());
    defaultKeys.put(KeyEvent.KEYCODE_BUTTON_Y, keymap.getY());
    defaultKeys.put(KeyEvent.KEYCODE_BUTTON_START,
        keymap.getStart());
    defaultKeys.put(KeyEvent.KEYCODE_BUTTON_SELECT,
        keymap.getSelect());
    }

    // El resto de líneas han sido suprimidas

}
```

Figura 48 - Extracto de la clase GameControllerAdapter para el control de teclas

```
public class KeyboardAdapter implements DeviceAdapter {

  private final SparseArray<Key> defaultKeys = new SparseArray<>();

  public KeyboardAdapter() {
    configDefaultKeys();
  }

  @Override
  public Key getKey(int keyCode) {
    return defaultKeys.get(keyCode);
  }

  private void configDefaultKeys() {
    defaultKeys.put(KeyEvent.KEYCODE_A, Key.KEY_A);
    defaultKeys.put(KeyEvent.KEYCODE_B, Key.KEY_B);
    defaultKeys.put(KeyEvent.KEYCODE_C, Key.KEY_C);
    defaultKeys.put(KeyEvent.KEYCODE_D, Key.KEY_D);

    // El resto de líneas han sido suprimidas

  }
}
```

Figura 49 - Extracto de la clase KeyboardAdapter sobre el mapeado de teclas

7 Capa de datos

7.1 Configuración

La configuración de la aplicación es parametrizada a través de un fichero *JSON*. Los desarrolladores pueden ajustar diferentes propiedades en cada una de las principales partes de la aplicación con el fin de adaptarlas a sus necesidades.

Una vez se inicia la aplicación, el fichero de configuración *JSON* es mapeado sobre una clase *Java* haciendo uso de la biblioteca *Jackson*. Los valores que haya introducido el desarrollador serán validados y el resto serán asignados con valores por defecto.

Finalmente, el objeto resultante será inyectado a cada uno de los componentes que los requiera para que puedan construirse acorde a los parámetros recibidos.

```
public class Config {

  private Splash splash;
  private Layout layout;

  @JsonProperty("keymap")
  private KeyMap keymap;
  private Video video;

  public static class Splash {
    private boolean enabled = true;
    private String background = "#000000";
    private Integer delay = 3000;
    private Orientation orientation = Orientation.AUTO;
  }

  // El resto de líneas han sido suprimidas

}
```

Figura 50 - Extracto de la clase Config sobre el que se mapea un fichero JSON

7.2 Parámetros

Con el objetivo de centralizar y definir los diferentes tipos de valores que pueden ser utilizados en la aplicación, se han definido una serie de enumerados.

El enumerado *Key* define cada una de las teclas de entrada disponibles *Amstrad CPC*. Al mismo tiempo establece la palabra reservada a la que hacer referencia desde

el fichero de configuración *JSON* y el valor numérico requerido por *Retro Virtual Machine*:

```
public enum Key {
  @JsonProperty("none") NONE(0),
  @JsonProperty("key_a") KEY_A(4),
  @JsonProperty("key_b") KEY_B(5),
  @JsonProperty("key_c") KEY_C(6),
  @JsonProperty("key_d") KEY_D(7),
  @JsonProperty("key_e") KEY_E(8),
  @JsonProperty("key_f") KEY_F(9),

  // ...

  @JsonProperty("joy_up") JOY_UP(250),
  @JsonProperty("joy_down") JOY_DOWN(251),
  @JsonProperty("joy_left") JOY_LEFT(252),
  @JsonProperty("joy_right") JOY_RIGHT(253),
  @JsonProperty("joy_fire1") JOY_FIRE1(254),
  @JsonProperty("joy_fire2") JOY_FIRE2(255);

  // El resto de líneas han sido suprimidas
}
```

Figura 51 - Extracto del enum Key sobre la definición de teclas de Amstrad CPC

Mediante el enumerado *Orientation* se definen las diferentes orientaciones del dispositivo:

```
public enum Orientation {
  @JsonProperty("auto")
  AUTO,
  @JsonProperty("landscape")
  LANDSCAPE,
  @JsonProperty("portrait")
  PORTRAIT
}
```

Figura 52 - Extracto del enum Orientation para orientaciones de dispositivos

Y con *LayoutType* los diferentes tipos de *layout*:

```
public enum LayoutType {
  @JsonProperty("gamepad")
  GAME_PAD,
  @JsonProperty("onetouch")
  ONE_TOUCH,
  @JsonProperty("twobuttons")
  TWO_BUTTONS;
}
```

Figura 53 - Extracto del enumerado LayoutType sobre los diferentes layouts

7.3 Partidas guardadas

Aunque la característica para partidas guardadas no está del todo implementada debido a que el engine de Retro Virtual Machine todavía no lo tiene del todo operativo, el diseño para el sistema de guardado ya se encuentra planteado. La *activity* se comunica mediante un repositorio para realizar las operaciones básicas CRUD[24].

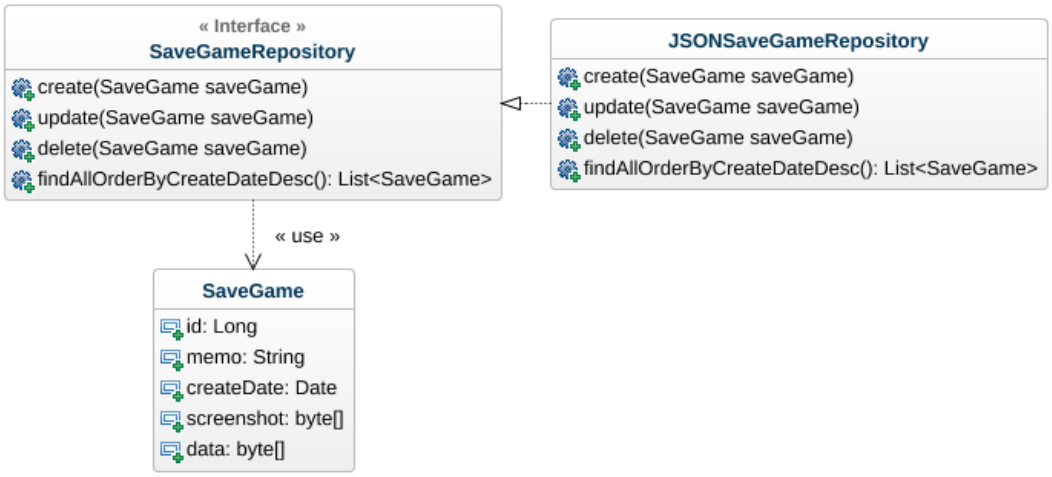

Figura 54 - Diagrama de clases sobre la implementación del repositorio

```
public class SaveGame {

  private Long id;
  private String memo;
  private Date createDate;
  private byte[] screenshot;
  private byte[] data;

  // El resto de líneas han sido suprimidas
}
```

Figura 55 - Extracto de la clase SaveGame para la persistencia de partidas

[24] CRUD es el acrónimo de las operaciones *Create*, *Read*, *Update* y *Delete* para referirse a las funciones básicas en bases de datos o la capa de persistencia en un software.

8 Integración en CPCTelera

8.1 Introducción

La ejecución de tareas automatizadas dentro de CPCTelera se realiza a través de ficheros *makefile*. Este tipo de archivos definen un conjunto de tareas a ejecutar a través de una serie de variables y reglas. Mediante la invocación de diferentes comandos de forma secuencial, que se explicará a continuación, CPCTelera es capaz de construir la aplicación que posteriormente podrá ser distribuida en teléfono Android.

8.2 Procedimiento

En Android las aplicaciones se distribuyen en un único fichero conocido comúnmente como *APK*. Los ficheros *APK* no son más que contenedores del código de la aplicación y sus recursos. Como ocurre con los paquetes *JAR* de *Java*, resultan ser un fichero en formato *ZIP*, con la extensión renombrada a **.apk* y a los que se le aplica una firma digital.

Su estructura es la siguiente:

Figura 56 - Componentes destacables que forman un fichero APK

Fichero	Descripción
AndroidManisfest.xml	Archivo “Manifest” en formato XML binario
META-INF/	Directorio que contiene el fichero MANIFEST.MF, el cual almacena meta-datos sobre el contenido del paquete. La firma del paquete también se almacena en este directorio.

assets	Directorio que contiene los assets de la aplicación y que pueden ser gestionados desde el AssetManager
lib/	Directorio que contiene el código de bibliotecas compiladas
res/	Directorio que contiene los recursos no compilados en resources.arsc
resources.arsc	Fichero que contiene todos los recursos pre-compilados de la aplicación en formato XML binario
cases.dex	Fichero que contiene las clases pre-compiladas de la aplicación en formato XML binario

Figura 57 - Tabla con la descripción de ficheros destacables en un fichero APK

En definitiva, son ficheros que pueden manipularse con cualquier gestor de paquetes *ZIP*. Los recursos a reemplazar son accesibles y manipulables en los directorios `assets` y `res`.

```
$ unzip -j game.apk
```

Figura 58 - Comando para extraer contenido del fichero APK con unzip

En un primer momento, parecía viable abrir el paquete, reemplazar los recursos que fueran necesarios y volver a empaquetarlo. Si este proceso fuese factible, con distribuir una versión pre-compilada de la aplicación sería suficiente, y evitaríamos así recurrir a Android SDK y su conjunto de herramientas para exportar el proyecto.

Sin embargo, este proceso no fue suficiente. Muchos de los recursos que requieren ser actualizados para generar la aplicación se encuentran compilados o convertidos en una versión binaria por la que difícilmente pueden ser manipulados. Para solventarlo, hice uso de la herramienta de ingeniería inversa *apktool* instrumento que permite extraer y construir paquetes apk de manera muy sencilla a través de la línea de comandos.

