Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Psychocactus: Desarrollo de un videojuego musical y pixel art para Android

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Javier Martínez Campos

Tutor: David de Andrés Martínez

2020 / 2021

Resumen

En una época en que el desarrollo de videojuegos es más accesible que nunca, prescindir de un motor de juego puede resultar extraño. Pero cuando deseas lograr una precisión milimétrica, y escapar de las limitaciones tecnológicas y legales de las grandes plataformas privativas de desarrollo, Java se presenta como un lenguaje de programación versátil, eficiente y sencillo.

Este es el caso cuando el propósito es desarrollar un juego musical de estética *pixel art*. Los motores de juego están orientados a un uso genérico, pero los requisitos de este proyecto exigen un control minucioso. Cada animación deberá estar estrechamente sincronizada con las pistas musicales, y las acciones del jugador pueden influir en gran medida sobre aspectos tecnológicos que debemos diseñar para cumplir con nuestro propósito.

El sistema operativo Android cuenta además con una gran base de usuarios, por lo que orientar al desarrollo a móviles resulta como una opción preferente al PC. Contaremos además con un llamativo apartado visual y sonoro, con el que promocionar el proyecto en redes sociales y plataformas de uso extendido en dispositivos móviles. El entorno elegido para realizar la aplicación será Android Studio, conocido por ser el IDE oficial para la plataforma Android.

Palabras clave: Android, videojuego, aplicación móvil, proyecto multidisciplinar, producto comercial.

Resum

En un moment en que el desenvolupament de videojocs és més accessible que mai, prescindir d'un motor de joc pot semblar estrany. Però quan es vol aconseguir una precisió mil·limètrica, i escapar de les limitacions tecnològiques i legals de les grans plataformes de desenvolupament privatiu, Java es presenta com a un llenguatge de programació versàtil, eficient i senzill.

Aquest és el cas quan el propòsit és desenvolupar un joc musical d'estètica *pixel art*. I és, de fet, el caràcter multiplataforma d'aquest llenguatge el que permet amb molt poc esforç transportar el projecte a múltiples entorns.

Per a una primera versió, el desenvolupament s'orientarà cap al sistema operatiu Android, ja que té una gran base d'usuaris, que està en constant creixement. A més, tindran una cridanera secció visual i sonora, amb la qual impulsar el projecte a les xarxes socials. L'entorn escollit per a fer l'aplicació serà Android Studio, conegut com l'IDE oficial per a la plataforma Android.

Paraules clau: Android, videojoc, aplicació mòbil, projecte multidisciplinari, producte comercial.

Abstract

At a time when video game development is far more accessible than ever, using a game engine seems like the only option. But when you want to achieve pinpoint accuracy and escape the technological and legal limitations of complex proprietary development platforms, Java presents itself as a versatile, efficient and simple programming language.

This is the case when the purpose is to develop a musical game involving pixel art aesthetics. And it is in fact the cross-platform characteristic of this language which allows to port any project to multiple environments almost effortlessly.

For a first release, the focus development platform will be the Android operating system, since it has a large user base, which is moreover constantly growing. In addition, we will have eye-catching visuals and a great sound section, which will be useful to promote this project on social networks. The environment chosen to make the application will be Android Studio, known as the official IDE for the Android platform.

Keywords: Android, video game, mobile app, multidisciplinary project, commercial product.

Agradecimientos

A Alejandro Jimenez Carrasco, que inspiró la idea de este juego, y rediseñó decenas de veces al bajista.

A Raquel Arias Montañana, artista con todas sus letras, y profesional mayúscula.

A Begoña Moret Gea, increíble diseñadora, y visionaria del estilo y concepto.

A Adrián Reina Saez, de un gran talento, y que se resiste a reconocerlo.

A Sergio García Garay, músico de aspiración mundial, pero con domicilio fiscal en *la terreta*.

Y gracias a todas las personas que en algún momento mostraron una gran ilusión por aportar su granito de arena.

Palabras clave

Android; videojuego; aplicación móvil; proyecto multidisciplinar; producto comercial.

Prólogo

Siempre he dicho que el día que más recordaré para siempre de mi paso por la universidad será precisamente el primero de todos. Durante la jornada de bienvenida de septiembre de 2016 que organizó la UPV, todos los alumnos que, y entre cuyo grupo me incluyo, estábamos desesperados por comenzar de una vez con nuestra vida en la universidad, tuvimos la oportunidad de acudir a talleres, conocer compañeros con nuestros mismos intereses, y escuchar los discursos de profesores que abrirían esta nueva etapa.

¡Y qué discursos! No soy capaz de dar nombre ni rostro en mis recuerdos al autor de la siguiente cita: "*A la universidad, tened muy claro que no venís a hacer videojuegos*". Eso fue doloroso para mí, lo admito.

Pero también reconozco que sabía muy bien a quiénes iba dirigido ese comentario. También sabía que a ese grado me inscribí para analizar costes de algoritmos, diseñar estructuras de datos, y comprender de una vez por todas qué diablos es internet.

Después del proceso que ha significado aprobar cada asignatura de las que he cursado, aunque no todas al primer intento, me siento orgulloso de decir que conozco las respuestas a muchas de esas preguntas que siempre me hacía.

Este proyecto comprende la intersección entre la pasión de mi vida, los videojuegos, y la que considero la única profesión a la que podría dedicarme, la Ingeniería Informática. Como tal, ha sido realizado desde el más absoluto cariño y respeto; aunque me sienta en parte dolido por no haberle podido dedicar uno, dos, o cinco años de mi vida, ya que solo me falta para sentirme realizado ver mi nombre deslizarse por los títulos de crédito de alguna gran obra.

Tabla de contenidos

1. Introducción 12
   0) Motivación 13
   1) Objetivos 14
   2) Impacto esperado 15
   3) Metodología 15
   4) Estructura 19
   5) Colaboraciones 20
   6) Conclusión 21
2. Estado del arte 22
   Inicios del juego móvil 22
   Entrada en juego de los smartphones 23
   0) Crítica al estado del arte 24
   Editor sencillo de videojuegos 26
   Motor de videojuegos 3D / 2D 26
   Framework de gráficos 27
   Android SDK 28
   1) Propuesta 28
3. Arquitectura software 30
   Especificación superficial de la arquitectura 30
   0) Sistema de gestión central 31
   Mecanismo de inicialización de la aplicación 32
   Las clases DrawThread y UpdateThread 33
   Mecanismo de dibujado 33
   Mecanismo de actualización de estados 34
   1) Escenas 35
   2) Controladores de entidades 36
   Dimensions 37
   Clickable 37
   MenuDisplay 37
   Drawable 38
   AbstractSprite 38

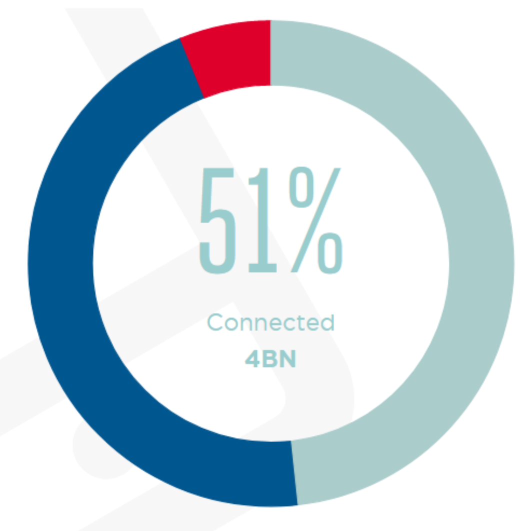
AnimatedEntity ... 38
InanimateSprite ... 38
ClickableAnimation ... 38
ClickableDirectSprite ... 39
ClickableSprite ... 39
CustomDrawableEntity ... 39
CustomClickableEntity ... 39
Menús contextuales ... 40
3) Reloj de tareas ... 41
4) Gestor de música dinámica ... 42
Temas fragmentados ... 42
Aislamiento de los instrumentos ... 43
Diseño e implementación ... 43
5) Lector de controles ... 46
Controles táctiles ... 47
Ejecución de la orden ... 48
Sistema de colisiones ... 48
6) Transición entre escenas ... 49
7) Cola de tareas ... 50
QueueManager ... 50
Automation ... 50
Efectos visuales ... 51
8) Utilidad de debug ... 52
DebugHelper ... 52
9) Ajustes de usuario ... 53
Selección de idioma ... 54
Selector de volumen y silencio ... 54
10) Gestor de recursos ... 54
Referencias ... 54
Instancias flyweight ... 55
4. Implementación en Android ... 56
0) Resumen de las características ... 56
1) Dibujado ... 56
2) Actividades ... 57
AndroidManifest ... 58

Ciclo de vida de la aplicación 59
3) Fragmentos 60
4) Nivel de API 61
5) Carga de recursos 63
6) Javadocs 64
5. Aspectos ajenos a la tecnología 66
0) Música 66
1) Arte 68
2) Jugabilidad 69
3) Promoción y monetización 70
6. Futuras versiones y metas 71
0) Optimizaciones 71
Sustitución de SoundPool por implementación OpenSL 71
Mayor eficiencia en el uso de memoria RAM 72
Mayor control sobre los hilos 73
1) Abstracción del motor 74
2) Características del juego 75
Soporte para jugar una partida completa 75
Nuevos niveles 75
3) Lanzamiento comercial 75
7. Resultados 77
Pantalla de carga 77
Pantalla de inicio / Menú principal 77
Pantalla de diálogo – Selección y alerta 78
Pantalla de juego 78
Zoom 79
Pantalla de pausa 80
Pantalla de opciones 80
Modo de debug 81
8. Conclusiones 82
9. Referencias 84

1. Introducción

La informática ha cambiado. Lleva evolucionando desde sus inicios, y avanza cada año a pasos agigantados. Desde que en 2008 es presentado lo que más adelante se convertiría en un revolucionario dispositivo, el teléfono inteligente o *smartphone*, la industria ha orientado gran parte de sus esfuerzos en el sentido de la accesibilidad y conectividad total: la mitad del mundo tiene uno o varios dispositivos encima, capaces de conectarse a internet para realizar todo tipo de actividades. Desde pedir comida a domicilio hasta contratar un préstamo, pasando por realizar video comunicaciones con familiares situados en la otra punta del globo.

Y este es un cambio tanto cuantitativo como cualitativo. Las cifras lo corroboran. Un 49% de personas en el mundo se han conectado a internet a través de un dispositivo móvil, según afirma GSMA [1], y habiendo sido producidos hasta la 2018 un total de 22 mil millones de dispositivos con capacidad de conectividad a la red [2], según estima *Strategy Analytics*.



Podríamos realizar todo tipo de valoraciones sobre las implicaciones que tiene esto en todos los ámbitos del estudio, tanto sociológicas, económicas y políticas como medioambientales; pero en para el caso que nos concierne vamos a hablar del mercado.

De la totalidad de teléfonos de los que dispone la población, hasta un 72.83% se basa en una distribución de Android, según datos de *statcounter* [3]. Esto representa la nada despreciable cantidad de casi tres mil millones de potenciales usuarios objetivo para nuestro producto. Sabemos que estos datos no se traducen en un éxito certero, ya que de la misma manera que encontramos tanta demanda, existe una saturación del mercado que dificulta enormemente a pequeños desarrolladores hacer una vida laboral basada en este sistema. La revista *xataka* recoge datos de *statista*, los cuales afirman que para 2019 había 3 millones de aplicaciones [4].

Pero esto no nos detiene, puesto que uno de nuestros objetivos es el de emular el proceso de desarrollo de una aplicación competitiva. Durante este Trabajo de Fin de Grado, será discutido el proceso por el que se lleva a cabo esta actividad, proponiendo soluciones para problemas al

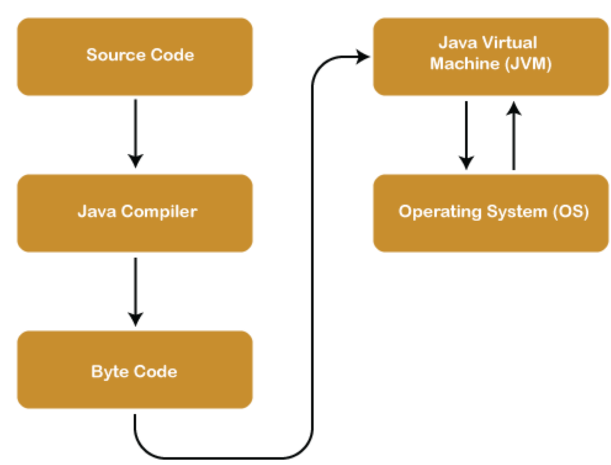
que se pueden enfrentar los desarrolladores, con la intención de exponer una vía que ofrece grandes ventajas, como se desarrollará más adelante.

0) Motivación

La idea de realizar este proyecto nace muy pronto. Allá por 2016, durante mis primeras clases de programación del grado de ingeniería informática de la *Universitat Politéctica de València*.

Durante estas clases nos explican por qué Java es elegido el lenguaje para dar los primeros pasos en este mundo. Entre las principales razones se encontraba la gran versatilidad de sus librerías, la gran inversión que existe tras su desarrollo el gran soporte que existe por parte de las compañías, y la enorme adopción entre la comunidad para este lenguaje. Y sigo estando de acuerdo con estos argumentos. Este gráfico aporta una de las pruebas que defiende su prestigio [5].

Pero uno de los puntos fundamentales que hacen de este lenguaje una herramienta tan versátil es intrínseco a su propio diseño. Java compila todo su código a un lenguaje intermedio, conocido como *bytecode*, y es tarea de la máquina virtual Java, instalada en cualquier sistema, ejecutar este código de manera posterior. Esta aproximación es la conocida como '*code once, run anywhere*'. De esta forma, cualquier sistema operativo que implemente su propia máquina virtual Java es capaz de ejecutar este código con escasas modificaciones.



Esto no es del todo preciso al describir la integración de Java en Android. En el caso de Android, Google compró desde una etapa relativamente temprana de su desarrollo este proyecto, cuando aún no contaba con la cuota de mercado de la que presume hoy en día. Y es que de hecho hoy día el gran éxito de Android le ha traído algunos dolores de cabeza a Google. Tal como cuenta Gabriel Erard en su artículo de la revista *hipertextual*, el pasado 5 de abril de 2021, Google finalmente venció a Oracle en el máximo tribunal de justicia estadounidense. ¿El motivo? Los derechos de autor sobre Java [6].

Oracle acusó a Google de **copiar unas 12 mil líneas de código** de la plataforma Java durante la creación del sistema operativo Android. Por su parte, la corporación de Mountain View aseguró que su implementación del código estaba enmarcada en los parámetros de "uso justo", y por ello no aplicaban las leyes de derecho de autor.

Muchos afirman ahora que toda esta batalla legal fue el principal motivo que impulsó la creación de *Kotlin*, lenguaje que desde 2017 es catalogado por Google como el lenguaje oficial de Android [7].

Volviendo a nuestro proyecto, es más tarde, cursando la asignatura de desarrollo en Android, cuando descubro la facilidad de desarrollar una aplicación para esta plataforma, si se cuenta con los conocimientos adecuados.

Ahora bien, existen diferentes vías de lograr este propósito. La que cuenta con el soporte oficial por parte de Google, es la de hacer uso del IDE Android Studio. Esta será la opción que exploraremos en este proyecto, pues es la que permite una mayor cercanía al propio sistema, y la que cuenta con el mayor grado de libertad de cara al desarrollador.

Es cierto que existen facilidades para desarrollar aplicaciones modelo, utilizando las librerías recomendadas por Google. Esta es la opción más sensata si se desea lograr la mayor usabilidad y funcionalidad con poco esfuerzo, y que además cuenta con la ventaja de que entre diferentes versiones no será necesario modificar más de unas pocas líneas de código. Pero si se opta por tomar control manual de los hilos de gráficos y de los eventos de entrada, es posible construir un auténtico motor de videojuegos desde nada más que líneas de código. De nuevo, esta será la opción escogida para la realización de este proyecto.

1) Objetivos

Aunque el fundamental propósito de este proyecto sea el de ilustrar el proceso de desarrollo de un videojuego de unas características particulares, colocando el foco principal sobre la ingeniería que se haya tras los bastidores, y en la que se ha destinado un mayor esfuerzo, se desea demostrar también qué cuestiones externas rodean este producto software, y cuáles son los pasos que debería dar un ingeniero para poder materializar una idea en un producto comercial y competitivo. Esto incluye las colaboraciones con artistas y con asesores de marketing, el contacto con stakeholders, y breves comentarios sobre otros asuntos, que escapan al dominio de la ingeniería, o guardan una escasa relación, pero sí cuentan con un papel protagonista en el mundo de los videojuegos, como lo es el diseño de la jugabilidad y la experiencia de usuario.

Pero vamos a intentar siempre conservar un punto de vista autocrítico y realista. Los recursos con los que cuenta una gran empresa, o al menos una pequeña o mediana que ya tenga sus frentes de negocio asentados, y una liquidez de la que sacar provecho, no son los mismos con los que cuenta un desarrollador independiente; que en cualquier caso debe vivir de ahorros previos, de un salario a media jornada, o contraer una deuda. Es decir, los inicios son duros, y

no se pretende lograr el próximo superventas, ya que para eso como mínimo es necesario invertir un tiempo sustancial, del que no se dispone para este trabajo.

De esta forma, no se establece ninguna meta económica, sino que se aspira a realizar un producto sólido, que cumpla con los más rigurosos estándares de desarrollo y calidad de software, así como redactar una memoria, que sirva como guía a cualquier futuro desarrollador cuyo proyecto comparta las mismas características que el que se encuentra aquí expuesto.

Al finalizar este producto, se habrán generado una serie de recursos que ilustrarán decisiones tomadas durante el desarrollo; estos aportarán un trasfondo sobre el que poder discutir las críticas, reflexionar sobre los errores cometidos, y proponer una metodología.

2) Impacto esperado

En términos cuantitativos, y como se ha hecho una breve mención antes, no se espera una apabullante acogida por parte del público de Android. Son bien conocidos algunos casos en que sonríe la suerte, y desarrolladores anónimos pasan a poblar las listas de multimillonarios; pero a menos que se cuente con esta esperanza, existen pocas garantías de éxito más allá de la viralización.

En otro sentido, sí sería de agrado que este proyecto sirviese como punto de partida para que nuevos proyectos tomen forma, ya que una redacción breve de un par de páginas, contando el por qué se ha tomado según qué decisión, puede haber supuesto originalmente decenas de horas de trabajo, que son evitables si se cuenta con esta información de antemano.

3) Metodología

Ya que este proyecto pretende emular las modernas técnicas y metodologías aplicadas por la industria, se intenta guardar una rigurosidad en este sentido, dentro de lo que cabe. Es cierto que, al tratarse de un proyecto realizado por una única persona, no sería perjudicial alterar los ritmos de trabajo, o tomar con cierto grado de flexibilidad algunas de las normas más estrictas. Todo proyecto de desarrollo software requiere de establecer una metodología clara, que permita a un equipo coordinarse y aunar esfuerzos, con el objetivo de transformar cada hora de trabajo en un avance significativo.

Dicho lo anterior, cabe destacar que no se ha descuidado ninguno de estos apartados, conociendo que de crecer en un futuro el producto o reimaginarse de cualquier manera, será necesario haber generado previamente una serie de documentos organizados, que puedan ser vinculados a líneas de código identificables, para que cualquier profesional pudiese sumarse sin mayor problema; y en general, para hacer del conjunto del software incrementar su calidad.

Este proyecto ha seguido en todo momento la filosofía del '*agile programming*', que se populariza como respuesta al alto grado de fracaso de los proyectos software, que seguían las metodologías más tradicionales, los cuales por múltiples motivos no terminaban siendo

satisfactorios; ya fuera por no cumplir con los plazos, con las funcionalidades prometidas, o por salir del presupuesto.

El blog de Jerónimo Palacios recoge un excelente resumen de esta idea, tratando en uno de sus artículos las ventajas de *Agile* frente al modelo *Waterfall* [8]. Para esto, repasa un informe de pago de Standish Group, llamado CHAOS [9].

En un resumen de sus características más relevantes de la aplicación del desarrollo ágil, esto implica la existencia de los siguientes conceptos.

- ‘*Backlog*’: Un conjunto de tareas unitarias, jerarquizadas y ordenadas por urgencia y otra serie de parámetros, que se encuentran pendiente de realización.
- ‘*Sprints*’: Plazos de tiempo de media duración, de entre 2 a 4 semanas, en los cuales se escogen una serie de tareas pertenecientes al backlog, que se espera completar para la fecha de finalización del sprint, y tras el que se desea obtener un producto mínimo viable.
- ‘*Producto mínimo viable*’: En inglés, ‘*Minimum Viable Product*’ o ‘*MVP*’, se trata de una versión intermedia, y única, del producto a desarrollar, que contiene solo un subconjunto de funcionalidades previstas para la entrega del proyecto, y que permite desde una fase temprana del desarrollo realizar pruebas para evitar en la medida de lo posible desperdiciar horas de trabajo.
- ‘*Épicas*’, ‘*Features*’, ’*Historias de usuario*’, ’*Tareas*’: Son las principales unidades en que se mide el trabajo, contenidas unas dentro de otras, por niveles jerárquicos. Las épicas representan características clave del producto, y son las de más alto nivel; y las tareas, al contrario, son labores de base que conllevan unas pocas modificaciones, y que deben especificar con gran exactitud el trabajo a realizar para permitir el progreso del software.

Explicadas las más fundamentales características, se considera que esta aproximación al trabajo permite depender de un menor volumen de documentación relativa al análisis y al diseño, y facilita la temprana implementación de código sin sacrificar la rigurosidad ni la calidad del producto resultante.

Por otro lado, también es relevante mencionar la metodología adoptada en cuanto al control de versiones, y la importancia de este apartado a pesar de trabajar una persona en exclusiva. Por los mismos motivos antes mencionados, es interesante contar con un historial de trabajo que permita identificar los motivos que han conducido a la existencia o no de cada fragmento de código software. Y este no es un asunto menor, pues se ha buscado implementar una aproximación a lo que sería el modelo conocido como ‘*Git-Flow*’. Esto se trata realmente de una metodología, o flujo de trabajo, ideada sobre las capacidades del extendido software Git. El control de versiones Git cubre la necesidad de guardar un detallado registro de todos los cambios por los que pasa cada fragmento del código. Este consta de ramas, que sencillamente apuntan a las versiones de cada fichero del proyecto, que se hayan generado tras realizar un ‘*commit*’, una instantánea del estado del conjunto de software. Git-Flow viene a proponer

unas normas que seguir para sacar el máximo provecho de las características con las que cuenta este software. Esto es extremadamente útil en proyectos que crezcan de manera iterativa, que tengan versiones publicadas en red y disponibles para los usuarios, o para equipos en que más de una persona tenga acceso al código.

Pero no todo en el desarrollo de videojuegos es el software. Para desarrollar este proyecto de principio a fin, se ha optado por contactar con diversos profesionales del mundo del arte, que han aportado en menor o mayor medida una labor vital. Esto es algo de lo que ni siquiera los más habilidosos ingenieros pueden escapar en la mayoría de las situaciones, y es que una persona especializada puede lograr resultados asombrosos en mucho menos tiempo del que le llevaría a un experto en ingeniería del software. Y es por este motivo por el que no se ha hecho excepción con este proyecto. Si se desea plasmar un proceso de desarrollo fiel a la realidad, es necesario contar con asistencia artística, y aquí es donde no es posible escapar de la documentación, pues se tratan de personas ajenas al proyecto, y que no tienen por qué conocer las metodologías implementadas en este proyecto. Es por esto por lo que se han redactado cuatro documentos diferentes, independientes de la presente memoria del TFG, que tratan cuatro de las categorías creativas más relevantes del mundo de los videojuegos. Estos tratan, pues, acerca de:

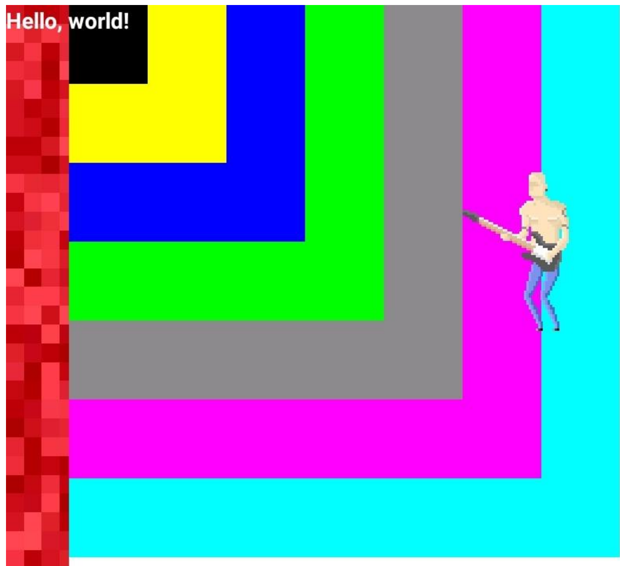
- El diseño de jugabilidad. Toda aquella cuestión relativa a las normas de juego, algo parecido a lo que serían las instrucciones de que traen cualquier juego de mesa, pero destinadas al entendimiento por parte de profesionales, y no para usuarios finales. La analogía más parecida que se podría trazar en materia de documentación software podría ser la de diagramas de casos de uso, o la de diagramas de flujo, pero que en ningún caso guardaría una relación paralela, ni sería beneficiosa su aplicación forzosa para el caso que nos trata.
- La especificación de requisitos del arte visual. Una descripción de todo requisito artístico que pueda necesitar un dibujante, animador o profesional de ámbito similar. Este documento clasifica además cada una de las animaciones y posibles estados en que se puede encontrar un personaje, y que requieren de una materialización en forma de ficheros PNG antes de ser introducidos en la propia aplicación móvil.
- La especificación de requisitos de la música. Quizás de encontrarnos con otro juego esto no sería tan relevante, o sería posible salir del paso con música genérica; pero en nuestro caso, y al tratarse este proyecto de un videojuego musical, cada pista de sonido generada y utilizada debe formar parte intrínseca de la propia arquitectura software diseñada. Esto tiene una sencilla justificación; y es que, al contrario que en

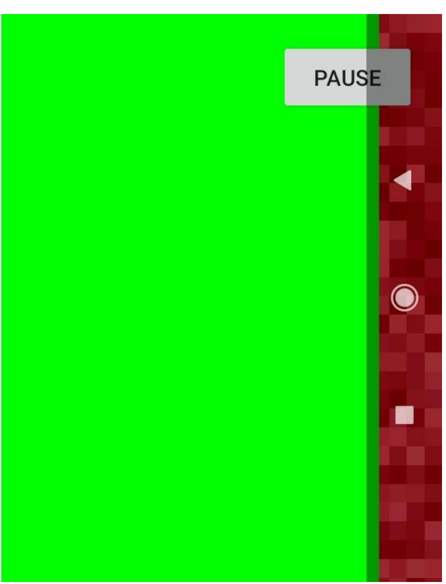
películas, musicales, o cualquier otra pieza cultural existente, los videojuegos se caracterizan por tratarse de ser un medio dinámico, autogenerado, y en que la interacción del usuario con la obra es impredecible en diferentes sentidos, por lo que el código que corre el videojuego debe, y es importante enfatizar en esto, responder ante cualquier posible estado que se dé; con lo que las pistas de música deben estar diseccionadas y preparadas para ser lanzadas por el motor en cualquier momento.