```
$ apktool decode game.apk –f –o path/ (Extraer)
$ apktool build game.apk–o path/ (Empaquetar)
```

Figure 59 - Comandos para manipular ficheros APK con apktool

Tan pronto como lo probé, me encontré con el primer problema. Si modificaba el contenido del fichero, la firma asociada ya no era válida y la aplicación no podía instalarse. Por lo tanto, me pareció que el siguiente paso lógico sería volver a firmar la aplicación tras volver a re-empaquetarla.

Al consultar la documentación oficial, pude comprobar que el firmado de un fichero *APK* podía hacerse de varias formas. La primera, a través del propio *AndroidSDK*, ya fuera a través *Android Studio* o su herramienta de línea de comandos *apksigner*, o bien, mediante *jarsigner*, distribuido en *OpenJDK* u *OracleJDK*.

Dado que quería evitar la dependencia con *AndroidSDK*, puesto que de todas las alternativas es la más pesada y molesta de instalar, elegí la opción de utilizar *jarsigner*.

```
$ jarsigner -sigalg SHA1withRSA -digestalg SHA1 game.apk
  androidkey
```

Figura 60 - Comando para firmar un fichero APK con jarsigner

Por otro lado, por una cuestión de optimización, los ficheros *APK* deben ser alineados para garantizar que todos los datos descomprimidos comienzan con una alineación de *bytes* en particular, relacionada con el comienzo del archivo, lo que reduce el consumo de memoria *RAM* de la aplicación.

Para alinear los ficheros *ZIP* puede utilizarse la herramienta *zipalign*, la cual se distribuye libremente, separada o como parte de *Android SDK*.

```
  $ zipalign -f -v 4 game.apk game.aligned.apk
```

Figura 61 - Comando para alinear un fichero APK con zipalign

Finalmente, el fichero *APK* se modificó correctamente y pude completar la instalación de la aplicación. De este modo, determiné las tareas y las herramientas necesarias para su automatización.

9 Ejemplo de uso

9.1 Introducción

Dado que el propósito de este capítulo no es crear un videojuego y requerimos de uno para poner en prácticas este proyecto, haremos uso del juego de ejemplo *Platform Climber,* incluido en CPCTelera. Este proyecto tiene por objeto ser un modelo de referencia para los usuarios y un punto de partida a la hora de hacer pruebas, demostraciones de la herramienta.

9.2 Instalación de CPCTelera

En primer lugar, es necesario acceder el repositorio oficial de CPCTelera en la siguiente URL: https://github.com/lronaldo/cpctelera. Al pulsar sobre el botón «**Clone or download**», un diálogo indica la URL necesaria para clonar el proyecto vía HTTPS.

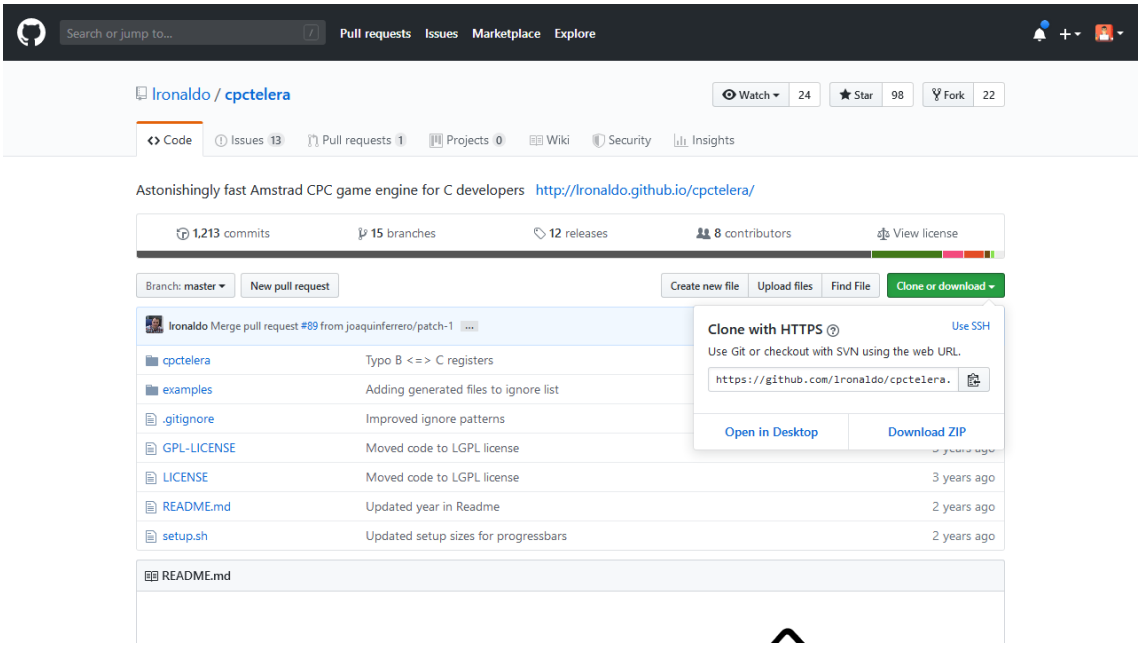

Figura 62 - Captura de pantalla del repositorio en GitHub de CPCTelera

A continuación, desde una terminal, lanzamos la orden *git clone* junto a la *URL* obtenida en el paso anterior. Al cabo de unos poco segundos, una copia local del repositorio se encontrará en el entorno local CPCTelera.

```
$ git clone https://github.com/lronaldo/cpctelera.git
```

Figura 63 - Comando para clonar el proyecto con git

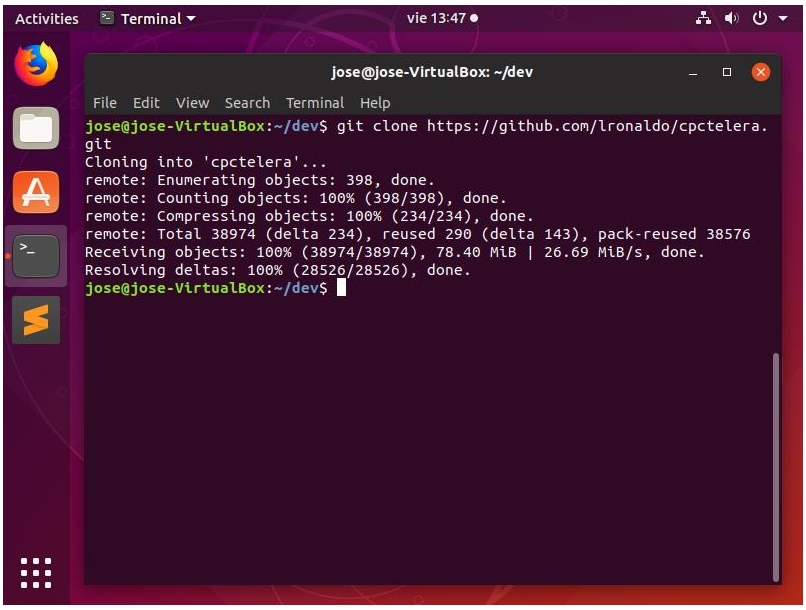

Figura 64 - Captura de pantalla tras clonar el repositorio de CPCtelera

Dado que esta nueva característica todavía no ha sido integrada en la rama estable de CPCTelera, es requisito indispensable hacer un cambio a la rama de android. Para ello, es necesario ubicarse en la raíz del proyecto y hacer el cambio con la siguiente orden:

```
$ cd cpctelera
$ git checkout –t origin/android
```

Figura 65 - Comando para cambiar a la rama android con git

A continuación, proceder a instalar CPCTelera en el sistema:

```
$ ./setup.sh
```

Figura 66 - Comando para instalar CPCPTelera

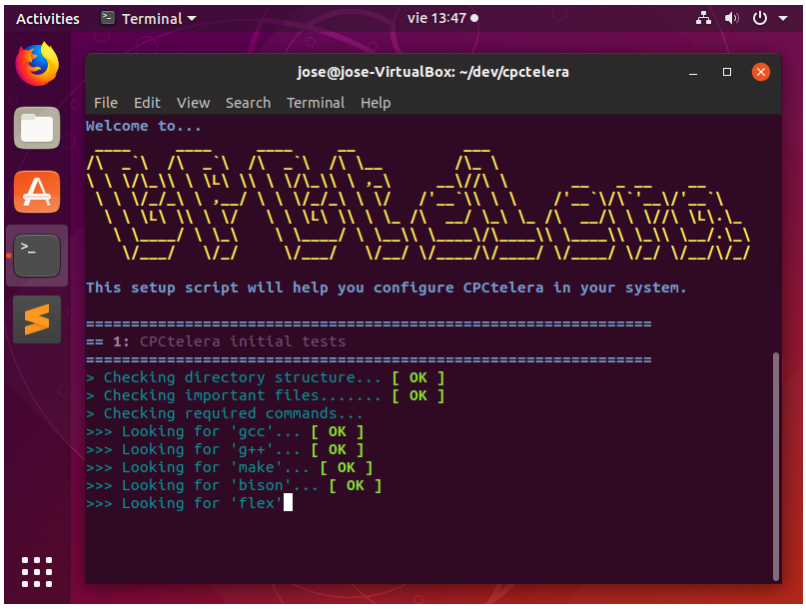

Figura 67 - Captura de pantalla durante el proceso de instalación de CPCTelera

Durante este proceso, se verifica que el equipo cuenta con todas las dependencias para funcionar. El proceso puede llevar algunos minutos de duración, pero pasado un tiempo, y si todo fue bien, se muestra un mensaje indicando que CPCTelera está listo para ser usado en el sistema.

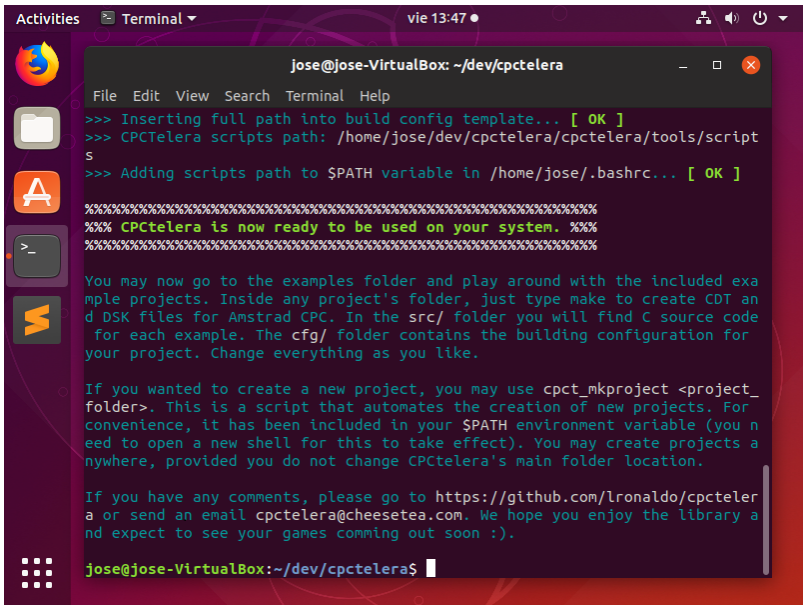

Figura 68 - Captura de pantalla tras una instalación satisfactoria de CPCTelera

9.3 Configuración y compilación del proyecto

Una vez instalado CPCTelera es necesario desplazarse al directorio raíz del proyecto de ejemplo PlatformClimber. Desde la misma terminal, basta con ejecutar la siguiente orden en la línea de comandos:

```
$ cd examples/games/pclimberAnd
```

Figura 69 - Comando para desplazarse al directorio del proyecto

A continuación, ha de compilarse el juego y generar las imágenes DSK, CDT y SNA. Cabe recordar que la versión del juego para Android utiliza la imagen SNA como estado inicial del juego en el emulador:

```
$ make
```

Figura 70 - Comando para compilar el proyecto con make

Como se puede observar en la siguiente imagen, el proceso de compilación ha de terminar indicando que ha podido crear de forma satisfactoria el fichero `game.sna`.

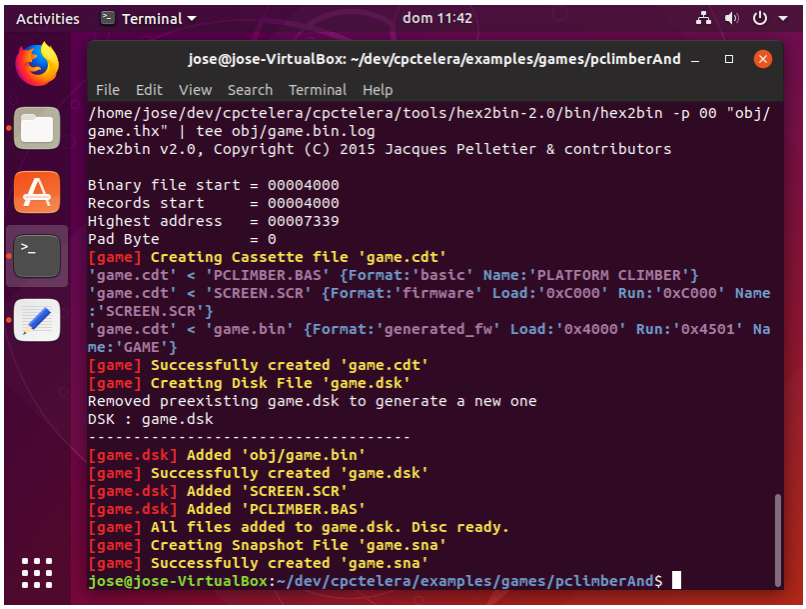

Figura 71 - Captura de pantalla tras la compilación del proyecto

Seguidamente se deben ajustar los parámetros y comportamientos de la aplicación a través del fichero de configuración `config.json`. Un editor de texto es suficiente para editar el fichero que se ubica desde la propia raíz del proyecto en el directorio `exp/android/assets`.