- El *briefing* de negocio. Este documento es quizás el más fantasioso, puesto que, como se ha declarado antes, el objetivo principal no es en absoluto el lucro. En contraste, continúa siendo obligatoria su realización, puesto que deseamos recrear con la máxima fidelidad el proceso de desarrollo de un proyecto software real, en que la financiación es un punto crítico y limitante, pues al menos deben ser mantenidos económicamente los profesionales que intervengan en el mismo. Es por este motivo que se ha optado por planificar toda una ruta de negocio, plasmada en este documento.

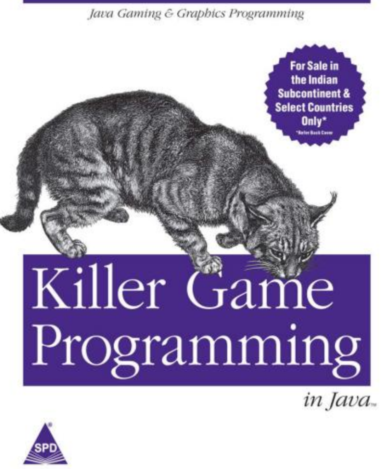
Y como último a tratar en cuanto a la introducción a las metodologías aplicadas, cabe mencionar el modo en que se ha realizado esta misma memoria. Mientras me encontraba cursando los diferentes semestres de los que se compone este grado, realizar este proyecto no era más que una ilusión, una suerte de meta por la que merecía la pena continuar estudiando. Pero no es hasta que arranca la rama de ingeniería del software que veo viable una manera de llevar estas ideas a la práctica, ni es hasta cursar la asignatura de 'Desarrollo de aplicaciones para dispositivos móviles' que realmente creo que debo iniciar con la puesta en marcha del proyecto.

Desde un principio, y hasta escoger y haber asentado las metodologías de desarrollo, no está del todo claro cuáles deben ser los primeros pasos para dar; pero tras un par de pruebas exitosas, que no fueron más que prototipos de juguete, comienza el proceso de análisis y diseño de las principales características con las que debe contar este producto software.





Lo que se puede considerar como el esqueleto fundamental de la arquitectura implementada fue basado en varios libros: ‘*Core Techniques and Algorithms in Game Programming*’ [10], que sirvió para comprender entre otros asuntos la construcción de motores software a puro código; ‘*Killer Game Programming in Java*’ [11], que sirvió para entender los fundamentos de la programación del dibujado gráfico, y la gestión de hilos para lograr estabilidad en Java; y ‘*Android Game Programming*’ [12], un fantástico libro, que puede ser entendido a modo de manual, y que explica casi todo acerca de la materia que nos ocupa en este Trabajo de Fin de Grado.



El inicio de redacción de la memoria toma lugar aproximadamente a medio camino de obtener la que sería la primera ‘entrega’, o versión mayor del producto. Desde entonces, se va escribiendo esta memoria en paralelo al desarrollo, reflejando lo que se conoce a ciencia cierta que formaría parte del producto final, e implementando progresivamente nuevas funciones presentes en el ‘*backlog*’, hasta alcanzar el estado de producto deseado.

4) Estructura

El proyecto realizado tiene como fin último el de obtener una aplicación, ‘*Psycho Cactus*’, que es la pieza alrededor de la cual todo gira, junto a la documentación de todo el proceso necesario para alcanzar este fin. Por lo tanto, en esta memoria será descompuesta, paso a paso, cada decisión que justifique la forma en que está construida la arquitectura de todo conjunto de clases; esto será visto en el punto tercero, ‘Arquitectura software’, el punto central del proyecto.

Por supuesto, la forma en que se cuente esto estará condicionado por la propia metodología de trabajo, ya tratada anteriormente. Las metodologías ágiles se pueden entender en la práctica como una atomización de un ciclo de vida software completo, aplicado a todas y cada una de las tareas a realizar. De este modo, cada *épica*, *feature* e *historia de usuario* consta de su propio análisis, diseño, implementación, pruebas e integración. Se opta entonces por recopilar a posteriori las notas y apuntes resultantes de este proceso, para ilustrar de la manera más didáctica posible cómo se ha llevado a cabo el proceso de desarrollo.

1. Introducción
2. Estado del arte
3. Arquitectura software
4. Implementación en Android
5. Aspectos ajenos a la tecnología
6. Futuras versiones y metas
7. Resultados
8. Conclusiones
9. Referencias

Se debe recordar también en todo momento que este desarrollo está fuertemente condicionado por la plataforma a la que está destinada el producto software, Android. Todas las particularidades, restricciones y ventajas serán tratadas en un punto independiente, el cuarto, ‘Implementación en Android’.

Así como Android ocupa uno de los capítulos de este trabajo, lo mismo sucede con el mundo de los videojuegos, del que se deben hacer mínimas menciones para contextualizar el trabajo realizado y justificar algunos de los requisitos técnicos del producto. Esto será visto en el quinto punto.

Ya que el tiempo con el que se cuenta para realizar este proyecto no es demasiado extenso, y sumando el hecho de que se lleva a cabo compatibilizando esta labor con un trabajo a tiempo parcial, será vital describir toda aquella característica que se haya quedado en el *backlog*, aportando también el análisis y el diseño a modo de compromiso a futuro. Una vez finalizado el TFG, la intención es la de impulsar este proyecto y convertirlo en un verdadero portento que poder mostrar como experiencia acreditable. Es por este motivo que se juzga imprescindible tratar este contenido en el punto sexto, 'Futuras versiones y metas'.

Complementario a este punto, se encuentra inmediatamente después el séptimo, uno centrado en realizar la exposición de las características finales del producto que ha sido logrado realizar. Esto será mostrado sin mayores detalles, ya que todo el trabajo realizado para hacer este punto posible habrá sido desarrollado anteriormente.

La memoria finalizará con unas breves conclusiones en el punto octavo, y dejará paso al punto noveno, con un detalle de las referencias en que se apoya todo el trabajo realizado.

5) Colaboraciones

Tal y como se ha mencionado anteriormente, son varias las personas implicadas de una manera menos formal con este proyecto. Dada la naturaleza de este trabajo, suele ser imposible que una sola persona se encargue de todos los apartados, y especialmente en los proyectos grandes. Pero se ha de despejar todo rastro de duda, y es que cada una de las cuestiones relacionadas con la ingeniería y la tecnología de este proyecto han sido resueltas por el autor de esta memoria. Hecha la aclaración, se enumera a los colaboradores que han participado en este trabajo, todos de manera desinteresada.

- Raquel Arias Montañana: Artista especializada en ilustración. Ha dedicado gran parte de su trabajo en el proyecto a definir, cuadro a cuadro, todos los píxeles de los que se componía cada animación.
- Alejandro Jiménez Carrasco: Artista especializado en diseño gráfico y videojuegos. Ha tomado decisiones para guiar la manera en que debe ser construida la composición gráfica para reforzar la cohesión visual. También se ha ocupado de indicar el cómo lograr que los personajes expresen su personalidad a través de su apariencia.
- Begoña Moret Gea: Artista especializada en '*concept art*'. Aunque ha desempeñado distintas labores en el terreno artístico, ha destacado por anticipar las primeras traducciones de texto a imagen, ideando una apariencia que posteriormente se vería integrada en el diseño final de personajes y entornos.
- Sergio García Garay: Compositor y productor de la música presente en el videojuego. Su entrega ha constado de las diferentes pistas de música finales, separadas por instrumentos, y cortadas por fragmentos de tiempo. Estos fragmentos se utilizan para, según el estado del juego, mezclar de manera dinámica el sonido de la banda sonora; en que los factores que influyen en la generación de sonido son: el estado de ánimo del público, el estado de salud de los músicos, el progreso del jugador en la partida, y algunos otros datos internos.

Para hacer posibles estas colaboraciones, ha sido suficiente con sistematizar las comunicaciones de una manera consistente con la planificación central. Para la implementación de cada hito, se ha considerado necesario contar de manera previa con una

serie de recursos gráficos o musicales; por lo que, al haber trasladado la información pertinente con suficiente antelación, no se ha dado ninguna situación de cuello de botella.

6) Conclusión

A lo largo de este punto introductorio se han tratado diversos temas, todos de un carácter general, y que han lanzado algo de luz sobre la naturaleza y fundamentos del proyecto que se plantea.

Se ha hablado sobre los orígenes de su desarrollo, y justificado el motivo para realizarlo de la manera en que se resuelve. Más adelante se despeja la duda sobre la serie de motivos que se encuentran tras su puesta en marcha. Se ha predicho de qué manera se espera que se desenvuelva el resultado, y cuál podría ser la cabida del software en el mercado y en la academia. Después se expone, por primera vez, un resumen del conjunto de métodos de trabajo perseguidos durante el desarrollo, y se deja entrever las implicaciones que tiene esto sobre la tecnología que se implementa, así como exponer la definición de varios conceptos que pueden no ser del todo familiares, pero que son de integración vital para contar con la garantía de que los plazos y las expectativas se cumplen. Luego, se ha indicado superficialmente cómo es afrontada la redacción de esta memoria. Y, por último, se ha hecho una mención especial a las cuatro personas que han hecho posible algunos de los apartados creativos de este producto, que es intrínsecamente multidisciplinar. En definitiva, se ha pretendido familiarizar al lector con el contexto que rodea al videojuego realizado.

A continuación, se pasa a discutir el estado del arte, y cuál es la posición que pretende ocupar el producto desarrollado, en un sentido comercial y tecnológico.

2. Estado del arte

De la misma forma que las primeras palabras de la introducción a esta memoria iban dedicadas a la historia reciente de la ciencia de los computadores y su industria, comenzamos analizando lo que se juzga como el estado contemporáneo de las obras más sobresalientes de este medio, haciendo hincapié en la evolución que ha sufrido, con el paso de un muy breve tiempo, el compendio de los videojuegos móviles más influyentes de la pasada década, y cómo este medio está tan atado, y a la vez es propulsado, por la tecnología en que se desenvuelve.

Cualquier comentario o valoración que se haga sobre la realidad material del mundo de los videojuegos móviles, y también de la ingeniería del desarrollo que hace posible cada año el nacimiento de nuevas propuestas, debe estar vinculada a una referencia temporal, un periodo específico, junto a otros factores esenciales. Es decir, en base a las tecnologías disponibles hoy sería posible en ocasiones caer en un engaño, pudiendo calificar como de valor mediocre obras que en su día presentaron méritos técnicos sin precedentes.

En ocasiones, se comete el error de juzgar este medio atendiendo únicamente a los titulares y al baile de cifras que genera la industria trimestre a trimestre. Y es que se trata de un sector del que no solo es interesante fijarse en los beneficios de cada empresa, o de los logros de ejércitos de un solo hombre que logran en ocasiones eclipsar a las grandes corporaciones. Tras cualquier industria en que la creatividad se entienda como variable, y el arte juegue un papel imprescindible, existe también un mostrador del potencial humano, una galería de joyas ocultas; y con los videojuegos no cabe hacer excepción.

Pero no se debe olvidar el foco principal de discusión, que es el de la investigación y desarrollo tras los grandes y pequeños títulos. A través de un eje cronológico que resalte los principales estilos y géneros que han tenido más relevancia en las últimas décadas, se desarrollará un esquema de lo que han sido los métodos para hacer crecer el medio de los videojuegos móviles, dando un breve repaso a su historia. Esto servirá como contexto para, más adelante, discutir sobre el estado del arte a través de una referencia actualizada.

Antes de hablar del estado actual de los videojuegos móviles, y para comprender algunas de las decisiones tomadas tras la concepción y el desarrollo del producto, es necesario exponer lo que se interpreta como las etapas que ha recorrido el medio en su larga trayectoria.

Para hacer esta exposición trazaremos una línea de separación entre dos eras, en que el punto de inflexión es, sin duda, el lanzamiento del primer iPhone. Estamos hablando del antes y el después de los smartphones.

Inicios del juego móvil

Allá en los años 80, los videojuegos eran un negocio incierto, y las tecnologías aun primitivas. Tanto es así que estuvieron muy cerca de desaparecer. El conocido como '*crack del videojuego*' (1983), hizo temblar los cimientos de la industria. Tan solo Nintendo fue capaz de reconstruir el panorama del momento, como bien explica José Altozano en su canal, *DayoScript*, con el documental web titulado '*La Leyenda del Videojuego*' [13].

Por entonces, el catálogo de dispositivos móviles disponibles en el mercado no era tan diverso como en la actualidad. De entre los pocos productos que permitían portabilidad,

abundaban calculadoras y relojes. De hecho, uno de los primeros ejemplos de videojuego portátil lo encontramos en Nintendo con sus Game & Watch, un dispositivo que funcionaba a modo de reloj, y que contenía en sus circuitos un videojuego. Es aquí donde nace, por ejemplo, el famoso Donkey Kong. Esto producto podía ser vendido como de utilidad práctica, pero era deseado por sus cualidades de entretenimiento.

Los primeros teléfonos portátiles se extendieron por la década de los 80. Japón fue en este campo pionero, inaugurando la primera red de radiotelefonía comercial [14]. Ya en los 90 todos los móviles que distribuía Nokia contenían una copia del mítico Snake, juego que les ayudó a vender su terminal, y que registra la enorme cifra de 400 millones de copias distribuidas [15]. Esto permitió que dispositivos de comunicaciones fuesen percibidos también como consolas portátiles de videojuegos.

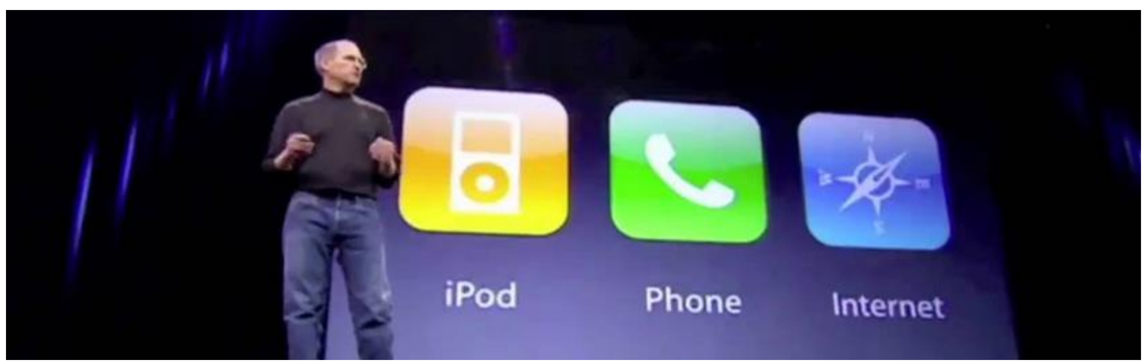
Ya en la década de los 2000, era común que las operadoras de telefonía móvil ofreciesen un servicio de descargas a modo de *marketplace* de aplicaciones, para las que generalmente había que pagar no solo el contenido, sino cada kilobyte de datos descargado. No había un ecosistema homogéneo, por lo que lanzar un mismo videojuego de forma común a un gran porcentaje de usuarios se hacía tarea imposible.

Entrada en juego de los smartphones

Desde su primera serie de *Game & Watch* (1980), hasta el lanzamiento de la Nintendo DS, Nintendo disputaba un mercado en que prácticamente su único gran contendiente era Sega. Con la caída de la compañía que dio vida a Sonic, y su posterior retirada de la industria de videoconsolas, Nintendo y Sony se convirtieron en los nuevos reyes del mercado portátil. Fue durante pleno ciclo de vida para las consolas PSP y Nintendo DS cuando todo cambió.

Steve Jobs presenta su primer modelo de iPhone en un momento en que el mercado móvil estaba enormemente fragmentado. Ofrece un dispositivo táctil, con una cantidad considerable de RAM para la época, un sistema operativo multitarea, y lo presenta como la

unificación entre tres necesidades básicas: llamadas, multimedia y navegación web, en cualquier sitio y en la palma de tu mano [16].



Esto, sumado a una tienda unificada de aplicaciones, con políticas abiertas a las *third party*, que buscaba llamar la atención de todo desarrollador, pequeño o grande, sumado a las populares historias de éxito de gente previamente desconocida, resultó en la mezcla ideal que popularizaría el smartphone. La app store de iOS, el sistema operativo del iPhone pasó de tener unas 500 aplicaciones en el lanzamiento de la tienda [17] a más de 10.000 en cuestión de un año [18].

Google, con su reciente adquirido sistema operativo Android, no quiso ser menos en este fenómeno cultural y económico. Puso todos sus medios en marcha para lograr hacer frente a Apple. Ni la veterana BlackBerry, ni el aspirante Windows Phone, has conseguido seguir la estela hasta el día de hoy. La cuota de mercado en el ámbito de los smartphones está dominada por las dos compañías rivales, mientras que el resto de las marcas se limita a ofrecer terminales sencillos, tradicionales, y accesibles, para un público con otro tipo de necesidades.

De aquí en adelante, el videojuego móvil sería cada vez más popular. Vemos en esta plataforma ejemplos como *Candy Crush*, *Angry Birds* y *Clash of Clans*, que lograron llegar a un público que jamás había tenido contacto con el videojuego. Hasta el día de hoy, las cifras que mueven solo los diez títulos más populares del mercado superan, cada uno, los tres mil millones de dólares estadounidenses, y alcanzan los nueve mil millones en los primeros puestos [19].

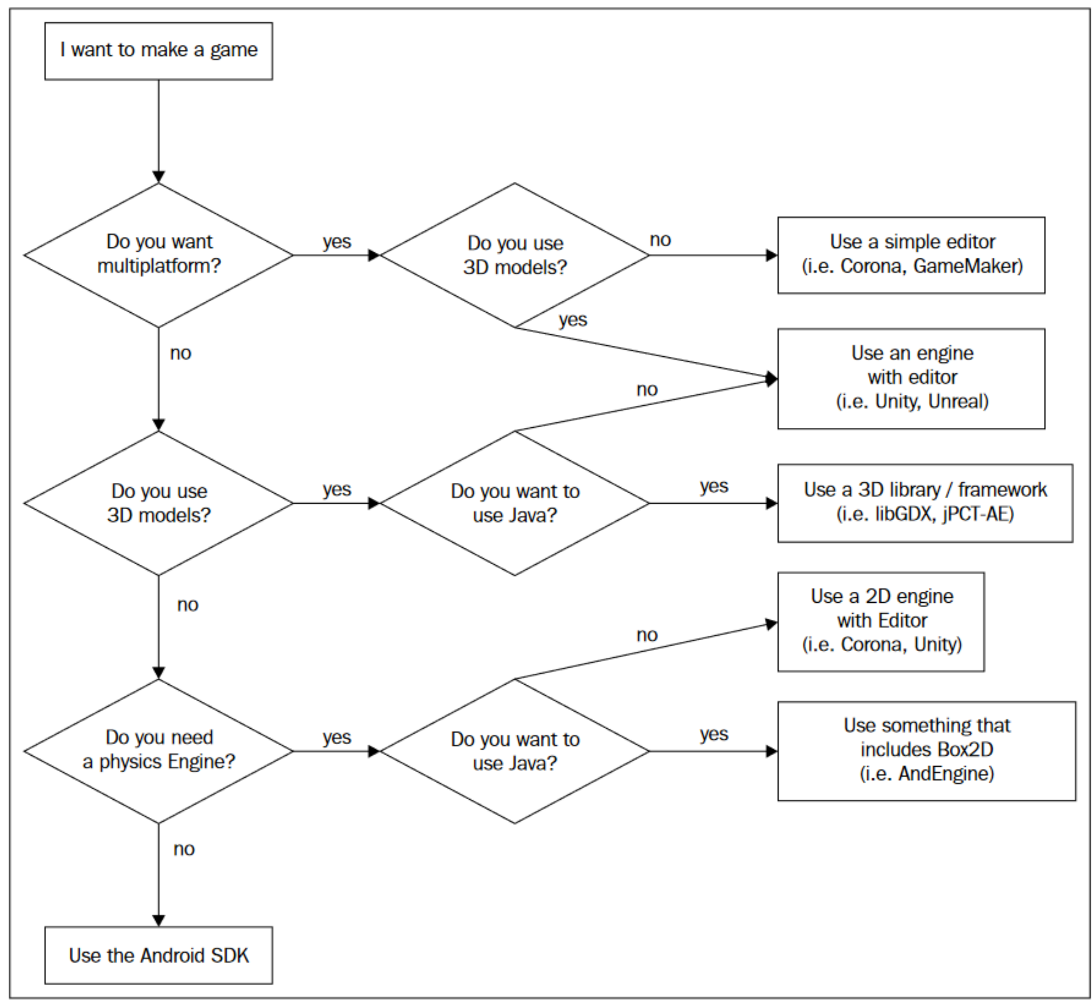
0) Crítica al estado del arte

Una vez expuesto un contraste entre las diferentes etapas de la historia de los videojuegos móviles, es imprescindible juzgar el estado actual de este mundo.

Con un claro duopolio absoluto entre las compañías Google y Apple, la forma en que la gran mayoría de videojuegos toman vida es a través de sus políticas de negocio. Estas dos empresas controlan cómo sus sistemas evolucionan, y son capaces de alterar su relación con los desarrolladores de manera unilateral, sin ni siquiera temer a las consecuencias. Existen varios ejemplos claros de esto, a gran y a pequeña escala, y es que la naturaleza de ambos ámbitos es distinta, y tiene que ser tratada como tal.

Uno de los casos más sonados del año 2020 fue el cómo Epic, una de las desarrolladoras con mayor facturación a costa del mercado móvil, se lanzó a una guerra comercial en la que arrastró a sus clientes como rehenes, e incluso pintó esta campaña de un extraño tinte pseudorevolucionario, haciendo que una disputa que debió resolverse en los tribunales un fenómeno social dirigido por grandes inversores [20]. Por suerte para pequeños desarrolladores, esto les ha acabado salpicando, ya que después de la mala imagen que generó para ambas compañías esta discusión, Apple ha movido ficha reduciendo hasta en un 50% la comisión que cobra a los estudios más modestos por las ganancias que generan sus propias obras [21].

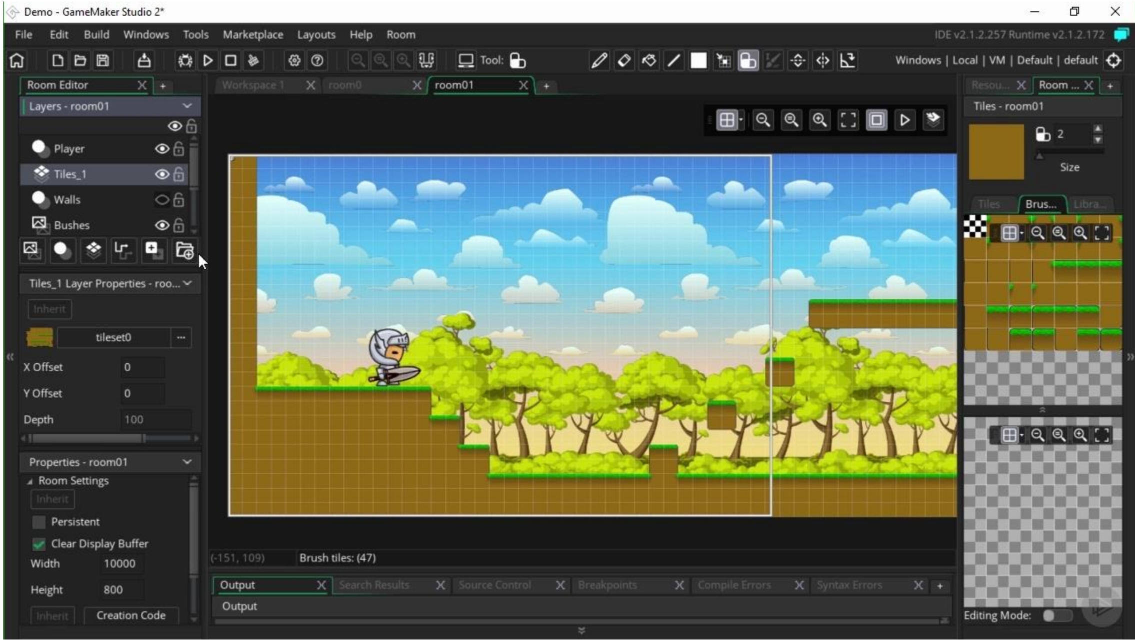
Pero lejos de asuntos empresariales, ahora se hará una breve exposición de cuáles son los métodos y herramientas que permiten la realización de un videojuego hoy en día. John Horton y Raul Portales hacen un buen trabajo ilustrando un diagrama que representa prácticamente todas las opciones por las que podemos optar para realizar un videojuego.



Editor sencillo de videojuegos

GameMaker Studio, así como RPG Maker, son grandes ejemplo de pequeños entornos de desarrollo sin demasiadas pretensiones que se limitan a cumplir con una serie de características básicas con las que es posible hacer videojuegos más bien modestos.

Estamos hablando de herramientas en las que el grueso del desarrollo consiste en elementos "arrastrar y soltar", y donde la programación tiene un papel muy secundario. De hecho, para el caso de GameMaker Studio, el único lenguaje integrado en la plataforma es el Game Maker Language. Pero este lenguaje de programación tan solo puede ser usado para los objetos de la propia herramienta, y es muy restrictivo, por lo que no tendría sentido siquiera considerar aplicar técnicas avanzadas de ingeniería software. Estas están ya integradas, y el desarrollador se limita a definir cómo quiere usarlas.



Motor de videojuegos 3D / 2D

Los motores que encontramos englobados en esta categoría son considerablemente superiores en capacidades. Los dos principales exponentes de esta categoría son Unity y Unreal Engine.