```
$ gedit exp/android/assets/config.json
```

Figura 72 - Comando para editar del fichero de configuración con gedit

A continuación, cada parámetro de configuración debe ajustarse en función del comportamiento deseado, tal y como se ha detallado en los diferentes capítulos del presente documento:

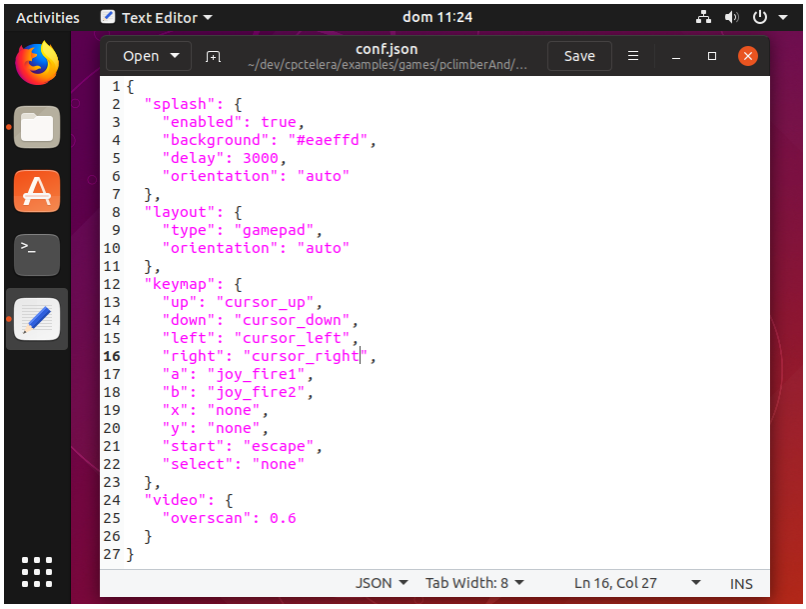

Figura 73 - Captura de pantalla mientras se edita el fichero de configuración

Por otro lado, se brinda la posibilidad de personalizar diferentes tipos de recursos utilizados en la aplicación. Los iconos, la imagen de splash, las texturas aplicadas a botones, etc. Cada densidad de pantalla tiene un directorio dedicado para cada uno de estos recursos. Mediante un explorador de archivos es posible acceder a la ruta de proyecto `exp/android/res` para editar y manipular los recursos que serán utilizados en la aplicación.

```
$ nautilus -w exp/android/res
```

Figura 74 - Comando para explorar el directorio de recursos con nautilus

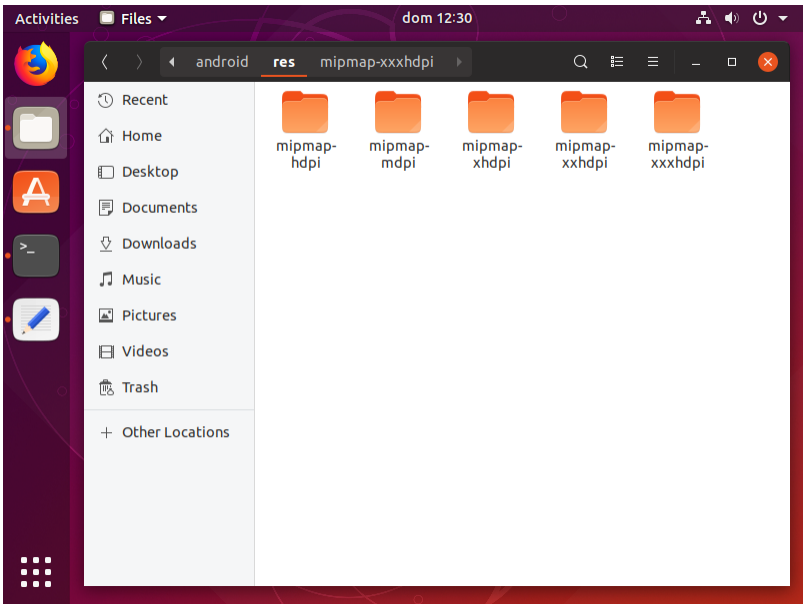

Figura 75 - Captura de pantalla mientras se explora el directorio de recursos

Otro punto muy importante a tener en cuenta es que la asignación requiere de un identificador de aplicación. El identificador debe ser único para cada aplicación distribuida en Google Play o en cualquier teléfono móvil. Es la forma que tiene Android de distinguir una aplicación de otra. Por lo tanto, el identificador debe ser definido una única vez y mantenerse en compilaciones posteriores. De lo contrario, las compilaciones posteriores serán consideradas para Android como una aplicación diferente. Al mismo tiempo, la aplicación requiere de un nombre que se mostrará a los usuarios en los menús y ajustes de aplicaciones instaladas. En este caso, el nombre no tiene restricciones y puedes cambiarse en todo momento. No tiene ninguna implicación en Google Play o en cómo reconoce Android la aplicación. Mediante un editor de textos editaremos el fichero `cfg/export/android.mk`.

```
$ gedit cfg/export/android.mk
```

Figura 76 - Comando para editar el fichero de construcción con gedit

A continuación, buscaremos a lo largo del fichero las variables *GAME_ID* y *GAME_NAME* para seguidamente definir sus valores acorde a los parámetros anteriormente mencionados:

```
GAME_ID   := jluri.platformclimber
GAME_NAME := Ejemplo Platform Climber
```

Figura 77 - Líneas que requieren ser modificadas durante la configuración

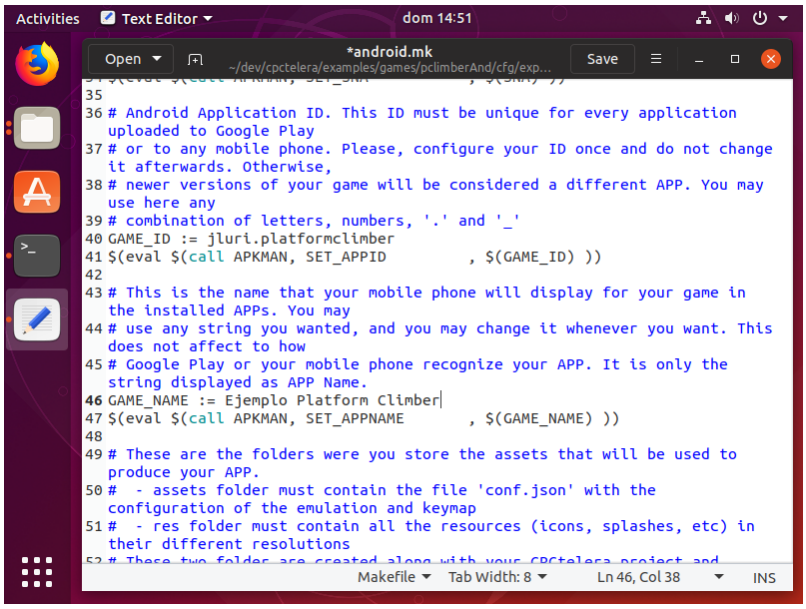

Figura 78 - Captura de pantalla mientras se configura la aplicación

Una vez se hayan completado todos los pasos anteriores, desde la raíz del proyecto procederemos a generar la aplicación para Android a través de la orden *make apk*. Tras finalizar el proceso, se habrá generado un fichero `game.apk`, el cual contendrá el juego compilado y totalmente funcional para plataformas Android.

```
$ gedit cfg/export/android.mk
```

Figura 79 - Comando para editar el fichero de construcción con gedit