Ambos están orientados a un uso profesional, considerando en su diseño también facilidades para que desarrolladores independientes sean capaces de trabajar sin invertir un elevado esfuerzo.

La principal desventaja que presentan estos motores se discute numerosas veces a lo largo de la redacción de este documento. Estos motores están pensados para servir a todos los géneros posibles de videojuego, pero al no poder conocer con exactitud la especificación de los productos que serán realizados con esta plataforma, es imposible realizar una

implementación que descarte los módulos de mayor gasto de recursos, obligando al usuario a ejecutar un completo sistema repleto de funcionalidades para quizás ejecutar una versión del videojuego móvil Snake.

Tanto los motores más simples como los más refinados tienen dos problemas que se cumplen en mayor o menor medida, y estos son lo que se exponen a continuación.

- Su estructura supone la existencia de restricciones de diseño: Es evidente ver, tan solo al visualizar la interfaz de desarrollo, que es imposible implementar solo parcialmente las utilidades que ofrece cada entorno. Esto implica que cada uno tiene su manera de trabajar, y es un paradigma de programación en sí mismo, lo que nos lleva al siguiente punto.
- Un nuevo entorno que aprender: Aunque una vez superada la curva de dificultad inicial el desarrollo tome velocidad, un ingeniero del software que haya aprendido a trabajar directamente con decenas de lenguajes de programación ha adquirido una serie de conocimientos para resolver los problemas específicos de construir nuevos sistemas, o modificar directamente el sistema con el que trabaje. Esto no es así en los motores presentados, cuyo sistema es cerrado y no puede ser alterado a voluntad del programador. Y esto último tiene que ver con el punto final.
- Limitaciones de licencia: Los motores de videojuegos comparten una característica clave en común, y es que son esencialmente productos; productos por los que sus desarrolladores buscan ganarse un sueldo, que llevan a condiciones poco generosas con el desarrollador final del videojuego, y que empujan a gastar sus versiones *freemium* descafeinadas. Para estas tecnologías siempre existe una barrera de pago, la cual, si no es tumbada a base de financiación, siempre limitará el producto final.

Framework de gráficos

Personalmente, soy defensor de hacer el trabajo a mano, como he ido dejando entender en los anteriores puntos, y esto llevado al extremo es el caso de OpenGL, Vulkan, DirectX y demás librerías y frameworks de gráficos.

Esta opción es genial para construir productos ultra exigentes, junto con equipos considerablemente grandes, y cuando se cuenta con importantes cantidades de recursos.

Para el caso del desarrollo de una aplicación Android, y dadas las limitaciones de tiempo existentes para este proyecto, se ha desestimado esta opción, pues supondría gastar un tiempo excesivo no ya para realizar un videojuego o un motor, sino para realizar la implementación básica que permita mostrar una aplicación corriendo en pantalla. Esto es sencillamente demasiado trabajo para el alcance de este proyecto

Android SDK

Para diseñar cualquier tipo de aplicación Android, la opción recomendada por Google es Android SDK. Estas iniciales se traducen al español como kit de desarrollo software, y el caso de Android SDK incluye depurador de código, bibliotecas de librerías oficiales, emuladores de terminales físicos Android, y recursos como documentación o ejemplos de código. Esto hace perfecta simbiosis con el entorno de desarrollo integrado Android Studio.

Tomando como lenguaje Java, o Kotlin si es preferido por el desarrollador, este conjunto de herramientas permite una forma sencilla y nativa de desarrollar aplicando directamente código específico de Android.

1) Propuesta

A la hora de la verdad, para cualquier equipo, la solución más sencilla para enfrentarse al tipo de desafío que plantea este Trabajo de Fin de Grado sería el de escoger el motor de videojuegos con el que los desarrolladores se sintiesen más cómodos trabajando, y adaptarse a los límites y exigencias que impone este sistema.

El mayor conveniente que pudiese presentar esta opción consistiría en que no existe una sintonía entre el producto que se desea desarrollar y las capacidades del motor de videojuegos, como ya se ha desarrollado antes.

Pero para un profesional que haya estudiado ingeniería del software, la opción que presenta mayores ventajas sería la de diseñar y construir un motor pensado a propósito para satisfacer los requisitos del proyecto en mente. Esto por supuesto tiene sus inconvenientes. Es cierto que un estudio pequeño o, de una manera más acusada incluso, un desarrollador independiente, cuenta con unos recursos financieros y temporales más limitados al de una gran desarrolladora, por lo que es posible que recurrir por norma a un motor de videojuegos signifique la mayor cuota de eficiencia en la relación esfuerzo - resultado.

Por otro lado, esto podría mitigarse en el caso de que se reaprovechase el esfuerzo invertido. Si un estudio desarrolla videojuegos que guardan similitudes, y se invierte el tiempo necesario para analizar los requisitos que guardan en común este tipo de videojuegos, es posible construir un motor que, sin perder las ventajas de la optimización exhaustiva, permita abstraer módulos que puedan ser intercambiados con facilidad.

Pero siempre que se busque sacar el máximo partido a las características de la máquina en que se desea lanzar el producto, y sea posible realizar este esfuerzo, se hace especialmente útil contar con un motor propio.

Este es precisamente el tema central del proyecto que se está realizando, construir una arquitectura software que supere en eficiencia a Unity, para el caso particular de Psycho Cactus. A lo largo del punto de arquitectura serán expuestos los puntos clave del producto que se ha logrado desarrollar, y ejemplos de cómo cada herramienta ha sido concebida para resolver un problema específico derivado de la especificación del videojuego.

3. Arquitectura software

Diseñar e implementar la arquitectura software del juego que se ha ido describiendo previamente se presenta como el mayor reto a la hora de realizar este proyecto. Comenzaremos por describir lo que se entiende como arquitectura software.

Con los requisitos dados, es decir, realizar un juego haciendo uso única y exclusivamente de las librerías nativas disponibles en el desarrollo Android, partimos de un buen conjunto de herramientas estructuradas que permiten: manipular cada pixel de la imagen que se muestra en pantalla, reproducir sonidos por cada uno de los canales de audio, conducir los diferentes hilos de proceso para tanto coordinar las tareas a ejecutar como exprimir las limitaciones del hardware, y construir complejas estructuras de datos que representen objetos lógicos del mundo que se desea ilustrar.

Java es considerado un lenguaje de alto nivel. A pesar de que en la actualidad se han popularizado múltiples lenguajes y tecnologías, como el lenguaje interpretado Python, en que la gestión de los asuntos relativos al sistema operativo o la capa de hardware pasan a un segundo plano, Java no deja de ofrecer funcionalidades destinadas a abstraer las complejidades de las distintas máquinas en que corre su código, y está diseñado para que los desarrolladores puedan pasar a implementar las cuestiones que atañen a sus propios sistemas.

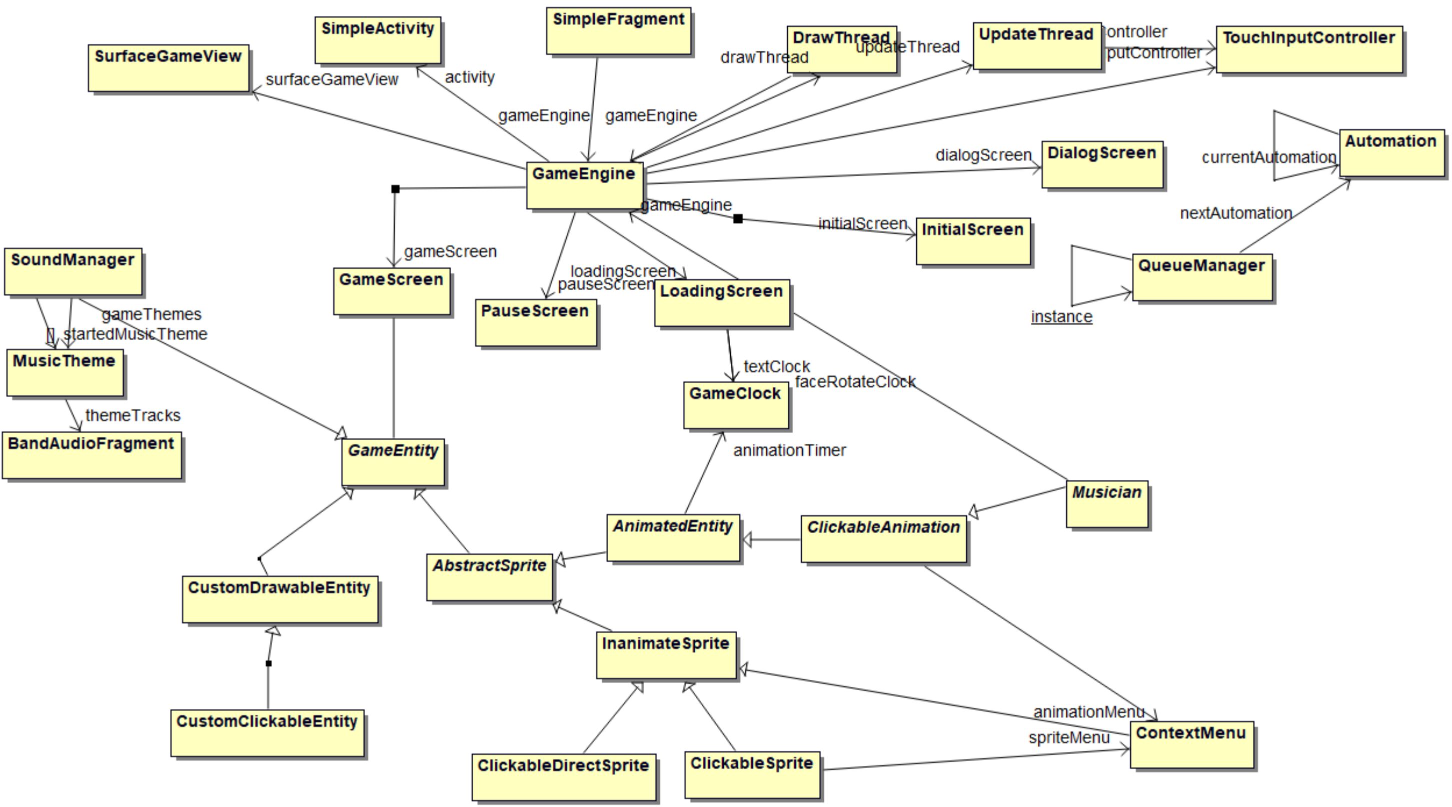
Dicho lo anterior, la arquitectura software es un concepto que generalmente hace referencia al diseño de la estructura del código. Es una solución a la problemática de organizar los diferentes componentes lógicos que interactuarán entre sí durante la ejecución del programa. De este modo, es posible escalar en complejidad sin temor a obtener un código ininteligible e inmantenible, en que la calidad del software permanezca sólida a la hora de implementar nuevas funciones.

Ahora, con los términos ya definidos, resulta mucho más sencillo discutir el alcance del proyecto. Lo que se va a tratar en el presente punto, la arquitectura software, será precisamente esto. Serán descritos todos los componentes desarrollados, el sistema que los gestiona, el cómo se prevé dejar espacio a nuevas características, y el por qué las decisiones adoptadas resultan las más adecuadas para el tipo de producto que se desea realizar.

Como se ha sugerido en anteriores apartados, el objetivo de este Trabajo de Fin de Grado no es el de obtener un juego comercial, sino el de construir los pilares fundamentales del software que hará posible que este desarrollo continúe en el futuro. Por este motivo, se hará hincapié en las implementaciones del motor, y no tanto en la lógica interna que sigue la ejecución del juego; es decir, la implementación de las “normas” del juego diseñado, así como podríamos hablar de las normas del parchís.

Especificación superficial de la arquitectura

Las dimensiones del motor diseñado son algo que dificulta mostrar su funcionamiento desde un punto de vista general sin entrar en el detalle particular de cada uno de sus componentes. Aún así, como pista para tener una idea durante la lectura, y que resulte más sencillo entender el trabajo realizado, se describirá muy superficialmente cómo los diferentes componentes están interrelacionados para ofrecer las características que han sido soportadas.



Este es el aspecto que guarda el mapa general de los componentes más relevantes.

En un primer lugar, tenemos una clase GameEngine que centraliza todo flujo de proceso de los datos. Esta cuenta con referencias a las diferentes escenas, que son las pantallas que se mostrarán en el videojuego; pero también a los elementos más cercanos a Android: la actividad, el fragmento y el elemento raíz del layout usado por la aplicación, SurfaceGameView. GameEngine también cuenta con referencias a los dos threads de proceso y el gestor de controles. Todo esto nos servirá para dotarnos de una instancia general que representa el juego en su conjunto.

La pantalla GameScreen, la principal, cuenta con referencias a múltiples objetos GameEntity. Estos son los controladores que mostrarán los personajes y controlarán su estado. Tenemos un árbol de herencias perteneciente a estos elementos que será detallado más adelante, y algunas de estas clases guardan referencias de ContextMenu, una de las entradas principales de las acciones que querrá realizar el usuario.

Por otro lado, tenemos SoundManager y sus otras dos clases relacionadas, que gestionan la música del videojuego.

Por último, vemos un grupo de dos clases aisladas del resto, cuyo principal gestor es QueueManager. Sirve para controlar los scripts y ponerlos en cola, para ejecutarlos uno tras otro.

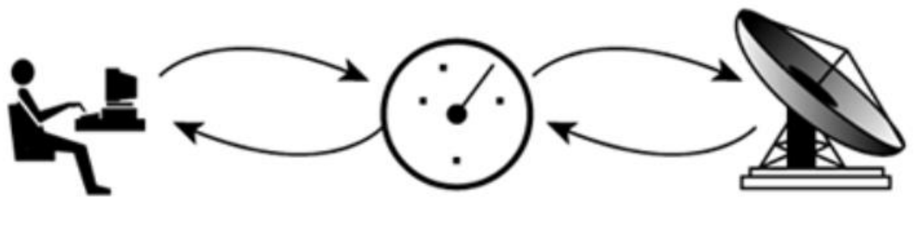
Para cerrar este breve apartado, se debe aclarar que aquí solo han sido representados los elementos más importantes, y que el resto serán descritos a lo largo de esta memoria.

0) Sistema de gestión central

En productos software existe un concepto arquitectónico llamado "*sistema de tiempo real*". En palabras de Daniel Sánchez-Crespo, autor de '*Core Techniques and Algorithms in Game Programming*', "software de tiempo real hace referencia a aplicaciones informáticas que

son de una naturaleza de tiempo crítico o, más generalmente, aplicaciones en que la adquisición de datos y la respuesta deben ser realizadas bajo unas restricciones temporales" [10]. Este tipo de diseño software tiene aplicaciones prácticas y vitales en el mundo real. Algunos ejemplos de estas aplicaciones serían un programa de control de vuelos, un robot

Figure 2.1. Air traffic controller.



mecanizado en una cadena de producción, o el sistema informático de un coche. Las características que comparten estos casos tan diversos son efectivamente las mismas. Estos sistemas necesitan responder en un tiempo crítico y reaccionar ante distintos eventos.

Un videojuego guarda exactamente estos mismos requisitos. Son un tipo de producto que debe mostrar suficientes refrescos de pantalla por segundo para producir la ilusión de animación, debe reaccionar ante las acciones del jugador, y debe ejecutar sus tareas en un margen de tiempo determinado, para las normas sean sólidamente cumplidas. De lo contrario, pueden producirse bugs, o los conocidos como '*glitches*'; famosos fallos de programación que los jugadores explotan para obtener una ventaja de manera inapropiada.

Para dar soporte a este concepto, debe ser diseñada una estructura que sea independiente de la velocidad de reloj de la máquina en que se ejecute. Cuando la aplicación no está iniciada, es necesario llevar a cabo una serie de tareas para preparar la ejecución de este sistema.

Mecanismo de inicialización de la aplicación

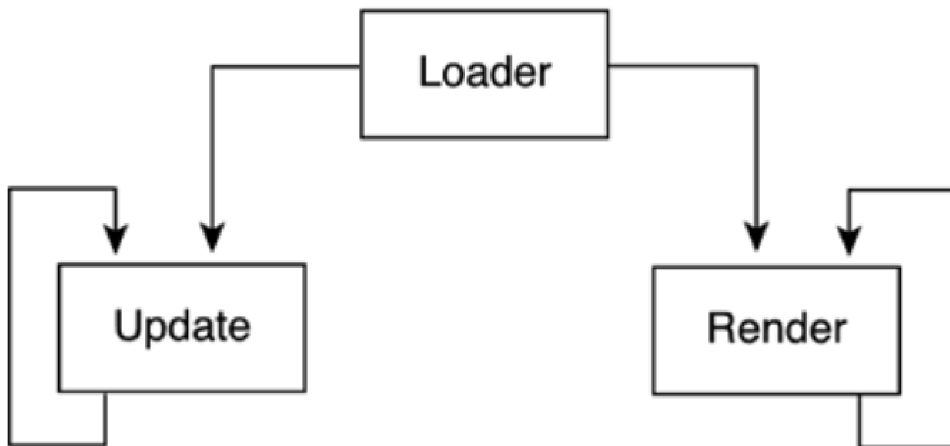
Los detalles acerca de las características propias de Android, y cómo este sistema operativo organiza sus recursos, será contado más adelante. Ahora nos centraremos en las propiedades del motor escrito en Java.

En primer lugar, Android requiere especificar un punto de partida, descrito en "AndroidManifest.xml". Esto llamará a la clase SimpleActivity, la especificada como principal al iniciar la aplicación. La actividad a su vez arranca un fragmento, y esto desencadena la serie de llamadas que se describe a continuación.

La clase que centraliza la gestión de la ejecución de la aplicación es la de GameEngine, que se traduce del inglés como motor de juego. Su constructor inicia y vincula las referencias relativas a las diferentes escenas de las que consta el videojuego, ajusta las medidas de

renderizado gráfico en función del tamaño de la pantalla del dispositivo físico, inicializa los controles táctiles y, por último, arranca el juego.

Esto consiste en establecer el estado inicial de todas las entidades que participarán en la partida, y lanzar los dos hilos principales del sistema: el de actualización de gráficos y el de actualización de estados, definidos por las clases DrawThread y UpdateThread.



Aunque GameEngine también realiza algunas tareas más, como la de servir como puente entre las clases nativas de Android y el resto del videojuego, tiene como función principal gestionar las transiciones entre escenas y contener sus referencias.

Las clases DrawThread y UpdateThread

Estas dos clases son realmente sencillas. Sin embargo, abstraen un mecanismo que guarda una considerable profundidad. Ambas implementan la interfaz GameThread, consistente de tres métodos: start(), stop() e isRunning().

Ambas han sido implementadas haciendo uso de una tercera clase utilitaria, desarrollada específicamente para gestionar los hilos de la aplicación, la clase GameClock. Aunque se describirá más adelante, esta clase permite que los hilos de actualización del juego programen sus tareas para ser ejecutadas repetidamente en periodos de tiempo constantes, y definidos en su clase.

Una vez realizado este trabajo, los recursos han sido preparados y la aplicación está lista para ser mostrada en la pantalla del usuario. El siguiente paso en el recorrido sobre la gestión central de la aplicación es el de explicar el trabajo que es desencadenado en cada uno de los ciclos de actualización del videojuego. En todo momento, la estructura ha sido diseñada para mantener separado el dibujado de la manipulación de estados, y siempre que los dos hilos actúen sobre las mismas referencias, es establecida una sincronización entre estos.

Mecanismo de dibujado

Cada ciclo de dibujado tiene una única tarea por realizar, la de llamar al método draw() ubicado en la clase SurfaceGameView. Este método contiene el fragmento de código que manipula directamente el fragmento usado por la aplicación, y sigue los siguientes pasos:

- Obtiene la instancia del objeto Canvas vinculado al fragmento, y lo bloquea.
- Dibuja el fondo por defecto para sobrescribir la imagen generada anterior.
- Dibuja la escena que actualmente esté activa.
- Aplica efectos gráficos, si los hay.
- Imprime elementos internos de monitorización si el modo de debug está activado.
- Escala la imagen a las dimensiones del dispositivo físico.
- Libera los recursos gráficos del Canvas y aplica finalmente la nueva imagen generada.

```java
138    public void draw() {
139        // Abandona el método si el SurfaceHolder no está disponible
140        if (!this.surfaceReady) {
141            return;
142        }
143        // Obtiene la referencia al canvas nativo y lo sincroniza
144        Canvas screen = getHolder().lockCanvas();
145        if (screen == null) {
146            return;
147        }
148        // Dibuja el fondo y sobrescribe el anterior frame
149        this.backgroundSprite.draw(screen);
150        // Recupera la escena actual y la dibuja
151        synchronized (GameEntity.entitiesLock) {
152            this.gameEngine.getCurrentSceneDrawable().draw(this.frameCanvas);
153        }
154        // Aplica efectos al bitmap obtenido
155        this.applyEffects();
156        // Dibuja el último mensaje de debug, si este modo está activado
157        if (GameEngine.DEBUGGING) {
158            this.frameCanvas.drawText(DebugHelper.getLastDebugComment(), 20, 620, this.basicPaint);
159            GameEngine.getInstance().getDebugHelper().draw(this.frameCanvas);
160        }
161        // Reescala el frame de juego y lo posiciona en la pantalla del dispositivo
162        Bitmap scaledBitmap = Bitmap.createScaledBitmap(this.frameBitmap,
163                this.adaptedWidth, this.adaptedHeight, false);
164        screen.drawBitmap(scaledBitmap, this.basicMatrix, this.basicPaint);
165        // Plasma el frame obtenido tras aplicar el dibujado de los elementos
166        getHolder().unlockCanvasAndPost(screen);
167    }
```

Todos estos pasos son realizados de manera constante a cada nuevo ciclo del hilo de gráficos. Pero una de estas tareas es especialmente interesante, pues es la que se encarga de intercambiar las pantallas que se muestran a lo largo de la ejecución del videojuego. El concepto de escena y su integración en el motor será descrito más adelante en la memoria.

Mecanismo de actualización de estados

El caso de los ciclos de actualización de estados es algo más sencillo conceptualmente, pero guarda algunas complejidades que conviene resaltar. Cada ciclo de actualización tiene tres tareas: la de leer el último estado de los controles, la de efectuar las actualizaciones de entidades de las que dispone la escena activa, y la de cambiar de escena si es preciso.

Para el primer caso, sabemos que cuando el usuario hace *click* en un lugar de la pantalla de su dispositivo, y la aplicación está abierta, queda registrada la coordenada donde sucedió esto. Así pues, el hilo de actualización recoge este punto y se lo entrega a la escena que

actualmente se encuentre seleccionada, aislando de una manera elegante la gestión de controles según la escena que el usuario esté visualizando. Los detalles técnicos que permiten este comportamiento para la clase TouchInputController será también explicado más adelante.

Para la segunda tarea, también será tomada la referencia a la escena activa, y será ejecutado su método para manipular las entidades vinculadas a esta. Es decir, tal como está planteado el motor, las entidades nunca son añadidas en medio de un ciclo, sino que se pide sincronizar el *lock* de entidades, y se procede a alterar este listado cuando no se está leyendo.

En último lugar, el bucle de actualización también consultará si se ha solicitado un cambio de escena a GameEngine. De existir, llama al método de transición entre escenas y ejecuta uno de los métodos presentes en la interfaz Scene, *onSceneChange(SCENE_TYPES oldScene)*.

Con esto, el esqueleto fundamental de la arquitectura en cuanto a hilos ha sido expuesto. Pero muchos de los componentes que intervienen en estos procesos no han podido ser desarrollados en detalle, puesto que son numerosos, y cada uno exige un tratamiento aparte. Por este motivo, a continuación, se expone uno de los elementos también fundamentales para hacer posible un videojuego de gran envergadura, las escenas.

1) Escenas

En cualquier motor gráfico, es común encontrar un concepto conocido como escena. Cada escena contiene toda la información que representa y gestiona una pantalla distinta. De esta forma, podríamos dividir las pantallas de nuestra aplicación entre pantalla de carga, menú principal, niveles jugables y pausa. Adicionalmente, en este videojuego se ha considerado útil la adición de la escena de diálogos, ya que se suele mostrar texto en pantalla muy a menudo.

```
this.gameScenes = new HashMap<SCENE_TYPES, Scene>() {
    {
        put(SCENE_TYPES.GAME, gameScreen); put(SCENE_TYPES.PAUSE_MENU, pauseScreen);
        put(SCENE_TYPES.DIALOG, dialogScreen); put(SCENE_TYPES.LOADING, loadingScreen);
        put(SCENE_TYPES.INITIAL_SCREEN, initialScreen);
    }
};
//
```

¿Y qué define una escena exactamente? En el motor desarrollado existe una interfaz llamada Scene, la cual obliga a implementar una serie de métodos, de entre los que destacan los siguientes:

- Drawable definedDrawable(): Devuelve una instancia de Drawable, una interfaz funcional que, dado un Canvas, contiene un conjunto de instrucciones que imprime elementos en pantalla.

- Touchable definedTouchable(): Devuelve una instancia de Touchable, que de nuevo es otro conjunto de instrucciones para la captura de los controles del usuario, dadas las coordenadas en que el usuario ha pulsado.
- void onSceneChange(SCENE_TYPES oldScene): Un método que realiza una serie de ajustes para preparar el arranque de una nueva escena después de abandonar la escena anterior. Como parámetro, se pasa el tipo de escena que estaba activa previamente, para poder preparar la transición correctamente.
- void updateSceneEntities(): Método que será ejecutado recurrentemente por el hilo de actualización de estados. Es thread safe, y garantiza que todas las entidades del juego que sean añadidas, eliminadas, o modificadas, mientras la escena está siendo visualizada, serán procesadas correctamente.

Las principales escenas que dan soporte a la aplicación son inicializadas y contenidas en GameEngine, como se ha mencionado antes. GameEngine, por lo tanto, también dispone de un método para recuperar la referencia a la escena activa. Ya que las escenas son clasificadas por tipos y almacenadas en un HashMap (que usa como clave el tipo de escena, y como valor la referencia a la interfaz de la escena) para recuperar la escena actual basta con guardar una variable que indica qué tipo de escena está abierta y luego recuperarlo del HashMap indicando este parámetro como clave. Este sistema después es usado por todas las clases que interactúan con las escenas, que ya no deben preocuparse de buscar con cuál desean tratar.

2) Controladores de entidades

Para realizar el videojuego que se está desarrollando, la arquitectura debe permitir gestionar la lógica y los datos vinculados a las diferentes entidades que existirán en cada partida. Conocemos que las habrá de diferentes tipos y naturalezas, pero todas comparten unas u otras características en común que hacen deseable la abstracción donde sea posible. De esta manera, han sido identificados los siguientes tipos de capacidades:

- Que sea dibujable.
- Que conste de unas dimensiones.
- Que sea animada; es decir, muestre diferentes imágenes en función del tiempo.
- Que sea *clickable*; es decir, que sea posible interactuar con la entidad.
- Que dispongan de un menú contextual mediante el cual realizar acciones.

Veamos esto con un ejemplo. Los personajes, y en particular los músicos, son tipos de entidades vitales en el videojuego. Queremos mostrarlos en pantalla, moverlos, interactuar con ellos, y además tienen varios estados, en que cada estado correspondería con una animación diferente. En cambio, los menús contextuales, a los que los anteriores músicos están vinculados, son también dibujables e interactuables, pero no son animados. De hecho, y esto se explicará más adelante, sí será posible aplicar efectos visuales sobre estos menús, a pesar de no ser animados. Pero la cuestión a destacar es que, una vez formada la imagen que el menú representa, no tendrá que ser reconstruida con cada ejecución del ciclo de dibujo, ahorrando unos pocos muy preciados milisegundos, cuyo desperdicio debemos siempre minimizar para preservar una alta tasa de refresco de imagen por segundo.

Habiendo sido analizadas las entidades que formarán parte de la versión final del producto, ha sido estimada la necesidad de implementar una serie de clases que será repasada a

continuación, junto con la declaración de algunas interfaces que nos permitirían extraer comportamientos de estos controladores para extrapolarlos a otros objetos del juego, como se darán ejemplos también. Estas clases siguen un orden jerárquico, en que gradualmente, y según se especializan los tipos de controlador a representar, su árbol de herencia se hace más grande.

Gracias a la herramienta de generación de *javadoc*, es sencillo representar esto gráficamente, tal como se ve a continuación.

Hierarchy For Package com.psychocactusproject.graphics.controllers

Class Hierarchy

- java.lang.Object
  - com.psychocactusproject.graphics.controllers.**AnimatedEntity.AnimationResources**
  - com.psychocactusproject.engine.manager.GameEntity
    - com.psychocactusproject.graphics.controllers.**AbstractSprite** (implements com.psychocactusproject.graphics.interfaces.Dimensions, com.psychocactusproject.graphics.interfaces.Drawable)
      - com.psychocactusproject.graphics.controllers.**AnimatedEntity**
        - com.psychocactusproject.graphics.controllers.**ClickableAnimation** (implements com.psychocactusproject.states.menu.MenuDisplay)
      - com.psychocactusproject.graphics.controllers.**InanimateSprite**
        - com.psychocactusproject.graphics.controllers.**ClickableDirectSprite** (implements com.psychocactusproject.input.Clickable)
        - com.psychocactusproject.graphics.controllers.**ClickableSprite** (implements com.psychocactusproject.states.menu.MenuDisplay)
    - com.psychocactusproject.graphics.controllers.**CustomDrawableEntity** (implements com.psychocactusproject.graphics.interfaces.Dimensions, com.psychocactusproject.graphics.interfaces.Drawable)
      - com.psychocactusproject.graphics.controllers.**CustomClickableEntity** (implements com.psychocactusproject.input.Clickable)

Según se puede comprobar aquí, todas las clases heredan de GameEntity, la cual no define demasiado comportamiento, más allá de obligar a implementar el método abstracto llamado getRoleName(), el cuál devuelve una String, que debe representar el rol que cumple cada instancia que pretenda ser una entidad del juego.

Las interfaces usadas en este paquete son las expuestas a continuación.

Dimensions

Una interfaz fundamental que es implementada por casi la totalidad de entidades del juego, que sencillamente exige especificar una posición y un tamaño, y poder consultar estos datos.

Clickable

Hereda de Dimensions, y además implementa varios métodos fundamentales para dar soporte al sistema de control de usuario. Obliga a las entidades que lo implementan a definir un comportamiento para los casos en que se haga *click* sobre estos objetos. Obliga también a devolver una lista de las Hitbox de las que consta la instancia, clase que será desarrollada en el punto en que se habla de los controles. Permite consultar si el objeto puede ser *clickado* en ese momento o no, y también obliga a definir los métodos para activar o desactivar las diferentes opciones que tiene la instancia vinculada.

MenuDisplay

Esta interfaz hereda de Clickable, incorporando todos los métodos desarrollados arriba. Los nuevos métodos que incluye esta interfaz son todos orientados a permitir que la clase que lo implemente pueda mostrar un menú contextual. Así pues, es posible solicitar, con tan solo disponer de la referencia al objeto que implemente MenuDisplay, métodos para abrir y

cerrar el menú, para obtener un listado de las opciones que se permiten, para dibujar el menú, y para comprobar si la instancia que contiene el menú permite en este momento ejecutar alguna acción.

Drawable
Como ya se ha sugerido antes, esta interfaz es de carácter funcional, e implementa un único método, el cual toma como parámetro una instancia de Canvas y permite dibujar el objeto en cuestión. Esta interfaz es implementada por todos los controladores, como cabe esperar, pero no es explícitamente implementada por GameEntity porque los objetos musicales también se consideran entidades del juego, y no son dibujables.

Desarrollado lo anterior, se procede a enumerar todos los controladores que forman parte del motor, junto con una breve descripción de cada uno de ellos.

AbstractSprite
Esta clase tiene como única función dar soporte a lo que es común para todos los controladores. Declara un método abstracto que obliga a toda su descendencia a devolver la imagen que represente el estado actual del controlador, y además guarda en su clase las variables que definen su dimensión, tal como especifican las interfaces que implementa.

AnimatedEntity
Clase base de cualquier objeto que desee contar con animaciones. Contiene en su interior otra clase anidada, llamada AnimationResources, la cual debe ser entregada instanciada a través del método abstracto *obtainAnimationResources()*. Esta referencia será usada por el constructor de AnimatedEntity, que asignará la acción por defecto, el nombre del personaje, que dará nombre a las acciones, les asignará sus imágenes, arrancará el reloj de su animación, y definirá varias variables auxiliares propias de la clase. Las imágenes son asignadas mediante descriptores de ficheros, que pueden ser encontrados organizados por carpetas en el directorio "*assets*". La subclase AnimationResources también define las diferentes acciones que puede realizar el personaje, y las relaciona con diferentes animaciones. De esta forma, AnimationResources centraliza la entrega de todos los recursos utilizados por el controlador, y por todos los controladores que heredan de este.

InanimateSprite
Como su nombre sugiere, esta clase toma el nombre y ubicación de una imagen (al contrario de la clase anterior, la cual requería de una numerosa cantidad de recursos a través de AnimationResources), la carga y la vincula a la instancia creada, de manera que su dibujado produce la impresión de una imagen constante. Como está también pensada para dar soporte a las utilidades de debug propias del motor diseñado, incorpora un constructor específico que recupera una imagen adicional para ser mostrada cuando el modo de debug está activado. La clase InanimateSprite también define el método de *resizeBitmap(int sizeX, int sizeY)*, el cual trata de transformar su propia imagen con los parámetros de tamaño pasados a la función.

ClickableAnimation
Una subclase de AnimatedEntity, la cual implementa adicionalmente la interfaz MenuDisplay, ya que está pensada para dar soporte a las principales entidades del juego.

Estas serían entidades que disponen de distintas animaciones, y son seleccionables haciendo *click*, lo que muestra un menú de acciones propios de la animación. Para su inicialización basta con crear una instancia de ContextMenu a la que se pasa la propia referencia de la ClickableAnimation creada. El resto del trabajo que realiza esta clase es implementar los métodos de la nueva interfaz usada.

ClickableDirectSprite

Así como la anterior clase descrita hereda de AnimatedEntity e implementa un menú, esta clase tan solo pretende servir como una imagen que al ser *clickada* realiza una acción. No dispone de menú ni define más comportamiento del que ya es heredado.

ClickableSprite

Esta clase, de un uso más extendido que la anterior, sí dispone de menú, pero no cuenta con animaciones. Es especialmente útil cuando imágenes estáticas son interactuables. Un ejemplo de uso de esta clase se encuentra en dos botones mostrados en la pantalla de juego cuando está activado el modo de debug, que abren su propio menú con las diferentes opciones declaradas durante su construcción.

Si encontramos el caso de algún objeto que deba ser implementado en el juego, y no cumpla estrictamente con las características predefinidas de las clases comentadas arriba, disponemos de dos controladores adicionales que cuentan con un grado de flexibilidad considerablemente superior, pero por el coste de obligar a definir prácticamente la totalidad de los métodos que deben ser implementados por un controlador.

CustomDrawableEntity

Esta clase toma como parámetros en su construcción, además de los habituales en los anteriores controladores, dos instancias de Drawable, que es la interfaz funcional vista arriba. Gracias a esta clase es posible, por ejemplo, implementar DebugHelper, una clase con la que leer entrada pasada por teclado a la aplicación e introducir comandos cuyo comportamiento es programado con antelación. DebugHelper guarda una referencia a CustomDrawableEntity, a la que después de indicarle un conjunto de instrucciones es capaz de imprimir el comando que está siendo introducido, y ser tratada como una entidad jugable más.

CustomClickableEntity

Esta clase, siguiendo el modo de trabajo que está siendo utilizado para el resto de los controladores, se limita a extender de la clase CustomDrawableEntity, pero implementa adicionalmente la interfaz Clickable, con la que es posible también definir un método de control único. Un ejemplo de uso es el que permite crear el botón de pausa, para el que basta con especificar en la instancia de Drawable que sea impresa la imagen del cartel de pausa, y como comportamiento a ejecutar cuando el objeto es modificado, se pide pausar el juego. Gracias a CustomDrawableEntity, no es necesario crear una nueva clase tan solo para definir la funcionalidad que requiere el botón de pausa.

Con estas dos últimas clases, hemos visto las posibilidades que ofrece el motor desarrollado para integrar cualquier tipo de entidad nueva al videojuego, independientemente de sus características.

La última de las clases reseñables en este punto no pertenece estrictamente al grupo de los controladores, pero es usado por todos los que implementan MenuDisplay. Esto se trata de ContextMenu.

Menús contextuales

Los menús contextuales son una parte fundamental del videojuego, ya que han sido diseñados como la principal vía para que el jugador indique los comandos que desea realizar. A un nivel superficial, lo que permite cualquier instancia de la clase ContextMenu es generar un menú contextual, que contiene una serie de opciones seleccionables, y las cuales pueden ser activadas o desactivadas según el estado de la lógica jugable. Es decir, si según las normas del videojuego el estado de un personaje impide tocar un solo mientras la fatiga permanezca elevada, bastará con añadir la orden para desactivar esta opción, y el usuario visualizará la opción con un tono distintivo grisáceo, que si es pulsada hará saltar una alerta indicando que la acción no se encuentra disponible.

Los menús contextuales ya integrados en las clases que implementan MenuDisplay son sencillos de crear. Ya que el constructor de ContextMenu exige que se entregue una instancia de MenuDisplay, que internamente será almacenada con el nombre de *father*, la gestión de estos menús es definida una única vez en esta clase, y los métodos públicos bastan para manipular todo lo que se desea de estos. La forma en que la clase ContextMenu almacena el estado actual de las opciones que contiene es mediante un array del tipo MenuOption, clase anidada que se encuentra definida en su interior, y que guarda la etiqueta de su opción y la información sobre si se encuentra disponible.

ContextMenu define como uno de sus métodos fundamentales *onUpdate()*, que será llamado al final del constructor de la clase y cada vez que uno de los parámetros que permiten su visualización y control cambia. De esta manera, cuando una opción es activada o desactivada, las variables de esta clase son calculadas de nuevo. Este método cumple dos tareas:

- Si las menuOptions no están definidas, son creadas por primera vez, usando valores extraídos de la implementación de MenuDisplay. Una vez asignada esta referencia, ya es posible activar o desactivar las opciones de ContextMenu.
- La siguiente tarea se realiza siempre, y es la de actualizar el Bitmap vinculado a la instancia actualizada. Ya que ContextMenu hereda de InanimateSprite, es tan sencillo como llamar al método *setBitmap(Bitmap bitmap)* de su clase directamente ascendiente. El bitmap vinculado, ahora sí, deberá ser generado de manera dinámica en función de los parámetros encontrados en la referencia de MenuDisplay *father*. En pocas palabras, recoge las imágenes que forman el marco del menú contextual, que han sido precargadas a modo de *flyweight*, y las coloca metódicamente para lograr obtener un Bitmap que, independientemente de las etiquetas pasadas y su longitud, es siempre capaz de crear una ventana de apariencia consistente.

Para dibujar esta clase no nos basta con utilizar el método definido por InanimateSprite, pero gracias a la flexibilidad del sistema de herencia de Java es posible sobrescribir este comportamiento para verificar una serie de condiciones antes de llamar al método de dibujado definido por su clase ascendiente. Estas comprobaciones son las de verificar si su

posición haya variado, en cuyo caso volveremos a tomar las coordenadas actualizadas de su instancia MenuDisplay vinculada (que podría ser un músico, por ejemplo); o verificar si por el tamaño del menú generado la imagen sale de los límites de la pantalla, en cuyo caso basta con cambiar su posición para continuar visualizando las opciones y poder seleccionarlas independientemente de la posición de los elementos en el juego.

Otras menciones notables que hacen uso del paquete de controladores pasan por dos de las clases anidadas en GameScreen llamadas RulerBar y FunBar. Ambas clases estáticas contienen una referencia de CustomDrawableEntity, que como se ha explicado antes permite definir un dibujado y controles únicos para estas instancias. El propósito de ambas clases es el de monitorizar el estado actual del concierto, dibujando cada una su propia barra de medición junto con una flecha, posicionada en función del estado que se desea mostrar. De nuevo, estas clases son una prueba de la sencillez de hacer crecer el juego que está siendo diseñado, reutilizando para una clase definida en no más de cuarenta líneas comportamientos definidos a lo largo de al menos cientos de líneas de código.

3) Reloj de tareas

Una de las herramientas fundamentales con las que debe contar este proyecto es una clase diseñada a propósito para servir como reloj. Como se ha declarado al principio del punto de la arquitectura del motor diseñado, un requisito del sistema es que permita operaciones en tiempo real; pero no suficiente con eso, además incorpora un sistema de música, que será descrito en el siguiente punto, y que es extremadamente sensible a pequeñas variaciones temporales, es decir, debe guardar una precisión literalmente milimétrica.

La decisión de diseñar una clase para este proyecto que gestione toda problemática relacionada con la gestión de hilos viene motivada por facilitar su mantenimiento. Al tratarse de una clase sobre la que tengo control, las actualizaciones en librerías, o los propios criterios que se adopten para modificar ligeramente el diseño, no afectarían en absoluto a la semántica externa definida, ya que se pretende que dotar de comportamiento a esta clase sin que nada de la implementación sea visible o manipulable desde fuera. Además, es más sencillo ampliar las funcionalidades que ofrece el sistema de gestión de hilos del motor si conceptualmente giran en torno a una única clase, definida expresamente para satisfacer las necesidades del motor.

Cuando antes se afirmaba que las clases descendientes de GameThread programan sus tareas para ser ejecutadas repetidamente, se está haciendo uso del método estático presente en esta clase, llamado *scheduleTask*. Este método presenta dos versiones, las cuales son idénticas, pero en que su versión con mayor número de variables pasadas por parámetro permite un mayor control sobre los ajustes de la tarea ejecutada. En particular, para cualquier tarea que se desee planificar es posible especificar su retardo inicial, la espera entre repeticiones, el tiempo restante hasta su cancelación automática, y una instancia de Runnable que representa las tareas que se desea llevar a cabo. Además, para casos en que conviene detener este reloj en un momento indeterminado, el método estático *scheduleTask* devuelve también un identificador único que se puede almacenar y entregar posteriormente a *cancelTimer*, que, tal como su nombre sugiere, recupera el objeto Timer vinculado al identificador y lo detiene inmediatamente, evitando futuras repeticiones de la tarea. El ejemplo más evidente de uso es

el del reloj que gestiona la reproducción de la música, que una vez salimos al menú principal, acaba la partida, o es minimizada la aplicación, detiene su Timer para evitar seguir reproduciendo la música.

Por otro lado, GameClock también define un comportamiento fundamental para dar soporte a las animaciones de cada entidad. Cuando una instancia de AnimatedEntity es creada, o cuando su acción vinculada cambia, es iniciado un nuevo reloj que almacena un índice, el cual incrementa progresivamente hasta alcanzar el número máximo de imágenes por las que está formada la animación en cuestión. Cuando una instancia animada solicita su imagen actual, la clase accede al array de imágenes propio de la animación que está siendo mostrada, y recupera la imagen adecuada para el instante de tiempo en que se visualiza. Con el paso de los segundos, ya que GameClock se actualiza constante a la velocidad especificada (la cual para los músicos depende de la velocidad de la música que se reproduce), se podrá observar una animación formada por las distintas imágenes que componen cada acción de un personaje.

4) Gestor de música dinámica

Una de las características fundamentales de este proyecto, que ha orientado el proceso de toma de decisiones, y en la que se ha querido incidir especialmente, es la capacidad de generar música de una manera completamente dinámica; que cada pista, cada músico, cada fragmento, y cada tema, sean variables y dependientes del estado actual de la partida, en función de las acciones que decide realizar el jugador.

Esto plantea un gran reto. La idea de la que se parte para definir los requisitos con que debe cumplir el motor desarrollado es que los músicos del videojuego representan una banda real de música. Y por ser ellos los protagonistas, hemos de poder interactuar con ellos, pedirles realizar acciones, y en general exigirles que dejen de tocar su instrumento para dedicarse a otra tarea. Por este motivo, se hace impensable la posibilidad de que los temas vengan completos, tal como han sido compuestos. Es necesario dividirlos según los siguientes dos requisitos.

Temas fragmentados

A lo largo del transcurso de la partida, la banda toca un total de cuatro temas distintos, distribuidos entre los veinte turnos que dura el juego. A nivel de tema musical, es decir, de pieza compositiva, no se debe permitir que un fragmento dure por un tiempo prolongado una vez la partida exija que se pase a tocar la siguiente pista. Si se alcanza el turno número seis, será puesto en cola de reproducción el segundo tema del juego, y de la misma forma sucede con el tercer tema en el turno once y el cuarto en el turno dieciséis. En el caso contrario, es más que probable que si el jugador no logra alcanzar alguno de los turnos indicados antes de que la pista finalice, debe arrancar de nuevo sin que pueda percibirse una mínima interrupción.

Ante este escenario, se hace vital partir la música por secciones. Estas particiones son comunes y naturales en el análisis musical, lo que permite implementar este sistema sin alterar significativamente las sensaciones que transmite la reproducción de la música del videojuego. Ya que este sistema fue analizado y diseñado con antelación para poder identificar qué características debía cumplir cada composición, su realización no ha supuesto complejidad adicional.

Una vez obtenida una pista separada por fragmentos, es posible crear un sistema de gestión de sonido que avance al siguiente fragmento de la pista actual o inicie la pista siguiente indiferentemente.

Aislamiento de los instrumentos

Como se ha mencionado anteriormente, los músicos son los protagonistas de este videojuego, y ocasionalmente deben dejar de tocar su instrumento, por lo que encontramos nuevas divisiones.

Por una parte, queda claro que cada pista y fragmento debe estar también dividido por los músicos presentes en el videojuego, que son un total de cinco: batería, bajo, guitarra, voz y teclado. Además, ya que un músico puede en cualquier momento tener que reaccionar a la acción que el usuario solicita realizar, no es posible dejar sonando la pista hasta que finalice el fragmento, sino que hay que detenerla inmediatamente. Esto obliga a que las pistas deban ser divididas a un nivel inferior que fragmentos, quedando separadas incluso a nivel de compás.

Estos son los requisitos con que debe cumplir la música entregada al motor del juego, pero este sistema se debe poner en práctica a través del código. Esto será descrito a continuación.

Diseño e implementación

Siguiendo los criterios desarrollados para lograr música dinámica, el paquete del proyecto dedicado al sonido cuenta con tres clases distintas, en que tenemos SoundManager, MusicTheme y BandAudioFragment.

SoundManager

Es, en primer lugar, la clase ideada para centralizar la gestión de todo el sonido del videojuego, incluyendo también los efectos; aunque en el estado actual del proyecto se ha priorizado de forma exclusiva la gestión de la música.

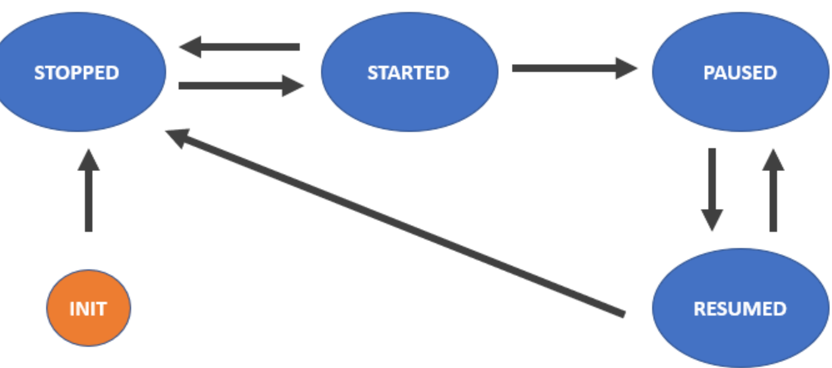
Esta clase inicializa en su constructor una referencia a SoundPool, una clase nativa de Android que es capaz de reproducir fragmentos pequeños de sonido, de las mismas características que la música que existe en el proyecto. Cuenta además con diferentes variables internas que le permiten monitorizar el estado actual de la pista en reproducción, y dispone de un array con las referencias a cada uno de los cuatro temas, encapsulados en instancias de la clase MusicTheme.

Los métodos públicos implementados en SoundManager son el punto de entrada a toda manipulación que se desee realizar tras realizar en cualquier momento un cambio del estado de la partida que pueda alterar la reproducción de la música. Así que, además de los habituales *getters* y *setters*, encontramos los siguientes.

- *startMusic()*: Tan sencillo como su nombre indica, arranca la música por primera vez. Pero al igual que el resto de los métodos dedicados a alterar la reproducción, guarda una restricción que no puede ser ignorada. La música solo puede ser arrancada si previamente se encontraba detenida. Después de arrancar la música, sonará el primer fragmento del tema activo en ese momento.

- *stopMusic()*: Este método, al contrario, detiene la música. Solo puede ser llamado cuando la música está sonando, ya sea tras salir de una pausa o detención, y de lo contrario lanzará una excepción. Además de impedir la reproducción de la música cuando es ejecutado, reinicia el estado de la pista actual, lo que significa que al iniciarla de nuevo comenzará a sonar desde el principio.
- *pauseMusic()*: La pausa está concebida de manera similar a la detención, pero para casos en que conviene reanudar posteriormente la música por el mismo punto. Este método no pierde el estado en que se encuentra la pista, pero silencia y detiene la reproducción momentáneamente. Al igual que la detención, debe ser llamado cuando la música está siendo reproducida.
- *resumeMusic()*: Método complementario a la pausa, y solo disponible cuando la música ha sido pausada. Reanuda las tareas de reproducción por el mismo lugar que habían quedado previamente.
- *setCurrentFragmentVolume()*: Cuando el volumen de la música cambia en el menú de opciones, no solo son reproducidas con menor volumen las próximas pistas, sino que también debe ser alterado el volumen de la música que suena en el momento actual. Esta es la función que cumple este método, que acude directamente a la referencia del tema actual, y de ahí, al fragmento actual, para ajustar el volumen de todos los clips de audio en reproducción.
- *nextThemeRequest()*: Llamado cuando la partida ha avanzado lo suficiente como para poner en marcha el siguiente tema del videojuego.

Para ilustrar con mayor facilidad la secuencia de estados que el motor permite recorrer, es posible representar estas relaciones con una especie de autómata muy simple, que se muestra a continuación.



El resto de los métodos podrán ser llamados sin riesgo a interrumpir la ejecución del videojuego, y responderán al estado actual de la partida.

En cuanto a la gestión de pistas que realiza esta clase, que tal como se ha dicho es completamente dinámica, se encargan dos métodos privados que son ejecutados de forma transparente, y los cuales arrancan o detienen una tarea del GameClock clase ya expuesta anteriormente. Aquí, es la tarea que se envía al reloj la que requiere más atención.

En primer lugar, cada tema cuenta con una velocidad distinta, que se mide musicalmente en BPM, o *beats per minute*, que en español se traduciría como latidos por segundo. Es el número de notas negras que suenan a lo largo de un minuto. Pero esta velocidad debe traducirse a una medida que pueda tratar el reloj, ya que este está preparado para tomar el plazo de tiempo que hay entre una ejecución y la siguiente. Por eso, ha sido calculado externamente el número de milisegundos que dura cada compás y ha sido almacenado como array estático de la clase MusicTheme.

Cada fragmento del motor cuenta con uno o dos compases, con lo que a la finalización de cada compás comprobaremos si hay compases restantes en el fragmento actual. Si no los hay, pasaremos al siguiente fragmento.

Puede suceder que en cualquier momento el tema deba cambiar. Si se solicita un cambio de tema, lo que sucede es que se comprueba cuál es el índice correspondiente al corte más cercano, y se almacena esta variable. Más tarde, a la finalización de cada compás, es comprobado si este índice existe, y si coincide con el índice del fragmento actual. De darse esta condición, ejecutamos el cambio de tema, lo que conlleva un ciclo de pausa y reanudación, que estando las variables actualizadas lanzará la siguiente composición del juego.

Adicionalmente, se debe mencionar que por motivos de optimización no son cargadas todas las pistas nada más arrancar la aplicación, ya que esta es una tarea realmente costosa dado el gran número de pistas de las que consta cada tema. Por este motivo, y después de que haya sido cargado el primer tema durante la muestra de la pantalla de carga que aparece al arrancar la aplicación, con cada finalización de compás se comprueba si el siguiente tema está cargado en memoria. Si no lo está, lanza un hilo independiente que recogerá cada fichero y lo llevará a la memoria de la aplicación.

MusicTheme

Esta clase lleva la gestión de la música a un nivel intermedio, conteniendo cada fragmento de los existentes en el tema. Cada instancia de MusicTheme representa un tema distinto, que únicamente requiere informar una variable privada del tipo de enumeración THEMES para indicar a qué tema se corresponde la instancia en particular, y condicionar el resto de su ejecución. Ya que el comportamiento de todos los temas es el mismo, no se ha estimado necesaria la creación de subclases para representar el primer tema, el segundo, y demás. En cambio, han sido implementados los métodos *getSpecifiedSectionEnds*, *getSpecifiedThemeLenght*, o *getSpecifiedThemeKeyword*, que toman como parámetro la variable *theme*, de tipo THEMES, y devuelven el caso que corresponde a cada tema. Por ejemplo, el primer tema dura treinta y siete fragmentos, mientras que el segundo dura solo treinta; el primer tema corta cada sección en los fragmentos cero, nueve, diecisiete, veintiuno y veintinueve, y el segundo corta en cero, siete, trece y veintiuno; y así con los demás casos.