```
$ make apk
```

Figura 80 - Comando para la compilación de la aplicación con make

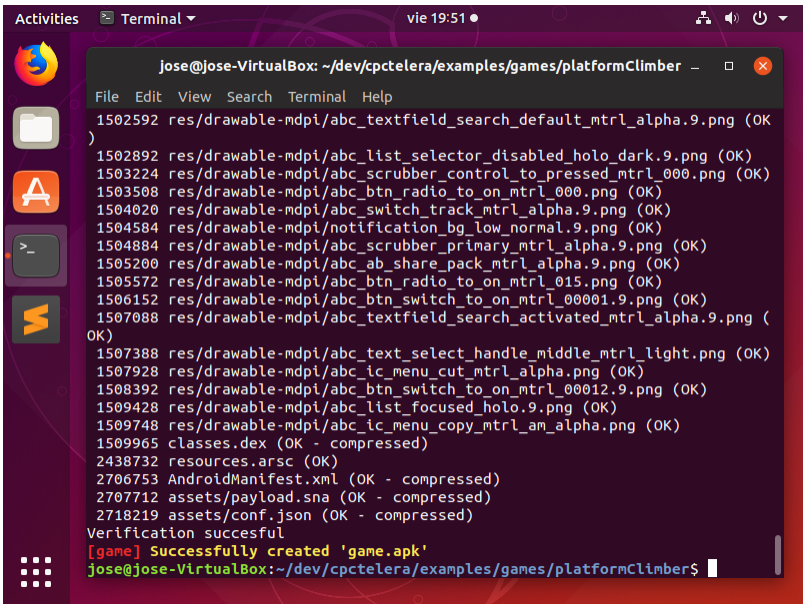

Figura 81 - Captura de pantalla tras generar la aplicación para Android

10 Referencias

Amestoy, J. C. G., 2019. *User manual of Retro Virtual Machine v2.* [En línea]
Available at: https://www.retrovirtualmachine.org/book/usermanual/en/

Android, 2019. *Activities.* [En línea]
Available at: https://developer.android.com/guide/components/activities.html

Android, 2019. *Activity Lifecycle.* [En línea]
Available at: https://developer.android.com/guide/components/activities/activity-lifecycle

Android, 2019. *Android Studio.* [En línea]
Available at: https://developer.android.com/studio/

Android, 2019. *App Signing.* [En línea]
Available at: https://developer.android.com/studio/publish/app-signing.html

Android, 2019. *Declaring a Layout.* [En línea]
Available at: https://developer.android.com/guide/topics/ui/declaring-layout

Android, 2019. *Exception on Java.* [En línea]
Available at: https://developer.android.com/reference/java/lang/Exception

Android, 2019. *Gradle en Android Studio.* [En línea]
Available at: https://developer.android.com/studio/intro

Android, 2019. *Processes and Threads.* [En línea]
Available at: https://developer.android.com/guide/components/processes-and-threads.html

Android, 2019. *Screen Densities on Android.* [En línea]
Available at: https://developer.android.com/training/multiscreen/screendensities

Android, 2019. *Sobre Android NDK.* [En línea]
Available at: https://developer.android.com/ndk

Android, 2019. *Support Library.* [En línea]
Available at: https://developer.android.com/topic/libraries/support-library

Arnold, 2017. *Arnold Emulator FAQ.* [En línea]
Available at: http://arnold.cpc-live.com/faq.html

CPCWiki, 2018. *CPC Loader.* [En línea]
Available at: http://www.cpcwiki.eu/index.php/CPCGamesCD-CPCLoader

CPCWiki, 2018. *Format CPC Digital Tape.* [En línea]
Available at: http://www.cpcwiki.eu/index.php/Format:CDT_tape_image_file_format

CPCWiki, 2018. *Format DSK.* [En línea]
Available at: http://www.cpcwiki.eu/index.php/Format:DSK_disk_image_file_format

CPCWiki, 2018. *Format SNA.* [En línea]
Available at: http://www.cpcwiki.eu/index.php/Format:SNA_snapshot_file_format

DotEmu, 2016. *Neo Geo Humble Bundle project: clarifications.* [En línea]
Available at:
http://web.archive.org/web/20160307220250/http://www.dotemu.com/Letter/NeoGeo.html

FMS Devel, 2011. *CPCDroid - Amstrad CPC on Android phone.* [En línea]
Available at: http://fmsdevel.wisecoding.es/blog/cpcdroid---2011-03-02

Free Software Fundation, 2019. *GNU General Public License.* [En línea]
Available at: https://www.gnu.org/licenses/quick-guide-gplv3.html

Google Play, 2019. *Condiciones de Servicios de Google Play.* [En línea]
Available at: https://play.google.com/intl/es-419_es/about/play-terms/index.html

Google Play, 2019. *Droid-CPC - Apps on Google Play.* [En línea]
Available at: https://play.google.com/store/apps/details?id=com.kokak.droidcpc

Hackernoon, 2019. *The SOLID Principles.* [En línea]
Available at: https://hackernoon.com/solid-principles-530b2cc2badf

ibotpeaches, 2019. *Apktool.* [En línea]
Available at: https://ibotpeaches.github.io/Apktool/

Iconjam, 2019. *En agradecimiento por el icono del Reaper utilizado en la BSOD.* [En línea]
Available at: http://www.icojam.com/

Jahrome, 2011. *AndCPC - Amstrad CPC emulator for android.* [En línea]
Available at: https://code.google.com/archive/p/andcpc/

Kokak Games, 2016. *Droid-CPC is an Amstrad CPC emulator for Android..* [En línea]
Available at: http://kokak.free.fr/android/Droid-CPC

Raldus, 2019. *API Roland CPC Emulator.* [En línea]
Available at: https://www.rolandemu.de/en/information/api.html

Ryantzj, 2017. *Structure of an APK Pacckage Part 1.* [En línea]
Available at: http://www.ryantzj.com/android-applicationpackage-apk-structure-part-1.html

SDL, 2019. *About SDL Surface.* [En línea]
Available at: https://wiki.libsdl.org/SDL_Surface

SDL, 2019. *About SDL Texture.* [En línea]
Available at: https://wiki.libsdl.org/SDL_Texture

SDL, 2019. *About Simple DirectMedia Layer.* [En línea]
Available at: https://www.libsdl.org/

Sega, 2019. *About SEGA Forever.* [En línea]
Available at: https://forever.sega.com/about

UXDesign.cc, 2018. *Building the Perfect Splash Screen.* [En línea]
Available at: https://uxdesign.cc/building-the-perfect-splash-screen-46e080395f06

W3Schools, 2019. *Introduction to JSON.* [En línea]
Available at: https://www.w3schools.com/js/js_json_intro.asp

W3Schools, 2019. *Introduction to XML.* [En línea]
Available at: https://www.w3schools.com/xml/xml_whatis.asp

Wikipedia, 2019. *Application Progrraming Interface.* [En línea]
Available at: https://en.wikipedia.org/wiki/Application_programming_interface

Wikipedia, 2019. *Blue Screen of Death.* [En línea]
Available at: https://en.wikipedia.org/wiki/Blue_Screen_of_Death

Wikipedia, 2019. *Command-line Interface.* [En línea]
Available at: https://en.wikipedia.org/wiki/Command-line_interface

Wikipedia, 2019. *Create, Read, Update and Delete.* [En línea]
Available at: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

Wikipedia, 2019. *Debugger.* [En línea]
Available at: https://en.wikipedia.org/wiki/Debugger

Wikipedia, 2019. *Game Engine.* [En línea]
Available at: https://en.wikipedia.org/wiki/Game_engine

Wikipedia, 2019. *Porting.* [En línea]
Available at: https://en.wikipedia.org/wiki/Porting

Wikipedia, 2019. *Scripting.* [En línea]
Available at: https://en.wikipedia.org/wiki/Scripting_language

Wikipedia, 2019. *Software Framework.* [En línea]
Available at: https://en.wikipedia.org/wiki/Software_framework

WinAPE, 2015. *The WinApe History.* [En línea]
Available at: http://www.winape.net/help/history.html