El método de mayor relevancia de esta clase es el de obtainBandAudioFragments. Toma como parámetro la clase contenedora SoundManager, obtiene las referencias al objeto AssetManager y SoundPool, ambos nativos de Android, e itera un número de veces equivalente al número de fragmentos que contiene el tema, formando en tiempo de

ejecución el nombre y ruta del fichero a obtener, cuyo descriptor, objeto de la clase AssetFileDescriptor, es devuelto por cada fragmento e instrumento presente en el videojuego. Evidentemente, el fichero al que se hace referencia debe estar presente en las carpetas de recursos del proyecto, en cuyo caso contrario saltará excepción.

Este método puede ser llamado en cualquier hilo, lo que facilita la carga paralela de la música sin que afecte significativamente al rendimiento de la aplicación.

BandAudioFragment

Finalmente, esta clase lleva la gestión de las pistas al nivel más atómico y directo en el motor diseñado. Está desarrollada pensando en el requisito de que para cada músico siempre debe existir un clip de audio. Todos sus métodos son privados o protegidos, ya que se desea que no puedan ser alterados de manera externa al paquete de gestión de sonidos. Y cuenta con una variable booleana, llamada *locked*, que condiciona la ejecución o no de cada método.

La secuencia por la que pasa cada instancia se describe a continuación. Con la construcción de un BandAudioFragment, tan solo reservamos el espacio que ocuparán sus variables en memoria, inicializando su array de identificadores de clip y sus variables internas. Después, durante la carga del MusicTheme, son entregados los identificadores de los clips generados para cada instrumento. Una vez se han obtenido todos y se bloquea la instancia, el objeto de la clase ya no puede ser modificado, solo leído.

Ahora bien, lo que permite hacer BandAudioFragment con sus pistas es muy sencillo. Para cualquier tipo de reproducción o detención de sus clips, que recordemos representan uno o dos compases de un tema determinado, solicitamos una referencia a SoundPool y le entregamos el identificador del clip correspondiente a la acción que se desea realizar. Así, para reproducir el fragmento de la banda en cualquier momento, se recorre un bucle en que se comprueba si cada músico puede o no tocar en ese momento, gracias a la interfaz InstrumentPlayer, la cual se limita a especificar el método booleano *isReadyToPlay*, y que devuelve verdadero solo si el estado de la partida le permite tocar en ese momento o no. De la misma manera, podemos pausar, detener o cambiar el volumen para cada músico únicamente dando también como argumento el objeto SoundPool.

Vista la implementación de las distintas clases que forman parte del gestor de sonido, y de las características de las pistas de audio que suenan en el juego, queda por ver cómo serán integrados los efectos de audio del juego en el futuro. Esto será expuesto en el punto de futuras implementaciones.

5) Lector de controles

Para el desarrollo de todo lo relativo al control del usuario de la aplicación, partimos de la base de que, al tratarse de un videojuego para plataformas móviles, descartamos al completo todo control de mandos físicos, y contamos únicamente con el caso táctil. Esta restricción podría ser solventado fácilmente en plataformas como PC, dado que disponemos de un ratón con puntero, o en el caso de que quisiéramos lanzar este videojuego en una consola que solo disponga de mando, es posible implementar un puntero virtual que sería desplazado con las interfaces hardware disponibles. En cualquier caso, todo el desarrollo de

esta clase partirá de la concepción de que cada interacción pasa por ajustar y leer coordenadas. De este modo, la clase TouchInputController ha sido diseñada para abstraer sus dos funciones fundamentales.

Controles táctiles

Por una parte, se define la gestión de los controles táctiles. Esto se hace implementando la interfaz OnTouchListener, presente en la clase View, que es un elemento fundamental de las librerías de Android.

Cuando la instancia de la clase TouchInputController es creada, recupera la referencia a la SurfaceView, presente en el fragmento existente de la aplicación, y asigna su propia instancia como *listener*, lo que permite capturar cada control táctil. A partir de este momento, cada vez que el usuario toque la pantalla mientras que la aplicación esté seleccionada, será ejecutado el método *onTouch*, que por defecto nos entregará dos referencias:

- La instancia de View sobre la que se haya pulsado, que nos permitiría interactuar con esta si quisiéramos, y además espera que ejecutemos su método *performClick* en el caso de que no hayamos tratado la pulsación, para poder propagarla a otros elementos de Android.
- Una instancia de tipo MotionEvent, que contiene toda la información que se desea extraer del evento de pulsación. De aquí obtenemos si la acción supone un deslizamiento, pulsación de la pantalla o liberación de otra pulsación previa.

Con esta información podemos deducir fácilmente si el usuario quería pulsar un botón, si quería deslizar una barra o si ha podido confundirse y realmente no quiere realizar ninguna acción.

Recordemos ahora algo mencionado durante la exposición del mecanismo de dibujado. Sabemos que existen literalmente miles de tipos de dispositivos móviles Android en el mercado, y que cada uno cuenta con sus propias características, o lo que nos interesa para este caso, distintas dimensiones de la pantalla, en que ni siquiera es garantizada la relación de aspecto entre ancho y alto. Es por esto por lo que el motor debe calcular, antes de tratar el evento de control, dónde el usuario ha querido pulsar realmente.

En el punto en que se habla sobre la inicialización de la aplicación, se dice que este sistema toma también las dimensiones del dispositivo físico. Esto se hace porque se desea mantener siempre una relación de aspecto de dieciséis novenos. Ahora, independientemente del tamaño de la pantalla, la aplicación será siempre centrada y serán añadidas barras verticales u horizontales según el espacio que quede sobrante. Este espacio debe ser restado de las coordenadas cuando el usuario pulse para realizar una acción. No solo esto, sino que el juego es renderizado siempre y de manera prefijada para todos los dispositivos a la resolución de 1280 por 720 píxeles. Por lo tanto, ante cualquier tamaño de pantalla, escalaremos la coordenada seleccionada como un porcentaje relativo a estas dimensiones.

El resultado de todo este proceso es el siguiente.

- En el caso de que el usuario haya viajado por la pantalla sin soltar el dedo, un deslizamiento, que es consumido en tiempo real por otros hilos. Además, es liberado cuando el usuario suelta la pantalla.
- En el caso de que el usuario haya apretado y soltado la pantalla sin moverse más de diez unidades de píxel, se considera que ha habido un *click*, y es almacenado un punto con las coordenadas dadas para ser tratado por otros hilos.

Ejecución de la orden

Como se ha explicado, el sistema anterior nos da como resultado una única coordenada que se interpreta como la que el usuario ha pulsado para interactuar. Así pues, por otra parte, TouchInputController proporciona un sencillo mecanismo para entregar a la escena adecuada esta coordenada, que la tratará de forma autónoma, y realizará las acciones que convenga. Como recordatorio, esto es faena del hilo de actualización, que a cada ejecución tiene como una de sus tareas la de hacer esto mismo, tomar la coordenada si existe y tratarla.

Pero visto estos puntos, queda por resolver otra cuestión fundamental para el correcto funcionamiento de la interacción con el usuario, y es la incógnita de cómo el motor averigua qué entidad ha sido pulsada de entre todas las que hay en el videojuego disponibles. Esto se explica a continuación.

Sistema de colisiones

Para detectar qué entidad ha sido pulsada contamos con el sistema de colisiones del motor diseñado. Así como ya hemos visto la interfaz Clickable, y el papel que juega en dotar a algunos de los controladores de funcionalidades relacionadas con la interacción, la clase Hitbox es una de las otras piezas fundamentales.

Hitbox, *SpaceBox y SquareInterface*

La clase Hitbox es perteneciente al paquete *util*, de *engine*, y extiende de SpaceBox. Por eso, antes de hablar de Hitbox, repasaremos la función que cumple esta última clase mencionada.

SpaceBox implementa SquareInterface, que es tan solo una abstracción de capacidades propias de SpaceBox. SquareInterface entra en juego también con el caso de Square, que es una clase que tiene como único propósito representar un rectángulo, o cuadrado, en pantalla.

Pero la utilidad de SpaceBox no es solo representar este rectángulo, sino hacerlo en relación con una entidad externa. La diferencia es sutil, pero este sistema ofrece una gran flexibilidad. Mientras que la clase Square indica directamente las coordenadas de sus vértices, SpaceBox las calcula en tiempo de ejecución, en función de qué entidad tenga vinculada. De una forma más precisa, SpaceBox indica los porcentajes de espacio que ocupa una especie de caja en el interior de una figura externa. Es decir, si tenemos que dibujar un personaje, este personaje tendrá una altura y un ancho, pero no todo este espacio es ocupado por el personaje. Este tendrá unas piernas y una cabeza en que el ancho de su imagen no esté ocupado al completo. Aquí entra en juego SpaceBox.

Por lo tanto, si queremos conocer qué coordenadas ocupa la cabeza de un personaje, consultamos su SpaceBox y llamamos a uno de sus métodos para obtener una coordenada o una medida. Estos métodos tomarán los porcentajes indicados y harán las operaciones requeridas para ofrecer el lugar en que se encuentra cada parte del personaje y cuánto espacio ocupan.

Aclarado este punto, hablar de Hitbox resulta mucho más sencillo, pues es un descendiente de la clase SpaceBox, que además obliga a entregar una referencia a Clickable, en lugar de una de tipo Dimensions. Aquí existe un paralelismo, y es que, de la misma forma que Hitbox hereda de SpaceBox, Clickable hereda de Dimensions. Hitbox, pues, es la versión de esta clase sirviéndose de las funciones extendidas con que cuenta Clickable, y SpaceBox se limita a utilizar una instancia de SpaceBox, que no permite por ejemplo hacer *click*. Gracias a la flexibilidad del sistema de herencia de Java, Para crear un objeto Hitbox podemos pasar como referencia al constructor de su clase ascendiente, SpaceBox, una referencia de Clickable, que ya implementa Dimensions.

Como resultado de este sistema, la clase Hitbox contará con la posibilidad de devolver una referencia a un objeto que implemente Clickable, interfaz que a su vez abstrae el procedimiento para gestionar cualquier acción. Hitbox además permite indicar un índice determinado, que para el caso de los menús que utilizan esta clase se corresponderá con un índice numérico relativo a cada una de las opciones existentes, estén disponibles o no para la instancia de ContextMenu.

Búsqueda de colisiones

Volviendo al lector de controles, y al hilo de actualización de estado que consulta constantemente su estado, lo que finalmente nos interesa hacer con el punto que el usuario haya pulsado es acudir a la escena y entregarlo para realizar una búsqueda de entre los elementos que la escena en cuestión registre como propios.

Hay suficientes diferencias entre la serie de escenas de las que dispone el videojuego como para que no sea beneficioso restringir a través de un sistema de abstracción el tratamiento de estas coordenadas. Por este motivo, encontramos casos como el de LoadingScreen, la escena que sirve como pantalla de carga, que implementa este método sencillamente no haciendo nada. En cambio, la clase de GameScreen sí realiza un proceso bien estructurado para encontrar de entre las diversas posibilidades a cuáles otorgamos preferencia, ya que es muy fácil que encontremos el caso en que diferentes objetos se superpongan.

Con independencia del modo de almacenamiento y selección de las entidades de una escena, TouchInputController dota al motor de una función estática, y con la que devuelve un valor booleano, en que dadas las coordenadas del punto y un objeto que implemente la interfaz SquareInterface, determina si el punto se encuentra o no dentro de la entidad.

6) Transición entre escenas

Aunque la gestión de la transición entre escenas y las características de los objetos que implementan la funcionalidad de escena ya puede intuirse en base a la descripción realizada de los componentes que utilizan escenas, es conveniente despejar las posibles dudas que pueda plantear la integración de este sistema en el motor a nivel global.

Todas las escenas del juego y sus referencias se encuentran en la clase GameEngine, que como se ha dicho en anteriores ocasiones, sirve como punto central para la ejecución de todos los sistemas del motor. Después de haber inicializado y almacenado las referencias a estas escenas, la clase GameEngine está lista para lanzar el método *startGame*, para el cual la última tarea es asignar la escena inicial y lanzarla a través de la pantalla de carga.

Cuando el método loadSceneTransition es llamado, la escena pasada como argumento se entrega a la variable estática *nextScene*, de la clase LoadingScreen, y es iniciada una cola de tareas que tiene como finalidad preparar el cambio de escena.

Cuando esta cola de tareas ha finalizado, la última llamada cambia la escena seleccionada y los hilos de gráficos y de estados automáticamente pasan a usar la nueva referencia, que gracias a haberse inicializado previamente, puede mostrarse de forma inmediata.

El resultado es un sistema de transición entre escenas al que adicionalmente podemos indicar una tarea que realizar y una escena a la que acudir cuando las tareas hayan sido resueltas. El resto del motor ya está preparado para mostrar y tratar las pantallas que sean mostradas de manera independiente.

7) Cola de tareas

Al igual que sucede con las escenas, es interesante disponer en cualquier motor de videojuegos de un sistema de scripting, el cual permita encadenar múltiples tareas de forma transparente, que no afecte al rendimiento general del sistema, y que sea transparente; es decir, que pueda ser usado sin cuestionarse cuál ha sido el método de implementación de la herramienta.

El sistema diseñado dispone de los siguientes componentes.

QueueManager

Esta clase pretende servir como el gestor central de las tareas que son programadas para su ejecución. De cara al exterior, su uso es realmente sencillo. Se implementa, como muchas de las clases presentes en el motor, como una clase Singleton, de la que es posible pedir su instancia, y esta será siempre única. Con la instancia podemos realizar algunas llamadas:

- *addAutomation*: Método sobrecargado que acepta tanto un objeto de tipo Automation o una lista de estos, su comportamiento en ambos casos será el mismo, tratando los objetos pasados uno detrás del otro.
- *hasEffectsPending*: Comprueba si en la clase existe, para la instancia Automation que actualmente esté vinculada a la clase, alguna tarea que identificada como de efectos.
- *obtainCurrentEffects*: De existir efectos actualmente dados de alta en la clase, acude al objeto Automation vinculado a QueueManager y los devuelve.

Estos métodos no dejan intuir las capacidades que hay tras este paquete de automatizaciones, pero precisamente en el objeto que exige *addAutomation* es donde se encuentra la virtud de esta funcionalidad.

Automation

La clase Automation ofrece la posibilidad de encapsular diferentes tareas en su interior y las gestiona de manera transparente. Las prepara creando un hilo para cada una y monitorizando su estado, con el objetivo de lanzar la siguiente instancia de Automation que se haya insertado en la cola de la clase QueueManager.

Y aquí es donde está la clave del asunto. Para usar QueueManager es necesario crear un objeto de tipo Automation, el cual puede crearse vacío o añadir sus tareas vinculadas inmediatamente. Cada tarea vinculada con Automation debe implementar la interfaz AutomatedTask. Esta interfaz provee de un método implementado por defecto, llamado run, el cual ejecuta el método startAutomatable, que es de implementación obligatoria por la interfaz, y al finalizar ejecuta una vez el método *endStep*. Este último método hace decrecer un contador en la instancia de Automation. Este contador, al indicar todas las AutomatedTask e inicializarlo, habrá sido almacenado como el número de tareas que se debe realizar.

Tras haber realizado todas las tareas que hayan sido entregadas a la clase Automation, se accede al objeto CyclicBarrier, presente entre sus variables, que es de una clase ofrecida de forma nativa por Android. Este objeto lanza una tarea que sencillamente comprueba si existe una nueva instancia de Automation disponible, actualiza en cualquier caso el estado de la clase QueueManager, y arranca el siguiente grupo de tareas si se encuentra.

A modo de resumen, para hacer uso de la cola de tareas se siguen los siguientes pasos:

- Es creada una instancia de Automation
- Se le entregan tareas, cada una de tipo AutomatedTask
- Se bloquea Automation para preparar su lanzamiento
- Se vinculan todas las instancias de Automation deseadas con gestor de colas

Una vez realizado esto, cada nueva Automation que sea añadida a QueueManager ejecutará el método *setNextAutomation* de su propia clase. Ya que cada instancia de Automation puede contener una referencia a la siguiente instancia que se pida añadir, cuando ya existe un grupo de tareas esperando a ejecutarse, este método se llama a sí mismo con la siguiente instancia de Automation como parámetro, y esto sucede hasta encontrar la última referencia en la cola, cuyo siguiente grupo de tareas se encontrará con valor nulo. La última instancia es colocada al final de la cola, y con cada nuevo arranque de un grupo de tareas, el primer objeto de Automation es eliminado de la cola, y secuencialmente van siendo ejecutados cada uno de estos grupos.

Efectos visuales

El paquete de automatizaciones facilita además la característica de programar efectos visuales como una de sus tareas. Si una clase que implementa AutomatedTask implementa a su vez GraphicEffect, añadirla a una instancia de Automation producirá su tratamiento por separado. Es decir, alterará el comportamiento de algunos métodos de esa clase para dar soporte a los efectos gráficos.

Un ejemplo práctico de la utilidad que suponen estas clases podía ser intuido en el punto sobre mecanismo de dibujado. Aquí, uno de los pasos decía que, si existen efectos gráficos, se aplican en ese momento. De hecho, lo que se está haciendo es acudir a la instancia de

QueueManager y recorrer la lista devuelta por *obtainCurrentEffects* para aplicar cada uno de ellos, entregándole los parámetros Canvas *frameCanvas* y Bitmap *frameBitmap*, tal como especifica la interfaz GraphicEffect con su método *applyEffect*.

En particular, esto permite que la clase Zoom exista y actúe. Ya que Zoom es un GraphicEffect, interfaz que hereda de AutomatedTask, y permite a Zoom exponer públicamente los métodos de ambas interfaces en su clase, cuando desde SurfaceGameView sea llamado el método mencionado anteriormente, la referencia al objeto Canvas y al objeto Bitmap podrán ser leídas y alteradas.

Eventualmente, cuando los objetos listados en la cola de tareas terminan desaparecen automáticamente, y no requieren de una atención posterior.

8) Utilidad de debug

Aunque la ingeniería del software tiene entre sus objetivos minimizar el retrabajo y construir sistemas software de calidad que, gracias a un buen trabajo previo de análisis y diseño, resulten libres de errores, lo cierto es que siempre pueden aparecer incógnitas al enfrentarse a la producción de un software de grandes dimensiones. Y no solo por errores, sino como herramienta para acelerar el proceso de desarrollo, las herramientas de debug están extendidas por la gran mayoría de lenguajes, entornos de desarrollo y tecnologías.

El desarrollo en Android no es tan sencillo como lo podría ser en sistemas de escritorio, en que GNU/Linux o Windows se presentan como entornos de enormes capacidades, gestión de multitarea avanzada y un grado de libertad casi absoluto. En Android solemos estar limitados, por la propia concepción de su arquitectura, a una aplicación corriendo al mismo tiempo, por lo que si nuestra aplicación falla, sería complicado tratar de abrir otras ventanas, introducir comandos de cualquier tipo, o extraer información de otros canales de información que implementa el sistema operativo.

En cualquier caso, se ha considerado conveniente la integración en el interior del propio motor de videojuegos desarrollado un completo sistema de introducción de comandos, así como modos de crear controladores de entidades destinados a mostrarse y permitir interactuar con ellos únicamente cuando el juego indique que se encuentra en modo debug, o de desarrollo.

DebugHelper

Esta es la clase que centraliza las herramientas que han sido desarrolladas para hacer posible una consola de comandos con funcionalidades flexibles y sencillas, más algunas utilidades que facilitan desde cualquier punto del código tener información del estado del motor del videojuego.

Terminal

La clase cuenta con un controlador de tipo CustomDrawableEntity, el cual dibuja el comando que actualmente haya introducido, sobre un fondo de tono oscuro.

Para capturar cualquier el uso de cualquier teclado registrado en Android es tan sencillo como sobrescribir el método *onKeyDown(int keyCode, KeyEvent event)* existente en la clase Activity. Originalmente, este método captura todos los eventos y los propaga por el sistema Android. Al sobrescribir el método original, podemos interferir en esta cadena y descubrir

qué botones se están pulsando. Como no queremos que la cadena de eventos continúe propagándose una vez es capturado, se procesa el evento que llega a la clase SimpleActivity y sencillamente se devuelve el valor booleano true, indicando que el evento ya ha sido consumido. La única excepción es el caso de que la tecla pulsada sea la de bajar o subir volumen, en cuyo caso devolveremos el valor false para que el método de la clase superior propague el evento y podamos manipular el volumen del sistema. Además de valores alfanuméricos, el objeto KeyEvent es capaz de conocer si la tecla pulsada ha sido la de retroceso, espacio, una de las flechas de dirección, o si está activado algún modificador de mayúsculas. En cada caso, se envía a la instancia de DebugHelper la correspondiente orden para manipular la referencia String guardada.

El terminal también dispone de un listado de comandos previamente introducidos, los cuales son navegables hacia atrás y hacia delante, en que se muestran comandos por el orden en han sido introducidos por última vez. Cuando el terminal es construido, el método initializeCommands es ejecutado, y llena un HashMap con pares de clave y valor en que la clave representa el comando que se debe introducir y el valor es una instancia de Runnable con las órdenes a ejecutar.

Cuando la tecla *enter* es pulsada, el comando actual es almacenado en el historial de comandos introducido, y se busca en la lista de posibles comandos esa clave. En caso de que se devuelva un valor diferente de *null*, la instancia de Runnable es ejecutada para cumplir con su propósito.

Además, con cada pulsación de tecla, la clase DebugHelper comprueba si la String formada por los últimos caracteres coincide con algún comando oculto, para ejecutar su tarea. Esto se aplica para el caso en que el modo debug esté desactivado por defecto, pero deseemos activarlo. Será tan sencillo como, mientras se tiene la aplicación abierta, teclear la palabra "debug". De esta forma, tenemos un sistema de códigos secretos que el desarrollador conoce y puede utilizar.

Métodos printMessage

La clase DebugHelper también facilita diversos métodos de carácter estático que pueden imprimir en la consola de logs un mensaje, o almacenarlo como variable de tipo String, que será considerada como el último mensaje introducido. Este mensaje, si el modo debug está activado, será mostrado para poder conocer desde la propia aplicación el progreso que sigue la ejecución del juego.

9) Ajustes de usuario

Android, entre una de sus características, permite el almacenamiento de forma nativa e interna a cada aplicación.

Aunque el método de hacer esto será descrito en el punto dedicado a la implementación en Android, es posible exponer el trabajo que realiza la clase GameSettings conceptualmente. GameSettings contiene distintas variables declaradas en su clase, que representan estados y preferencias que el jugador puede especificar según desee ajustar. Cuando la aplicación arranca, son cargados de la memoria del dispositivo los ajustes ya guardados con antelación, y en caso de no encontrarlos sencillamente los inicializa con un valor especificado por defecto.

Las diferentes opciones que puede gestionar esta clase pueden ser de diversos tipos, habiendo sido incluidos los casos de cadenas de caracteres, de números enteros y de valores booleanos.

En cualquier momento, es posible obtener la referencia a la instancia de GameSettings y consultar estos valores o modificarlos.

Selección de idioma

Uno de los ejemplos en que se aplica este sistema es en la propia selección del idioma de la aplicación. Desde el menú de opciones, presente tanto en la pantalla de pausa como la de inicio, es posible presionar sobre el botón de idioma y la aplicación cambiará entre sus diferentes idiomas disponibles. Ya que el texto es cargado en tiempo real según es impresa cada palabra, el cambio del idioma seleccionado producirá un efecto similar que con el de la transición entre escenas. La llave a la que accedan todos los puntos de la aplicación que quieran consultar el valor de una cadena de caracteres cambiará, y de esta manera, el idioma correcto será mostrado en pantalla instantáneamente.

Selector de volumen y silencio

Puede haber varios motivos por los que deseemos ajustar el volumen de la música, o incluso silenciarla. Puede ser que el usuario salga de la aplicación o desee modificar el valor de manera consciente. En cualquier caso, la clase GameSettings implementa todos los pasos necesarios para lograr esto, tratando las referencias del sistema de Android y modificando los valores que la aplicación está utilizando en la propia ejecución.

Todos los valores que sean modificados serán guardados entre sesiones de juego. Con esta metodología sería sencillo incluso guardar los datos de partidas guardadas si sencillamente es respetado un sistema que guarde coherencia al cargar y almacenar los datos.

10) Gestor de recursos

Para finalizar con la exposición del punto más denso de esta memoria, la arquitectura software del proyecto, cabe mencionar también qué sistema se ha llevado a cabo para organizar las variables iniciadas en memoria. Android es un sistema operativo que es muy sensible al gasto de recursos, partiendo de la base de que corre en dispositivos que muy comúnmente agotan la batería durante su uso y que además no cuenta con tantos recursos como un sistema de sobremesa o incluso un portátil. Por esto, en diferentes ámbitos, ha sido necesario sistematizar la manera en que los recursos son accedidos y utilizados.

Referencias

Una vez realizada la inicialización de los recursos del motor que se describe durante el apartado primero del tercer punto, la arquitectura, repartimos todas las referencias entre varias clases, según la naturaleza que guarden en relación con el papel que cumplen para el proyecto.

En primer lugar, tenemos a SurfaceGameView. Este es el segundo punto de entrada de la aplicación, tras el inicio de la Activity de la aplicación, que a su vez manda al Fragment a ser cargado. SurfaceGameView guarda fundamentalmente los tamaños del dispositivo físico en que se ejecuta el sistema, y para su inicialización, GameEngine no ha sido cargado aún, ya que esto se hace posteriormente, cuando el *listener onViewCreated* del fragmento ha sido

llamado. Por este motivo, contamos con varias limitaciones que dificultan la manipulación de esta clase desde un punto temprano.

Por otro lado, tenemos a GameEngine, la cual contiene las escenas del videojuego. Cada una de las escenas debe gestionar toda la información relativa a su visualización y control, con lo que manejan un buen número de recursos. Para acceder a cada escena pues, bastará con obtener la referencia de GameEngine, entregada y almacenada en GameInstances. GameEngine dispone de métodos para obtener tanto la actual escena como cualquiera de las que se desee, con tan solo indicando el tipo correcto como parámetro.

GameInstances es inicializada por GameEngine en su constructor. GameEngine después entrega su propia referencia a esta clase para poder ser accedida a través del método getInstance de GameInstances. Esta clase dispone de StateManager, GameEntityManager, Concert y SoundManager; diferentes clases que sirven para almacenar referencias de las que interesa acceder desde cualquier lugar de la aplicación.

En último lugar, tenemos la clase GameResources. Esta clase utiliza una referencia a la Activity de la aplicación para ajustar los valores que permitirán obtener recursos dependientes del sistema Android, para el caso de las imágenes, y el objeto Context o las preferencias nativas de la aplicación. GameResources además es el lugar en que son especificados los métodos para representar elementos genéricos como lo son los botones que se utilizan en distintos lugares del proyecto, y etiquetas o líneas dibujadas en los diálogos. La clase GameResources contiene además recursos que serán descritos a continuación.

Instancias flyweight

Este patrón de diseño, bien conocido y extendido en el mundo de las aplicaciones gráficas, y en especial en los videojuegos, se fundamenta en una idea realmente sencilla. Si cuentas con distintos objetos, los cuales hacen referencia a recursos gráficos, sonoros, o en cualquier caso que sean pesados, y que sean muy similares o equivalentes entre sí, la mejor opción es alojarlos una única vez en memoria, y que todas las instancias que deseen recurrir a este recurso en particular lo utilicen de un banco compartido de datos en lugar de reservar nuevos recursos.

El ejemplo de esto se puede encontrar en cinco casos bien conocidos en la aplicación. Estas son las múltiples clases estáticas declaradas en GameResources: ContextMenuFlyweitght, DialogMenuFlyweitght, PauseMenuFlyweitght, LoadingScreenFlyweitght y SliderFlyweitght.

Para todos los casos el procedimiento es el mismo. Durante la primera llamada a cualquiera de estas clases, el método *getInstance* se encarga de construir la referencia adecuada, la cual para todas las imágenes declaradas forma la ruta y el nombre de cada fichero y acude al método *loadBitmap*, el cual funciona de forma muy similar al ejemplo desarrollado durante el punto de carga de música.

Con este punto, cada elemento de la arquitectura del proyecto ha sido expuesto y definido en esta memoria.

4. Implementación en Android

En todos los puntos de la exposición que se ha realizado acerca de la arquitectura software del proyecto se ha seguido el criterio de tratar principalmente todos los detalles que podrían ser abstraídos a un ámbito que no sea el de Android exclusivamente. En cambio, hay fragmentos del código desarrollado que no pueden ser entendidos sin tener en cuenta las peculiaridades de este sistema. En primer lugar, se procederá a realizar una descripción básica de cómo el sistema Android permite desarrollar aplicaciones.

0) Resumen de las características

Toda aplicación Android funciona en base a la dualidad entre los ficheros *layout* (que Google traduce en su documentación pública como Diseños) [26], y los ficheros de clases, sean escritos en Java o en Kotlin (los dos principales lenguajes soportados para el desarrollo en Android).

```
simple_activity.xml

1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:tools="http://schemas.android.com/tools"
4      android:id="@+id/container"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      tools:context=".android.SimpleActivity"
8      tools:ignore="MergeRootFrame" />
```

Los *layout* definen la estructura gráfica de la capa de presentación. En una aplicación corriente de este sistema operativo, algo como un calendario, un cliente de correo electrónico o un reproductor de música, es habitual utilizar elementos nativos para poblar la interfaz, y contar con funcionalidades predefinidas que faciliten la implementación. En el caso de la aplicación desarrollada, el motor de videojuegos no necesita de ninguno de estos elementos, sino que, para mantener un estilo consistente y tener control absoluto sobre la información que corre el sistema, todos los elementos han sido diseñados en su totalidad. Cada menú, cada diálogo, cada control deslizable o efecto gráfico ha sido diseñado de manera independiente al sistema Android, por lo que no es en absoluto descabellado plantear en el futuro un desarrollo paralelo que defina interfaces comunes y abstraiga las restricciones de cada sistema para permitir reutilizar todo el código en de una manera sostenible. Pero lo que actualmente nos ocupa es exponer el trabajo realizado.

1) Dibujado

```
SurfaceGameView.java

143         // Obtiene la referencia al canvas nativo y lo sincroniza
144         Canvas screen = getHolder().lockCanvas();
145         if (screen == null) {
```

Para dibujar sobre una aplicación Android y tratar los píxeles de una manera eficiente, una de las maneras más simples y efectivas es colocar un elemento de tipo SurfaceView en el *layout* correspondiente al Fragment de la aplicación. SurfaceView dispone de entre otros atributos de SurfaceHolder, con el cual se puede llamar al método *lockCanvas*. Esto devuelve una instancia de la clase Canvas, la cual permite realizar todo tipo de operaciones que alterarán directamente los píxeles de la superficie [27].

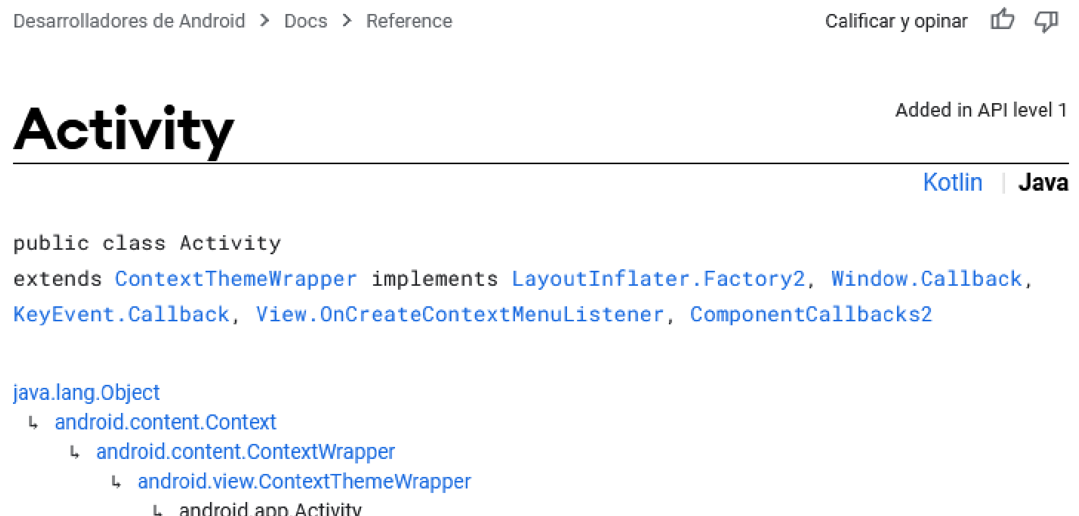
Cuando se desea terminar la edición de este Canvas, basta con recuperar de nuevo la referencia al SurfaceHolder y solicitar el proceso contrario, *unlockCanvasAndPost*, que sobrescribirá los contenidos mostrados en pantalla, haciendo posible el refresco.

```
SurfaceGameView.java

165        // Plasma el frame obtenido tras aplicar el dibujado de los elementos
166        getHolder().unlockCanvasAndPost(screen);
167    }
```

2) Actividades

Como se ha dicho antes, todas las aplicaciones requieren definir una Activity, la cual es el punto de entrada al sistema. En palabras de la documentación oficial "La clase Activity es un componente clave de una app para Android, y la forma en que se inician y se crean las actividades es una parte fundamental del modelo de aplicación de la plataforma. A diferencia de los paradigmas de programación en los que las apps se inician con un método main(), el sistema Android inicia el código en una instancia de Activity invocando métodos de devolución de llamada específicos que corresponden a etapas específicas de su ciclo de vida" [28].



Aquí hay dos conceptos clave a tratar; el manifiesto, un fichero presente en el proyecto Android que define detalles fundamentales para conocer cuáles son los puntos de entrada de la aplicación, y que permite definir varias propiedades que afectarán a la ejecución del

sistema; y el ciclo de vida de la aplicación, que es la secuencia de estados que recorre una aplicación a lo largo de su ejecución, y en que cada transición de un estado a otro cuenta con métodos que es posible sobrescribir para alterar su funcionamiento.

AndroidManifest

Este fichero es escrito en formato xml, y permite modificar opciones que alteran la configuración por defecto de la aplicación [28].

Algunas aplicaciones permiten acceder a través de distintas actividades. Esto es algo realmente útil cuando quieres permitir enviar un correo en lugar de ver la bandeja de entrada, comprobar una nota desde un acceso directo, o reenviar un mensaje sin pasar por la lista de chats.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.psychocactusproject">
    <!--supports-screens android:anyDensity="false"/-->
    <application
        android:allowBackup="true"
        android:icon="@mipmap/icon_barry"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/icon_barry_round"
        android:theme="@style/Theme.AppCompat">
        <activity
            android:name="com.psychocactusproject.android.SimpleActivity"
            android:label="@string/title_activity_game"
            android:theme="@style/Theme.AppCompat.NoActionBar"
            android:screenOrientation="sensorLandscape"
            android:persistent="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Pero en el caso del videojuego que se está desarrollando, tan solo tiene sentido definir una Activity, tal como se ve en el ejemplo. El modo de declarar cada una es introduciendo un objeto con el nombre “activity” dentro de “application”, que es obligatorio declarar dentro de “manifest”. Esta Las opciones que se destacan se listan a continuación:

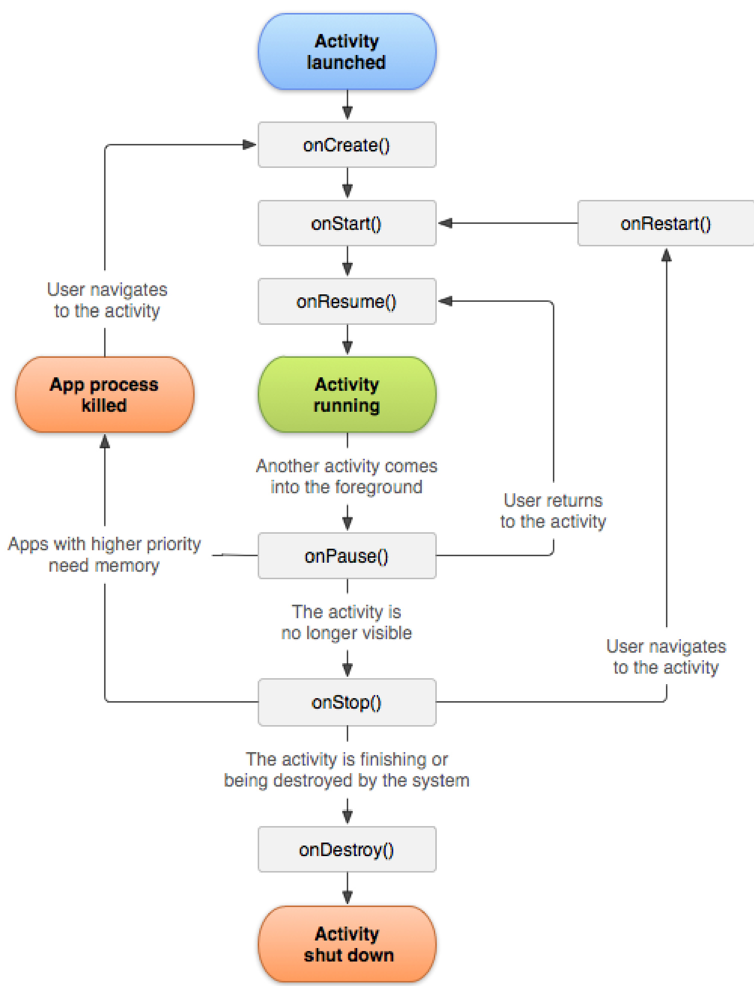
- “android:icon” y “android:roundIcon”: Variables que identifican la imagen que servirá como icono de la aplicación. En el caso del videojuego expuesto en este trabajo, este será Barry, el dueño del bar donde se da el concierto, y personaje clave en el transcurso de la partida. Según el estilo del sistema operativo seleccionado por el usuario, será mostrado un icono u otro.

- “android:label”: Determina el nombre que mostrará la aplicación como título.

- "android:name": Se sitúa dentro de cada actividad declarada, e indica la clase a la que se accede para crear la actividad por primera vez. Esta clase debe heredar de Activity, o como en el caso de la aplicación que se ha desarrollado, AppCompatActivity, que es una versión equivalente que adapta funcionalidades del legado de Android a los nuevos sistemas. Esto se hace para desarrollar para el mayor número de dispositivos disponibles, a costa de perder algunas funcionalidades modernas.
- "android:screenOrientation": Cuando es ajustada a "sensorLandscape", permite establecer como modo predeterminado el apaisado, permitiendo al móvil girar 180 grados sin destruir el contexto de la aplicación, como sí pasaría si se permite girar también 90 grados.
- "android:persistent": Ajuste fundamental para este fichero manifest. Obliga a la aplicación a correr constantemente en segundo plano, sin permitir la destrucción de su contexto a pesar de no haber entrado en un tiempo prolongado. Esto permite conservar partidas dejadas a mitad, pero tiene como contratiempo un alto consumo de recursos aún con el teléfono bloqueado.
- Los atributos intent: Especificando filtros de intent, podemos indicarle al sistema operativo de Android qué tipo de Activity estamos declarando. Con "action.MAIN" y "category.LAUNCHER" estamos especificando a cualquier sistema Android que utilice el usuario que la aplicación Psycho Cactus tiene una actividad principal que no recoge datos, tan solo lanza.

Ciclo de vida de la aplicación

Aunque el ciclo de vida que sigue una aplicación es realmente flexible y permite adaptar la



aplicación desarrollada a cualquier uso del que se quiera dotar, el motor diseñado no explota estas capacidades, y se limita a realizar una implementación modesta y suficiente.

Las transiciones clave que Android permite manipular se alteran a través de los métodos onCreate, onStart, onResume, onPause, onStop, onRestart, y onDestroy [29]. De todas las transiciones mencionadas, la única que interesa sobrescribir en nuestro sistema es la primera, onCreate, ya que permite introducir las funcionalidades que se han desarrollado en el primer paso de creación de la aplicación. Desde aquí llamamos al Fragment para continuar con el arranque del motor del videojuego.

SimpleActivity.java

```
42        // La actividad y su fragmento arrancan, inicializando el resto de componentes
43        if (savedInstanceState == null) {
44            getSupportFragmentManager().beginTransaction()
45                    .replace(R.id.container, SimpleFragment.fragmentInstance())
46                    .commitNow();
47        }
48    }
```

3) Fragmentos

Los Fragment en Android son también un elemento fundamental de Android, fueron introducidos en Android 3.0, con la API 11 [30]. Permiten un control más dinámico sobre la aplicación, y ofrecen un nivel independiente de manejo del ciclo de vida y manipulación de la ventana. Los elementos Fragment están diseñados para ser altamente reutilizables, y para hacer del desarrollo de una Activity un proceso más análogo al del software orientado a objetos.

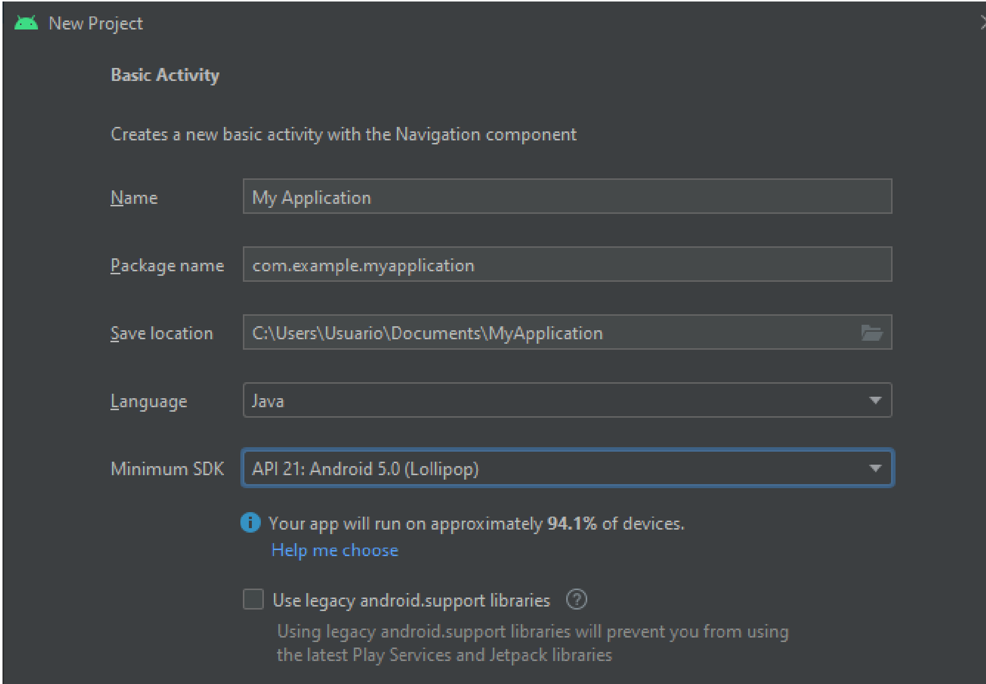
SimpleFragment.java

```
40    @Override
41    public void onViewCreated(View view, Bundle savedInstanceState) {
42        super.onViewCreated(view, savedInstanceState);
43        // Arranca el motor de juego a través de un listener cuando la vista ya haya sido creada
44        ViewTreeObserver observer = view.getViewTreeObserver();
45        final View createdView = view;
46        observer.addOnGlobalLayoutListener(new ViewTreeObserver.OnGlobalLayoutListener() {
47            @Override
48            public void onGlobalLayout() {
49                // Tan pronto como se acceda al código del listener, se borra la escucha
50                ViewTreeObserver currentObserver = createdView.getViewTreeObserver();
51                if (currentObserver.isAlive()) {
52                    currentObserver.removeOnGlobalLayoutListener(this);
53                }
54                // Es obtenida la instancia de activity, y su conversión al tipo del proyecto
55                SimpleActivity simpleActivity = (SimpleActivity) getActivity();
56                // Controlador principal de la impresión de los gráficos
57                SurfaceGameView surfaceGameView = getView().findViewById(R.id.simpleGameView);
58                // El motor es creado con la actividad y la vista
59                gameEngine = new GameEngine(simpleActivity, surfaceGameView);
60                // El gestor de controles es vinculado al motor
61                gameEngine.setInputController(new TouchInputController(gameEngine, getView()));
62                // Arranca el juego
63                gameEngine.startGame();
64            }
65        });
66    }
```

El uso que se va a dar de los Fragment por parte del motor no va a ser avanzado, pero se va a tomar ventaja de su ciclo de vida para disponer de parámetros muy útiles en la inicialización del motor desarrollado. Uno de los métodos que ofrece de forma adicional a Activity la clase Fragment es precisamente onViewCreated, que al ser sobrescrito por la implementación que hace el motor de Fragment, otorga acceso a la instancia View creada, lo que permite disponer de los parámetros del dispositivo en que corre la aplicación de inmediatamente después de ser lanzada.

4) Nivel de API

Una de las primeras decisiones que se debe tomar como desarrollador al iniciar cualquier proyecto que consista en realizar una aplicación Android, debe ser fijar una versión máxima y mínima a la que aspiramos dar soporte.

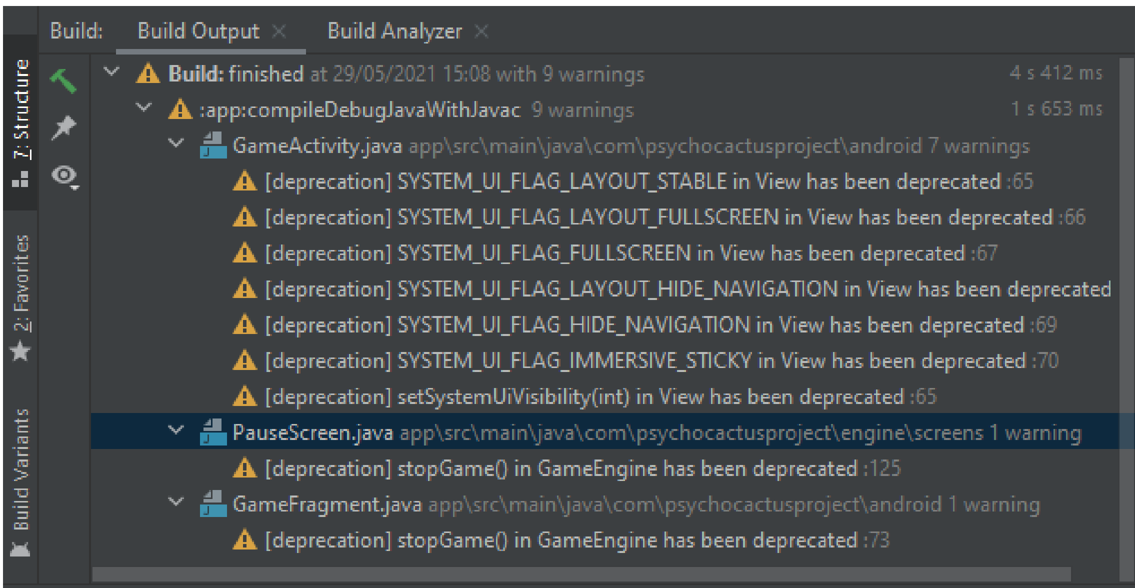


Una de las consecuencias más directas a la hora de escoger una versión mínima y una versión "objetivo" (targetSdkVersion) son dos.

Para el caso de la versión mínima, cualquier función que haya sido introducida en Android posterior al lanzamiento de la API 21 no podrá ser integrada en la aplicación. Y esta no es una cuestión trivial, pues en el caso que nos ocupa, con Android 5.0 es introducido Material Design, que redefine cómo la clase Activity o Fragment deben ser implementadas, y es el motivo por el que SimpleActivity, de la aplicación desarrollada, extiende de AppCompatActivity y no de sencillamente Activity.

```
build.gradle (:app)
1   apply plugin: 'com.android.application'
2
3   android {
4       compileSdkVersion 30
5
6       defaultConfig {
7           applicationId "com.psychocactusproject"
8           minSdkVersion 21
9           targetSdkVersion 30
10          versionCode 1
11          versionName "1.0"
12
```

En cambio, escoger una u otra targetSdkVersion implica la imposibilidad de hacer uso de algunas funciones que en versiones posteriores han sido deprecadas. Esto sucedió con la aplicación al subir de la versión de API 29 a la API 30, cambiando el modo en que debía ser tratada la redimensión y modo de visualización de la ventana vinculada a la actividad.



Gracias al control de versiones Git es posible observar cuáles fueron los cambios necesarios para solventar este problema.

```
65      -            View decorView = getWindow().getDecorView();
66      -            decorView.setSystemUiVisibility(View.SYSTEM_UI_FLAG_LAYOUT_STABLE
67      -                    | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
68      -                    | View.SYSTEM_UI_FLAG_FULLSCREEN
69      -                    | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
70      -                    | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
71      -                    | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY);
72      -        }*/
73      -
74      -        if (hasFocus) {
75      -            WindowInsetsControllerCompat insetsControllerCompat = new WindowInsetsControllerCompat
        (this.getWindow(), this.getWindow().getDecorView());
76      -            insetsControllerCompat.hide(WindowInsetsCompat.Type.statusBars());
77      -            insetsControllerCompat.hide(WindowInsetsCompat.Type.navigationBars());
78      -            insetsControllerCompat.setSystemBarsBehavior(BEHAVIOR_SHOW_TRANSIENT_BARS_BY_SWIPE);
79      -            /* if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
80      -                this.getWindow().getDecorView().onApplyWindowInsets(this.getWindow().getDecorView
        ().getRootWindowInsets());
81      -            } */
    73  +            this.hideAndroidWindowElements();
82  74           }
83  75       }
```

Y aquí se observa el resultado final, que en lugar de utilizar las viejas FLAGS pertenecientes a View, permite manipular la ventana perteneciente al Fragment en cualquier momento. Algo

especialmente interesante conociendo que el usuario puede entrar y salir de la aplicación cuando lo desee.

SimpleActivity.java

```java
public void hideAndroidWindowElements() {
    WindowInsetsControllerCompat insetsControllerCompat = new WindowInsetsControllerCompat(
            this.getWindow(), this.getWindow().getDecorView());
    insetsControllerCompat.hide(WindowInsetsCompat.Type.statusBars());
    insetsControllerCompat.hide(WindowInsetsCompat.Type.navigationBars());
    insetsControllerCompat.setSystemBarsBehavior(BEHAVIOR_SHOW_TRANSIENT_BARS_BY_SWIPE);
}
```

5) Carga de recursos

Este es otro ejemplo de un punto que ha cambiado ligeramente de implementación debido a problemas derivados del sistema Android. Tal como se explica en la sección en que se explica la carga de clips de audio para MusicTheme, y antes también cuando se habla del papel que cumple AnimationResources en los controladores de animaciones de personajes, para cargar todo tipo de recursos de audio o imagen, es posible situarlos en la carpeta del proyecto llamada “assets” y recurrir a objetos FileDescriptor, los cuales son capaces de, en base a una ruta, devolver un identificador con el que copiar a memoria el contenido de estos recursos, para poder ser utilizados.

MusicTheme.java

```java
trackName += " - " + singleInstrument.name() + ".mp3";
// Es cargado de memoria el track según el identificador
try {
    AssetFileDescriptor descriptor = assets.openFd("music/" + trackName);
    int loadedTrackId = soundPool.load(descriptor, themeLenght - i);
    long fileByteLenght = descriptor.getLength();
    audioFragment.setAudio(singleInstrument, loadedTrackId, fileByteLenght);
} catch (IOException e) {
    throw new IllegalStateException("La pista con identificador '" + trackName
            + "' " + "no ha podido ser cargada. Para iniciar el juego " +
            "necesitamos todas las pistas.");
}
```

También este es un gran ejemplo de cómo en ocasiones tener mayor control sobre el sistema operativo es preferible a hacer uso de la aproximación por defecto o más sencilla posible.

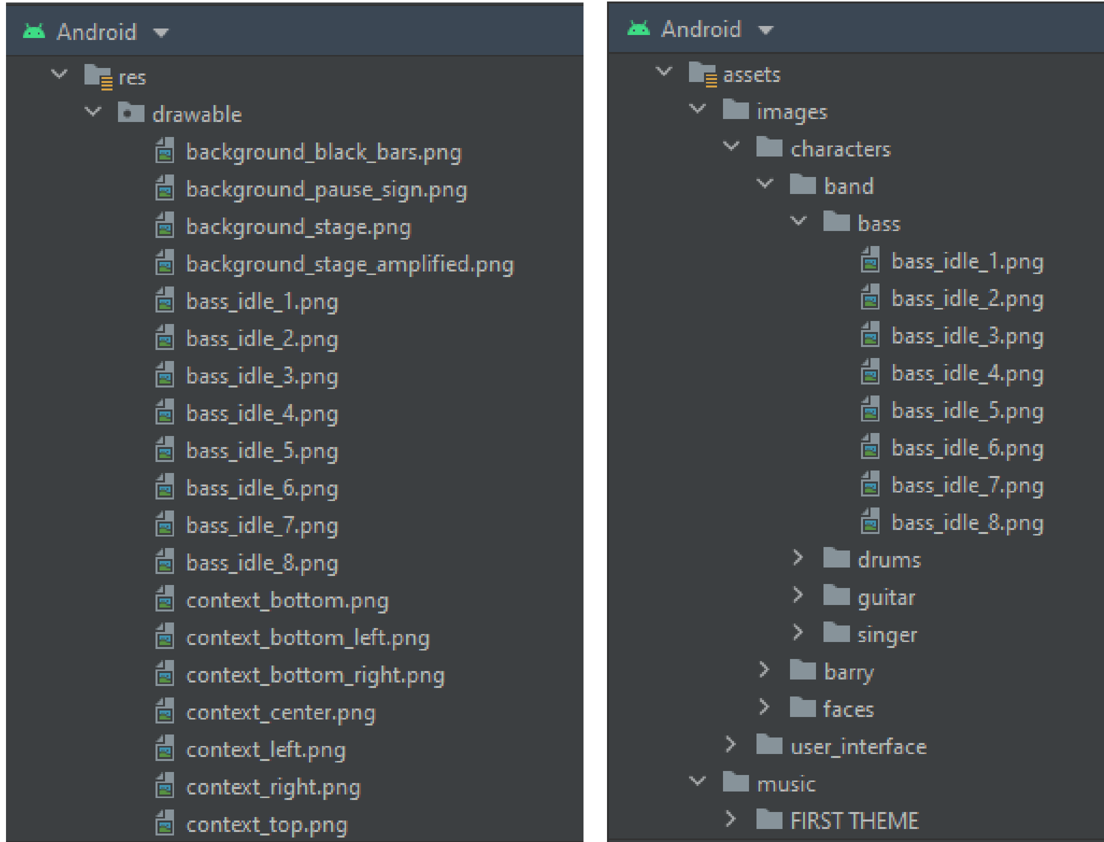
Anteriormente, para cargar las imágenes, en lugar de seguir este procedimiento, recurríamos a la clase R de Android. Esta es una clase que contiene identificadores dinámicos de cada nuevo recurso que sea introducido en el sistema, sean imágenes, layouts, o demás elementos.

Bass.java

```java
@Override
protected AnimationResources obtainAnimationResources() {
    String characterName = "Ronaldo";
    HashMap<String, int[]> animations = new HashMap<>();
    // Imágenes de animación de bajista por defecto
    int[] idleAnimation = new int[8];
    idleAnimation[0] = R.drawable.bass_idle_1;
    idleAnimation[1] = R.drawable.bass_idle_2;
    idleAnimation[2] = R.drawable.bass_idle_3;
    idleAnimation[3] = R.drawable.bass_idle_4;
    idleAnimation[4] = R.drawable.bass_idle_5;
    idleAnimation[5] = R.drawable.bass_idle_6;
    idleAnimation[6] = R.drawable.bass_idle_7;
    idleAnimation[7] = R.drawable.bass_idle_8;
    animations.put("Idle", idleAnimation);
```

Esto tiene la insostenible consecuencia de obligar a concentrar todos los recursos en la misma carpeta. Algo que

cuando existen cientos de animaciones en el juego, y cada animación está compuesta por aproximadamente ocho imágenes distintas, encontrar una imagen, borrarla, o cambiarle de nombre, supone dificultad por sí misma. Mostrado a continuación, el antes y el después:



6) Javadocs

Aunque Android prescinde de muchas de las características de Java y las sustituye por las suyas propias, como es el caso del sistema de compilación y la máquina virtual DALVIK, que Google sustituyó por ART [25].

Pero este no es el caso de javadoc, una simple pero poderosa herramienta capaz de interpretar los comentarios escritos en código Java y transformarlos de manera completamente automática en una serie de páginas HTML por las que navegar como si de la referencia oficial de Oracle se tratase.

La mejor parte es que Android Studio viene con una herramienta incorporada para lanzar la herramienta javadoc sin pasar por la tediosa interfaz de comandos.

https://docs.oracle.com/javase/9/javadoc/javadoc-command.htm#JSJAV-GUID-EAAAE17F-E540

Windows: For example, the following command generates the documentation for the package cc directory:

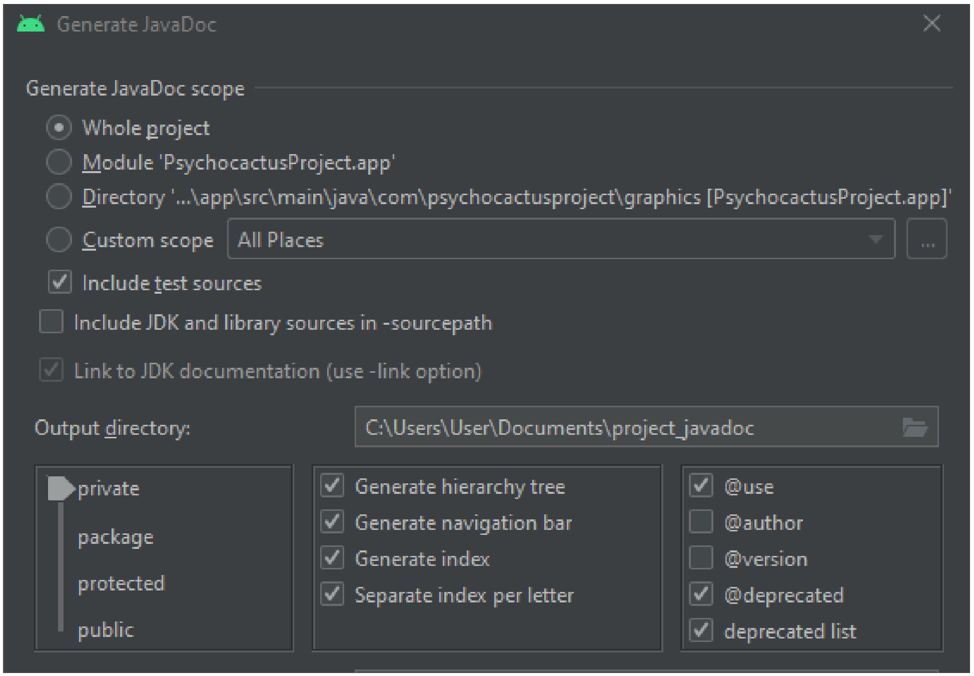
```
javadoc -d \user\doc\ com.mypackage
```

`-docencoding` *name*

Specifies the encoding of the generated HTML files. The name should be a preferred MIME name

If you omit the `-docencoding` option but use the `-encoding` option, then the encoding of the gen

Aquí, un ejemplo de lo sencillo que es ejecutar el comando para generar un conjunto de páginas que permitan visualizar gráficamente la estructura de la arquitectura desarrollada.



Y finalmente, un primer vistazo a la página generada.

5. Aspectos ajenos a la tecnología

El proyecto Psycho Cactus, además de ser un complejo proyecto software, es también un videojuego, con lo que debe ser diseñado como tal. Esto puede parecer trivial, pero añade una capa adicional de esfuerzo. Para expresar con claridad la idea que hay detrás de tal afirmación, el autor de '*The Art of Game Design*', Jesse Schell, ilustra de una manera excelente, y algo irónica, la respuesta a la pregunta: "¿Qué habilidades necesita tener un diseñador de videojuegos [24]?" La respuesta es algo extensa, y no es más que una enumeración alfabética de algunas de las habilidades que dice él que son las más importantes. Animación, antropología, arquitectura, cinematografía, comunicación, economía, ingeniería... y la lista sigue.

En definitiva, no basta con que el producto rinda bien y cumpla con los requisitos, si falla en algo como no ser suficientemente divertido, al usuario no le servirá de nada. Ya que el campo de estudio de la Ingeniería Informática no profundiza en asuntos de carácter humanístico en el sentido que merece ser tratado este punto, no se hará más que un breve repaso a cada uno de los diferentes ámbitos que más influyen en la realización de este tipo de obra.

Cada uno de los puntos corresponde, además, a un documento que fue redactado y enviado a los colaboradores que echaron una mano con este proyecto. Por su relación directa con el proyecto, estos puntos servirán como resumen de los citados documentos.

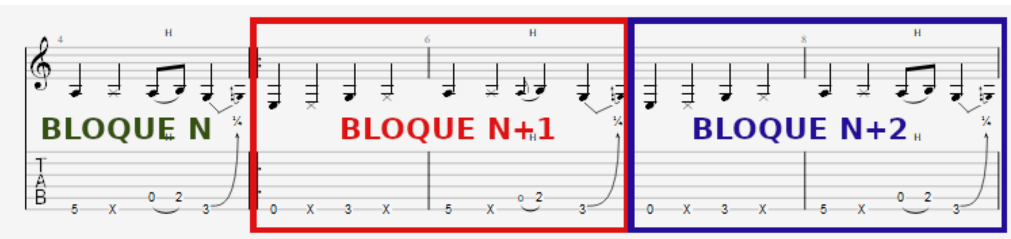
0) Música

Este apartado, que de por sí es vital en cualquier videojuego que se precie, en este caso es de extrema importancia.

Citando el propio planteamiento del documento realizado para reflejar los requisitos de la música compuesta para el videojuego: "Por encontrarse el componente musical en un primer plano, es de vital importancia que animaciones y sus correspondientes sonidos permanezcan permanentemente sincronizados."

Inmediatamente después, se dice lo siguiente: "Durante el transcurso de la partida, la propia banda sonora se irá adaptando al progreso del estado del juego, y de las decisiones tomadas por el jugador. De esta manera, si un personaje queda inhabilitado temporalmente, su pista de audio dejará de sonar o lo hará de manera errática. De igual forma, si este personaje cobra relevancia en determinado momento, porque se ha seleccionado para ejecutar una habilidad o ha entrado en un estado alterado, su pista de audio sonará junto a un pequeño arreglo que deberá siempre ser acorde con la música sonando en ese determinado momento y coherente con la escena".

La descripción técnica que se hizo de la música que debía ser compuesta no entraba tanto en detalle como su punto dedicado en esta memoria, que ha expuesto estos requisitos desde una perspectiva más cercana a la ingeniería. En cambio, este documento explica estos conceptos de una manera más gráfica.



Además de cuestiones técnicas, para que este videojuego fuese coherente con lo que pretendía representar, fue expresada una descripción de las sensaciones que debían trasmitir los distintos temas del videojuego.

Tema	Versión	Detalles de la versión
Partida 1	A	Primer tema que suena cuando arranca la partida, al principio del turno uno.
	B	Tercer tema, variación del tema P1A, entra al final del turno diez.
Partida 2	A	Segundo tema, arranca al acabar el turno cinco.
	B	Cuarto y último tema del bloque jugable, suena desde final del turno quince hasta que la partida acaba. Es una variación del tema P2A.
Menú	Principal	Es el tema que representa la totalidad del videojuego, suena en todo momento que nos encontremos en el menú principal.
	Victoria	Breve fragmento musical, basado en el tema principal, suena tras ganar la partida.
	Derrota	Breve fragmento musical, basado en el tema principal, suena tras perder la partida.

Para la finalización de este Trabajo de Fin de Grado, no fue posible lograr componer más que los dos primeros temas únicamente, los mostrados como Partida 1 – A y Partida 2 – B, que el compositor bautizó, respectivamente, como Oriental Revenge y Moshpit Adventure.

También los efectos de sonido han quedado descartados para la primera versión del proyecto, pero ya están contemplados y serán integrados en el futuro. Un fragmento de este listado se muestra a continuación.

Personaje	Evento	Descripción del efecto
Voz	Piña Colada	Agita el vaso y deja salir una risa suave.
	Gutural	Desata su rabia y lanza un grito gutural.
	Pogo	Suelta una carcajada malévola.
	Solo	Produce un coro angelical a cuatro voces.
Guitarra	Fumar	Un sonido de aspiración y otro de exhalación.
	Escupir	Carga en la boca y lanza un escupitajo.
	Romper guitarra	Destroza la guitarra contra el suelo.

Así como nuevos temas serán compuestos para dar por cerrado el primer nivel del juego, con la eventual adición de nuevas pantallas, también será necesario volver a realizar este proceso.

1) Arte

En cuanto al arte visual, sucede un caso muy similar que con la música. De hecho, igual que a cada personaje le corresponde su conjunto de clips de sonido según sus acciones, cada acción requiere de su traducción como una de las animaciones presentes en el videojuego.

Las claves al hablar de cómo el arte y animaciones son condicionados por los controladores de personaje diseñados ya ha sido desarrollado extensamente durante a lo largo de la memoria, así que no nos detendremos por más tiempo a discutir este punto.

Al igual que con el caso de la música, las únicas animaciones que han llegado a ser realizadas son las mínimas como para probar que el código implementado es correcto. Para futuras fases del proyecto será barajada la posibilidad de reunir una determinada cantidad de fondos como para profesionalizar las actividades que por ahora se han realizado como muestra.

Algunos ejemplo de lo que se ha logrado realizar se muestran tanto a continuación como en la sección de promoción de este mismo punto de la memoria.

2) Jugabilidad

Este asunto no ha sido tratado más que superficialmente durante la redacción de esta memoria, ya que muy pocas ideas extraídas de este sistema lógico influyen significativamente en la estructura del proyecto o el resultado del sistema software.

De hecho, a nivel de código, resolver el almacenamiento y manipulación de los estados lógicos de los personajes y el concierto ha sido tan sencillo como crear una clase para representar el estado del concierto, del que se obtiene la progresión de los turnos y valores globales; y otra clase para representar los estados particulares de algunos elementos del juego, como lo son los amplificadores, el fuego en el escenario o las grietas en la pared. Luego, cada personaje obtiene sus parámetros particulares de su propia clase.

Ya hemos visto a nivel técnico que cada personaje dispone de sus propias acciones, y que cada una desencadena inesperadas consecuencias en la partida. Un ejemplo de esto podemos encontrarlo en la tabla que resume las consecuencias de ejecutar la acción de *Solo* con cada músico.

Efectos de Acciones de Solo	
Músico	**Efecto**
Voz	Canaliza su irresistible voz para revivir a un compañero, que no podrá renunciar a escuchar su delicado canto angelical solo por el pequeño detalle de estar muerto.
Guitarra	Revitaliza a sus compañeros con su solo magistral y devuelve todos los estados de cada músico a su estado inicial, excepto por el propio Guitarra, que seguirá triste porque nadie le quiere en realidad.
Bajo	Obliga a toda persona que se encuentre en el local a dejar lo que esté haciendo y bailar tremendo cumbión, resolviendo así cualquier conflicto existente.
Batería	Entra en estado zen, tocando a compases matemáticamente improbables, lo que asombra el público y lleva la diversión al valor máximo, que se mantiene inalterable por dos turnos.

En este documento, a grandes rasgos, son definidas las normas que sigue la partida cuando se está jugando, desde una perspectiva nada relacionada con la tecnología que hace esto posible.

Aquí, por ejemplo, son detalladas las mecánicas fundamentales del videojuego.

del público cae a cero. Hay tres factores principales que determinarán el desarrollo de la partida.

- Diversión: Mide el grado de disfrute del público. Oscila entre los cero y diez puntos. Interesa mantenerlo en un termino medio, ya que llevarlo a los extremos implicará consecuencias negativas.
- Fatiga: Es un parámetro propio de cada músico, e indica el estrés acumulado. Puede ir de cero a tres puntos, aunque conviene mantenerlo cuan más bajo posible.
- Furia: Al igual que la fatiga, cada músico tiene su propia rabia acumulada. También va de cero a tres puntos, y conviene mantenerlo cuan más bajo posible.

Estos valores se relacionen en un mecanismo similar al de piedra, papel o tijera. Existen cinco acciones

Y más adelante se habla de personajes que aún no han sido implementados en esta versión del producto.

> El agente al que nos tendremos que enfrentar en caso de que vengan será puramente aleatorio, y las particularidades de cada agente se explican ahora.
>
> - Policía Violento: Se puede quitar de encima si Barry se enfrenta a él, lo que siempre acabará con un duro golpe a Barry, que será inutilizado por cinco turnos pero conseguirá que el policía se canse y se vaya.
> - Policía Corrupto: Se llevará algo del concierto, sean unos altavoces, un instrumento, o la ropa de algún músico (dejándole en ropa interior). De esta manera la diversión caerá en un punto cada turno hasta que Barry reponga ese objeto y todo vuelva a la normalidad. Esto le costará a Barry estar fuera por tres turnos.
> - Policía Fiestero: Detiene a un músico aleatorio a menos que le dejemos ser el teclista de la banda. El mal ambiente generado por su presencia hará que la diversión se reduzca en uno cada turno hasta que la policía complete su ronda de patrulla, y esta vuelta entrará para llevárselo, pero no para interactuar.
>
> Para traer de vuelta a un miembro encarcelado podremos enviar a cualquiera de nuestros músicos, que

Conocidos los requisitos técnicos de la arquitectura del videojuego: por turnos, en dos dimensiones, musical; esta redacción tan solo condiciona la lógica implementada, con lo que no guarda mayor interés para esta memoria.

3) Promoción y monetización

El documento relativo al videojuego entendido como proyecto de emprendimiento, a decir verdad, por ahora no es más que un ejercicio de reflexión acerca de cuál podría ser el modo de rentabilización económica del producto desarrollado.

El desarrollo de este videojuego está algo lejos de ser cerrado. Habiendo realizado suficiente esfuerzos como para obtener una versión que estudiar y de la que discutir su viabilidad económica, los puntos tratados en este fichero arrojan algo de luz sobre asuntos que verdaderamente serán vitales en el momento en que se baraje su eventual lanzamiento comercial.

Con esto en mente, los puntos que se han considerado tratar para este informe responden a un esquema clásico de un *briefing* empresarial. Se ha hablado de la aspiración comercial, el público objetivo, piezas de una prevista campaña de marketing, el tono que deberá seguir la comunicación hacia un nivel externo, el presupuesto que se manejaría, y algunas fotos para aportar una idea de la forma que tiene el proyecto en la actualidad.

6. Futuras versiones y metas

Como ya se comentaba durante la introducción de esta memoria, al plantear los objetivos de este proyecto se asume que no es posible realizar un videojuego de talla comercial con las limitaciones de presupuesto y tiempo de las que se parte. El alcance de esta versión es limitado, pero todas las futuras tareas aquí mencionadas han sido consideradas en algún momento desde la concepción del videojuego.

Por este motivo, es vital hacer una breve mención a los planes que existen más allá de la entrega que se defenderá como Trabajo de Fin de Grado. A continuación, serán recorridos diferentes asuntos que, o bien no han sido finalizados, o bien son contemplados como futuros objetivos que debería marcarse el proyecto para ser considerado un producto comercial.

0) Optimizaciones

Un objetivo fundamental que se ha fijado como requisito durante todo el desarrollo ha sido el buen rendimiento del sistema. Realizar una arquitectura software con el único apoyo de librerías nativas permite una gran eficiencia siempre que no se abuse de funciones que no han sido optimizadas para usos no indicados. Salir de estos umbrales tiene como consecuencia exponerse al riesgo de poner en peligro la estabilidad del sistema; y es que la única manera de escapar de este inconveniente es realizar una implementación más cercana a hardware que la realizada actualmente.

Sustitución de SoundPool por implementación OpenSL

Al desarrollar en profundidad el sistema que se ha construido para la integración de música en el videojuego, se ha descrito el uso de la clase SoundPool como elección para llevar la especificación realizada a la práctica. SoundPool es catalogada por la referencia oficial de Android como uno de los tres clientes que permiten el "Diseño Para Latencia Reducida" para el sonido [31].

Design For Reduced Latency

The Android 4.1 release introduced internal framework changes for a lower latency audio output path. There were minimal public client API or HAL API changes. This document describes the initial design, which has continued to evolve over time. Having a good understanding of this design should help device OEM and SoC vendors implement the design correctly on their particular devices and chipsets. This article is not intended for application developers.

Track creation

The client can optionally set bit `AUDIO_OUTPUT_FLAG_FAST` in the `audio_output_flags_t` parameter of AudioTrack C++ constructor or `AudioTrack::set()`. Currently the only clients that do so are:

- Android native audio based on OpenSL ES or AAudio
- android.media.SoundPool
- android.media.ToneGenerator

Sin embargo, esta implementación está lejos de ser perfecta. Por desgracia, no todos los dispositivos permiten a través de esta librería la reproducción de audio en latencia reducida, lo que tiene como consecuencia que algunas de las pistas de audio para las que se solicita su reproducción son lanzadas con un molesto retardo, o no son reproducidas incluso. Es posible observar esto leyendo con atención algunos de los mensajes que lanza esta clase cuando es utilizada en algunos dispositivos.

```
Run: app
I/OMXClient: Using client-side OMX mux.
W/AudioTrack: AUDIO_OUTPUT_FLAG_FAST denied by client; transfer 4, track 48000 Hz, output 44100 Hz
W/AudioTrack: AUDIO_OUTPUT_FLAG_FAST denied by client; transfer 4, track 48000 Hz, output 44100 Hz
W/AudioTrack: AUDIO_OUTPUT_FLAG_FAST denied by client; transfer 4, track 48000 Hz, output 44100 Hz
```

Por ejemplo, para el caso mostrado arriba, el dispositivo de prueba es un Nexus S, haciendo uso de la API 26. Tenemos un error que dice negar la reproducción en alta velocidad porque las frecuencias de muestreo usadas por las pistas no coinciden con las soportadas.

En cambio, observamos un caso radicalmente opuesto al hacer uso de otro dispositivo.

```
Run: app
I/OMXClient: Treble IOmx obtained
I/OMXClient: Treble IOmx obtained
I/System.out: DEBUG -> Fragments remaining = 0
I/AudioTrack: AUDIO_OUTPUT_FLAG_FAST successful; frameCount 115200 -> 115200
I/AudioTrack: AUDIO_OUTPUT_FLAG_FAST successful; frameCount 115200 -> 115200
W/AudioTrack: AUDIO_OUTPUT_FLAG_FAST denied by server; frameCount 115200 -> 115200
I/AudioTrack: AUDIO_OUTPUT_FLAG_FAST successful; frameCount 115200 -> 115200
```

Este es el caso de un Nexus S, también con la API 26. Sin realizar ningún cambio, la reproducción en alta velocidad es aceptada o rechazada sin un criterio observable. El mensaje de la frecuencia de muestreo ha desaparecido, y SoundPool rechaza ahora el modo rápido en la parte del servidor, y no del cliente, lo que indica que el problema está en la implementación nativa, que no podemos modificar.

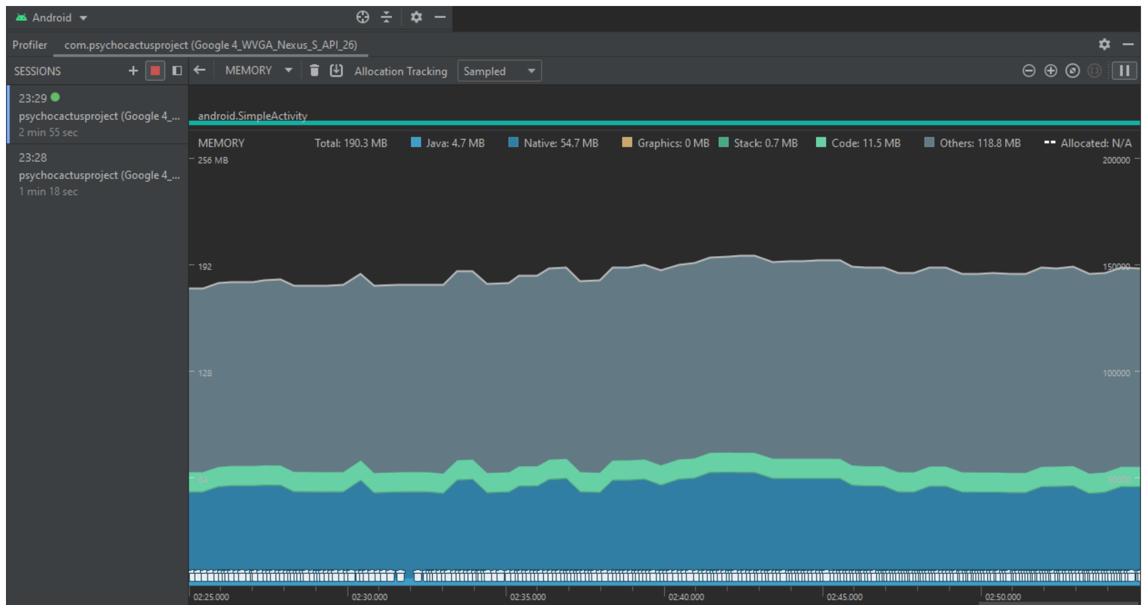
La solución, irremediablemente, pasa por realizar un estudio en profundidad de la forma en que el audio es tratado por Android e implementar vía OpenSL, lo que aumenta considerablemente los esfuerzos necesarios.

Mayor eficiencia en el uso de memoria RAM

Hay otro problema que salta a la vista cuando se hace un uso prolongado de la aplicación, y este es el del ciclo de vida de la Garbage Collector de la máquina virtual Java, y el de asignación de nuevos recursos físicos.

Como se ha podido comprobar, diferentes dispositivos implementan características de la máquina virtual de Java de manera distinta, algo que lleva a distintas eficiencias al correr el mismo código. Como contextualización, se debe recordar que la filosofía Java implica que todo su código es compatible con cada dispositivo que implemente una máquina virtual.

Esto tiene como desventaja que la transparencia hacia el desarrollador produce situaciones en que resulta complicado averiguar cuál es el origen de un problema de rendimiento.



Aquí se observa la utilidad con la que cuenta Android Studio para monitorizar el uso de RAM por parte del dispositivo. En el mismo terminal usado antes, un Nexus S, observamos una gestión de memoria realmente estable, sin picos apreciables, y que tiene como resultado una ejecución satisfactoria del código.

En contraste, sin haber aplicado ningún cambio en absoluto, la ejecución en un dispositivo Xiami Redmi 6 lanza un error fatal en algunas situaciones al intentar reservar memoria para utilizar las pistas de audio posteriormente. En concreto, el error observado es el siguiente.

```
app
I/OMXClient: IOmx service obtained
I/OMXClient: IOmx service obtained
W/libc: pthread_create failed: couldn't allocate 1040384-bytes mapped space: Operation not permitted
E/libutils.threads: androidCreateRawThreadEtc failed (entry=0xb2e4bcbd, res=11, Success)
    (android threadPriority=-16)
E/NdkMediaCodec: Failed to start the looper
E/SoundPool: Unable to load sample
W/libc: pthread_create failed: couldn't allocate 1040384-bytes mapped space: Operation not permitted
E/libutils.threads: androidCreateRawThreadEtc failed (entry=0xb2e4bcbd, res=11, Success)
```

Esto es realmente molesto, por el hecho de que la implementación no es en principio defectuosa, pero resulta un desastre solo en determinados dispositivos.

Mayor control sobre los hilos

Como se ha explicado anteriormente, todos los hilos son controlados por una clase creada específicamente para este propósito. Esta es la clase GameClock, la cual se beneficia de las herramientas ofrecidas por las clases Timer y TimerTask. Mientras que esto es de una

enorme facilidad de uso, deja entrever sus limitaciones a la hora en que el sistema planifica demasiados hilos, sucediendo en reducidas ocasiones bajadas de rendimiento apreciables.

Una manera de tener más control sobre las tareas que se mandan a ejecutar sería implementando estas mismas funcionalidades, sustituyendo las clases de Timer por la de ThreadPoolExecutor.

Mientras que con Timer estamos limitados a solicitar la creación de una nueva tarea, la cual creará presumiblemente un hilo cada vez, con el gasto de recursos que esto implica, ThreadPoolExecutor ofrece distintas características que hacen muy atractiva su inclusión en el motor. Ofrece utilidades para especificar el número de hilos que permitimos correr en paralelo, preconstruir hilos, mantener unos hilos determinados con vida por un tiempo prolongado, establecer colas de tareas y definir listeners que reaccionarán a determinados eventos relacionados con esta clase.

Desarrolladores de Android > Docs > Reference

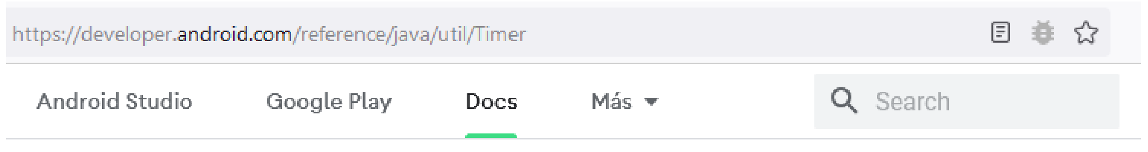
ThreadPoolExecutor

```
public class ThreadPoolExecutor
extends AbstractExecutorService
```

java.lang.Object
↳ java.util.concurrent.AbstractExecutorService
↳ java.util.concurrent.ThreadPoolExecutor

Known direct subclasses
ScheduledThreadPoolExecutor

Y este argumento es apoyado incluso por la propia referencia oficial de Android, la cual recomienda realizar esta mejora si la eficiencia es un asunto crítico [32].



Java 5.0 introduced the `java.util.concurrent` package and one of the concurrency utilities therein is the `ScheduledThreadPoolExecutor` which is a thread pool for repeatedly executing tasks at a given rate or delay. It is effectively a more versatile replacement for the `Timer` / `TimerTask` combination, as it allows multiple service threads, accepts various time units, and doesn't require subclassing `TimerTask` (just implement `Runnable`). Configuring `ScheduledThreadPoolExecutor` with one thread makes it equivalent to `Timer`.

1) Abstracción del motor

Aunque en el estado actual del desarrollo del motor creado para la realización de este proyecto considerar un desacoplamiento entre el videojuego y la arquitectura que le da soporte puede sonar algo lejano, lo cierto es que este esfuerzo podría reinvertirse con el propósito de crear nuevas experiencias jugables que hagan uso de las herramientas aquí desarrolladas.

Actualmente contamos con un sistema que podría permitir, con los cambios adecuados, cualquier tipo de videojuego en dos dimensiones, sea en tiempo real o por turnos, y con controles táctiles o físicos, ya que las mencionadas características han sido implementadas.

Los detalles sobre la viabilidad de según qué proyecto podrían ser discutidos de ser propuesto un género de videojuego, o conjunto de características que debería cumplir este supuesto nuevo proyecto. Pero lo cierto es que el objetivo que inicialmente se planteó ha sido cumplido hasta cierto punto, con lo que no parecería descabellado reorientar el desarrollo hacia nuevos entornos.

2) Características del juego

Pero antes de discutir nuevos proyectos, es necesario hacer un resumen de las funciones que han quedado en el tintero durante la realización de este proyecto.

Soporte para jugar una partida completa

Para la entrega de este Trabajo de Fin de Grado, la lógica de juego ha sido finalizada tan solo parcialmente. Así como tampoco se disponen de todas las animaciones que fueron especificadas, ni de las pistas de audio que acompañarán al juego final, no es posible actualmente tan solo realizando acciones jugables llegar hasta el final de la partida.

Esto se debe hacer introduciendo órdenes en la consola, lo que permite simular cuáles son las consecuencias de avanzar de turno, subir los valores de diversión, furia o fatiga, o que uno de los músicos resulte herido. Nada de esto tiene una traducción visual, ni los estados reales de cada personaje son actualizados. Esta es una tarea que no llevará demasiado tiempo tras la presente entrega, pero que debe ser considerada.

Nuevos niveles

Una vez finalizadas e implementadas las características que originalmente fueron concebidas para el videojuego, cabría pensar en la realización de nuevos niveles. Esto serían escenarios nuevos, con personajes hasta ahora no mostrados o la posibilidad de realizar acciones de nuevas maneras. Todo esto recordando que hablamos del mismo videojuego, y que cada nueva implementación debe guardar coherencia entre sí.

3) Lanzamiento comercial

Hace unos meses, cuando arrancó el proyecto, la idea original era la de subir la aplicación directamente a la Play Store, *marketplace* de aplicaciones Android, para permitir descargar una demostración con la total seguridad de que esta versión sería segura y habría pasado los filtros de Android, permitiendo documentar a su vez este proceso como parte del trabajo realizado, parte que podría entenderse, atendiendo al propio título que tiene esta memoria.

Ahora bien, parte de este trabajo ya fue realizado en el momento de obtener la primera

versión considerada estable de la aplicación, cuyo progreso iba a seguir siendo actualizado según se incluyesen nuevos avances. Pero tal como indica la propia Google Play Console, plataforma para lanzar aplicaciones en Google Play, “el tiempo de revisión es más largo de lo habitual”, lo que supone que, por desgracia, para la fecha de entrega de esta memoria, aún no ha sido aprobada.

7. Resultados

Teniendo en cuenta que en anteriores puntos ya se ha ido valorando progresivamente cómo el trabajo realizado difiere en determinadas medidas con los objetivos propuestos inicialmente, este punto de la memoria servirá principalmente para hacer un repaso del actual estado de la aplicación desarrollada, mostrando diferentes ejemplos prácticos de hitos logrados tras la finalización del Trabajo de Fin de Grado.

Cabe destacar que muchas de las características que aquí se van a mostrar no podrán ser expuestas en toda su extensión, pues este proyecto se trata de un videojuego, que, como se ha dicho, es un sistema en tiempo real, y resulta imposible presumir de determinados logros si no es demostranda la ejecución de esta aplicación en tiempo real; entiéndase el control, las secuencias de scripts, las animaciones o la propia reproducción de la música.

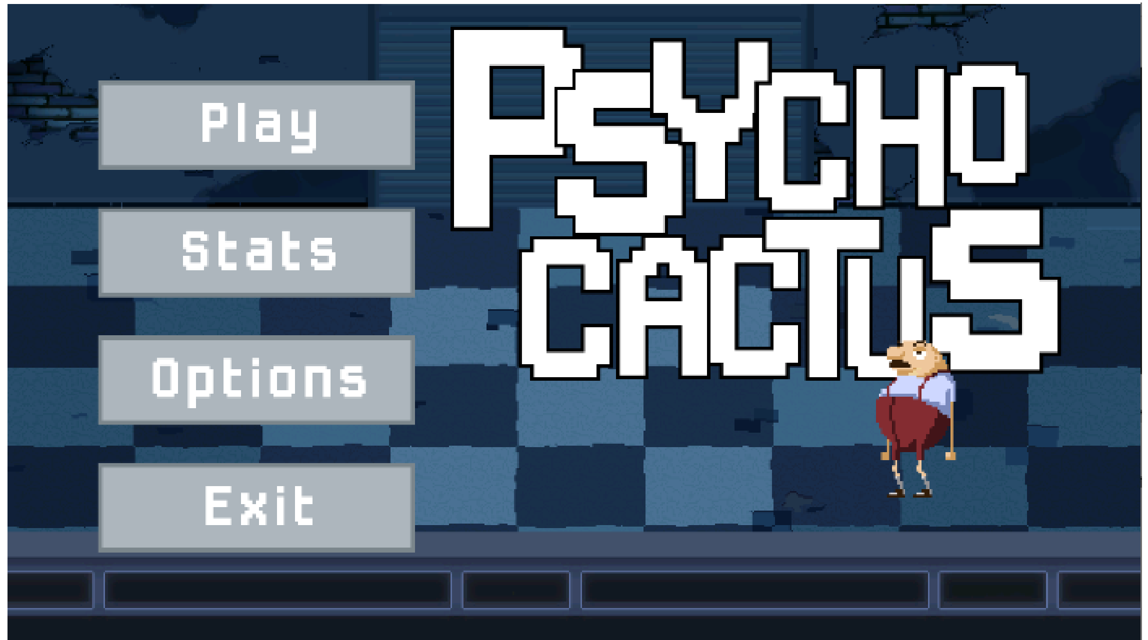
Para esta demostración será utilizado un emulador del modelo Nexus S, corriendo la versión de API 26, correspondiente a Android Oreo 8.0.

Pantalla de carga

Pantalla mostrada cuando es realizada una transición entre otras dos pantallas, y se desea ejecutar previamente una tarea de preparación que moviliza un volumen considerable de recursos. Muestra una etiqueta de carga con puntos suspensivos, y un personaje aleatorio en rotación para hacer animada la carga.

Pantalla de inicio / Menú principal

Pantalla que sirve para seleccionar las opciones principales y arrancar el juego. Permite acceder a las opciones o al historial de partidas. Esta última función no ha sido creada aún.



Pantalla de diálogo - Selección y alerta
En la misma pantalla de inicio podemos encontrar ambos tipos de diálogo existentes.

En el primer caso, como cabe esperar, el botón de Cancel cerrará el diálogo y el de Accept ejecutará la orden especificada. Para el caso del aviso tan solo podrá seleccionarse Okay y volver atrás.

La particularidad de la pantalla de diálogo es que implementa un alto grado de flexibilidad. En cualquiera de los casos, y sin importar desde dónde sea llamada, la escena de diálogo llamará también a la escena previamente seleccionada, oscurecerá la pantalla, y se mostrará por encima. Todas las etiquetas y botones que se muestran en el videojuego proceden de la misma clase utilitaria, y han sido definidos una sola vez. Permiten imprimir el texto centrando la altura y el ancho de cada línea, cuyos saltos son también automáticos.

Pantalla de juego
He de admitir, en primer lugar, que soy consciente de la inconsistencia visual de algunos elementos mostrados en pantalla, algo que sería un problema mayor si se aspirase a diseñar también el arte en este trabajo. Pero para estas versiones tempranas, no se le ha dado importancia a que los elementos de interfaz sean bonitos, sino funcionales.

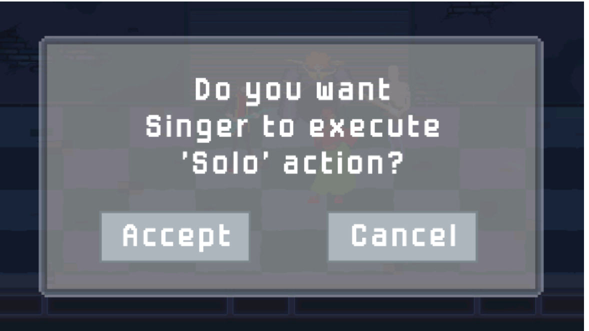
Esta que se muestra aquí es, pues, la pantalla principal de juego. La captura ha sido realizada con varios objetos en pantalla cuya función será explicada ahora.

En la esquina superior derecha se encuentra el botón de pausa, que de ser pulsado nos llevará a dicha escena. En la parte izquierda, el medidor de diversión, que indica gracias a la flecha situada sobre el color amarillo el grado de este atributo presente en el videojuego. Y en la parte inferior, lo que pretende ser un monitor del turno en que nos encontramos. Como se ha dicho, todos los elementos mencionados en este párrafo no han pasado por ningún filtro estético, pero serán sustituidos por otros modos de visualización más agradables en el futuro.

Luego, cada uno de los cuatro personajes mostrados es un músico. Estos músicos están animados, brillan de manera intermitente si pueden ser seleccionados, y al pulsar sobre ellos abren un menú contextual que permite realizar acciones.

Zoom

Aunque no se pueda apreciar correctamente, se tomarán distintas capturas de fases que recorre la pantalla cuando se realiza el efecto gráfico de ampliación.



Primero, con la selección de cualquier opción del menú contextual, saltará el diálogo mostrando este mensaje.

De aceptar, comienza el efecto de zoom.

Pantalla de pausa

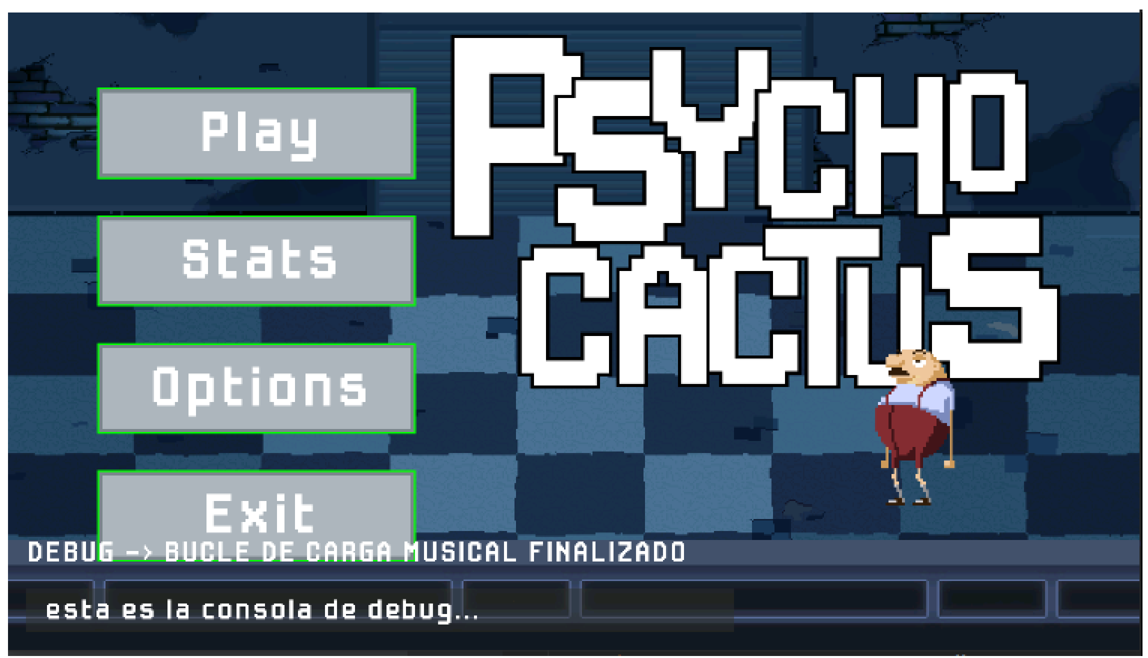
Estado del juego en que la acción se detiene y es posible solicitar una reanudación, un reinicio, salir al menú principal, o acceder al menú de opciones.

Pantalla de opciones

Y en el caso de que se acceda al menú de opciones, aquí tendremos los diferentes ajustes que en esta versión son modificables. Se permite salir sin guardar cambios, cargar los ajustes por defecto, o guardar cambios.

Modo de debug

Se ha mencionado en más de una ocasión cómo el modo debug imprime información en pantalla muy útil si se desea conocer cuál es el estado lógico de algunos elementos del motor del videojuego. Un par de ejemplos se muestran aquí.



8. Conclusiones

Dejar de admitir todas las dificultades que ha supuesto la elección de la tecnología, el framework con el que se contaba, y la magnitud del sistema a desarrollar no sería en absoluto honesto ni haría justicia al trabajo realizado.

Es cierto que programar un sistema desde un punto inicial relativamente primitivo no es una tarea que necesariamente puede resultar satisfactoria. Hoy en día, la opción que menos problemas presenta para realizar un videojuego sea del tipo que sea es aprender a exprimir al máximo las capacidades de un motor de videojuegos de primera orden. Las diferencias son evidentes. Usar un motor de videojuegos ya construido supone perder un grado de libertad enorme, pero también implica la ventaja de contar con un equipo inmenso de profesionales a tus espaldas, y una comunidad repleta de desarrolladores que se han cruzado con los mismos problemas que tú en algún momento.

A pesar de esta, que es la parte negativa, yo, como aspirante a ingeniero informático, y como individuo apasionado por la tecnología y por los sistemas software, he disfrutado como nunca de enfrentarme a este reto, y he extraído lecciones únicas durante el proceso.

En algún momento al principio del desarrollo llegué a pensar durante mis estimaciones más optimistas que llegaría esta fecha y podría presentar un completo producto listo para lanzar al mercado, cuando la realidad no ha sido esta. Cada día, a cada nuevo paso que daba, retrocedía por otros motivos. Realizar por primera vez cualquier proyecto supone un gran grado de incertidumbre, contra el que es difícil lanzar predicciones acertadas a largo plazo. También es cierto que el punto final del desarrollo no ha decepcionado mis expectativas, ya conocía desde un primer momento la dificultad que conlleva hacer realidad un videojuego, por pequeño que sea.

Y en un proyecto de estas características, no solo entra en juego lo que yo soy capaz de ofrecer. Es innegable que contar con recursos es un factor diferencial, y si se pretende lograr un objetivo partiendo de un bajo presupuesto, es complicado que terceras personas cumplan con promesas y compromisos. A pesar de esto, descubrir estas propias limitaciones ha sido una experiencia positiva en sí misma.

Llegados a este fragmento de la memoria del trabajo realizado, es posible ver en perspectiva qué significó este proyecto durante cada fase de su desarrollo. Se puede decir que se ha convertido en algo mayor que una mera tarea protocolaria para poner punto final a la carrera estudiada, pasando a formar parte de la vida diaria de uno como estudiante y profesional. Creo que las conclusiones merecen ser tratadas ligeramente desde un punto de vista más personal, ya que este mismo proyecto habría tenido un resultado posiblemente muy distinto de haber sido realizado por cualquier persona que no fuese yo mismo; ni habría tomado las mismas decisiones, ni habría resuelto los problemas que surgían de la misma forma.

En cualquier caso, creo que la experiencia de remar a contracorriente en un sentido tecnológico es algo extremadamente enriquecedor. Tal como a veces se encuentran noticias de apasionados que lanzan, ya bien entrados en el siglo XXI, un nuevo juego para la Nintendo Entertainment System, llevar a cabo un proyecto de una forma que ya no es común hoy en día

te enseña a pensar diferente, a adaptarte a diferentes entornos, y a entender el tema estudiado desde un punto de vista amplio.

De nuevo, personalmente, recomiendo a cualquier estudiante que pueda en algún momento leer estas líneas a embarcarse en un proyecto de características similares. Porque, aunque haya formas más fáciles de hacer la misma tarea, aspirar a realizar algo grande significa adquirir conocimientos que siempre serán de un gran valor.

9. Referencias

[1] Autor: GSMA, Título: The Mobile Economy, 2021, [Online] Disponible: https://www.gsma.com/mobileeconomy/, Última fecha de acceso: 08/07/2021.

[2] Autor: Strategy Analytics, Título: Global Connected and IoT Device Forecast Update, 2019, [Online] Disponible: https://www.strategyanalytics.com/access-services/devices/connected-home/consumer-electronics/reports/report-detail/global-connected-and-iot-device-forecast-update, Última fecha de acceso: 08/07/2021.

[3] Autor: statcounter, Título: Mobile Operating System Market Share Worldwide, 2021, [Online] Disponible: https://gs.statcounter.com/os-market-share/mobile/worldwide, Última fecha de acceso: 08/07/2021.

[4] Autor: Iván Linares, Título 15 curiosidades de la tienda Google Play Store, 2021, [Online] Disponible: https://www.xatakandroid.com/play-store/15-curiosidades-tienda-google-play-store, Última fecha de acceso: 08/07/2021.

[5] Autor: statista, Título: Los lenguajes de programación más usados del mundo, 2019, [Online] Disponible: https://es.statista.com/grafico/16580/lenguajes-de-programacion-mas-usados-del-mundo/, Última fecha de acceso: 08/07/2021.

[6] Autor: Gabriel Erard, Título: Google vence a Oracle en su larga disputa legal: Android no violó los derechos de autor de Java, 2021, [Online] Disponible: https://hipertextual.com/2021/04/google-vence-oracle-derechos-de-autor-android-java, Última fecha de acceso: 08/07/2021.

[7] Autor: Maxim Shafirov, Título: Kotlin on Android. Now official, 2017, [Online] Disponible: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/, Última fecha de acceso: 08/07/2021.

[8] Autor: Jerónimo Palacios, Título: Agile tiene cuatro veces más posibilidades de éxito que Waterfall, 2016, [Online] Disponible: https://jeronimopalacios.com/agile/agile-tiene-cuatro-veces-mas-posibilidades-de-exito-que-waterfall/, Última fecha de acceso: 08/07/2021.

[9] Autor: The Standish Group, Título: CHAOS, 2020, [Online] Disponible: https://standishgroup.myshopify.com/, Última fecha de acceso: 08/07/2021.

[10] Autor: Daniel Sánchez-Crespo Dalmau, Título: Core Techniques and Algorithms in Game Programming, Editorial: Berkeley New Riders, Año: 2004

[11] Autor: Andrew Davison, Título: Killer Game Programming in Java, Editorial: O'Reilly Media, Año: 2005

[12] Autor: John Horton y Raul Portales, Título: “Android : game programming : extend your game development skills by harnessing the power of Android SDK : a course in three modules”, Editorial: Packt, Año: 2016

[13] Autor: DayoScript, Título: El crash del 83 | La Leyenda del Videojuego [Episodio 4], 2019, [Online] Disponible: https://www.youtube.com/watch?v=G39At1Ojx-E, Última fecha de acceso: 08/07/2021.

[14] Autor: Joel Engel and Richard Frenkiel, Título: Fascinating facts about the invention of the Mobile Phone, 1983, [Online] Disponible: https://web.archive.org/web/20110713002602/http://www.ideafinder.com/history/inventions/mobilephone.htm, Última fecha de acceso: 08/07/2021.

[15] Autor: Chris Wright, Título: A Brief History of Mobile Games: In the beginning, there was Snake, 2016, [Online] Disponible: https://www.pocketgamer.biz/feature/10619/a-brief-history-of-mobile-games-in-the-beginning-there-was-snake/, Última fecha de acceso: 08/07/2021.

[16] Autor: Steve Jobs (John Schroter, YouTube), Título: Steve Jobs introduces iPhone in 2007, 2011, [Online] Disponible: https://www.youtube.com/watch?v=MnrJzXM7a6o, Última fecha de acceso: 08/07/2021.

[17] Autor: T. Ricker, Título: Jobs: App Store launching with 500 iPhone applications, 25% free, 2008, [Online] Disponible: https://www.engadget.com/2008-07-10-jobs-app-store-launching-with-500-iphone-applications-25-free.html, Última fecha de acceso: 08/07/2021.

[18] Autor: Rik Myslewski, Título: iPhone App Store breezes past 500 million downloads, 2009, [Online] Disponible: https://www.theregister.com/2009/01/16/half_billion_iphone_apps/, Última fecha de acceso: 08/07/2021.

[19] Autor: Varios autores (Wikipedia), Título: List of highest-grossing mobile games, 2021, [Online] Disponible: https://en.wikipedia.org/wiki/List_of_highest-grossing_mobile_games, Última fecha de acceso: 08/07/2021.

[20] Autor: IGN, Título: Fortnite vs Apple: Epic convierte su disputa legal en un evento dentro del juego, 2020, [Online] Disponible: https://es.ign.com/fortnite-pc/166305/news/fortnite-vs-apple-epic-convierte-su-disputa-legal-en-un-evento-dentro-del-juego, Última fecha de acceso: 08/07/2021.

[21] Autor: Patrick McGee, Título: Epic vs Apple: lo que hemos aprendido del juicio que podría cambiar el iPhone, 2021, [Online] Disponible: https://www.expansion.com/empresas/2021/05/25/60acf64a468aeba5718b45f8.html, Última fecha de acceso: 08/07/2021.

[22] Autor: Nick Caston, Título: Unity DOTS vs Handbuilt: Sample Project, 2019, [Online] Disponible: https://www.youtube.com/watch?v=tInaI3pU19Y, Última fecha de acceso: 08/07/2021.

[24] Autor: Jesse Schell, Título: The Art of Video Game Design: a book of lenses, Editorial: Boca Raton, Año: 2015

[25] Autor: Carlos González, Título: ¿Qué es ART? Android Runtime, el sucesor de Dalvik, 2019, [Online] Disponible: https://androidayuda.com/android/que-es/art-android-runtime/, Última fecha de acceso: 08/07/2021.

[26] Autor: Google, Título: Diseños, 2021, [Online] Disponible: https://developer.android.com/guide/topics/ui/declaring-layout, Última fecha de acceso: 08/07/2021.

[27] Autor: Google, Título: SurfaceHolder#lockCanvas(), 2021, [Online] Disponible: https://developer.android.com/reference/android/view/SurfaceHolder#lockCanvas(), Última fecha de acceso: 08/07/2021.

[28] Autor: Google, Título: Introducción a las actividades, 2021, [Online] Disponible: https://developer.android.com/guide/components/activities/intro-activities, Última fecha de acceso: 08/07/2021.

[29] Autor: Google, Título: Cómo interpretar el ciclo de vida de una actividad, 2021, [Online] Disponible: https://developer.android.com/guide/components/activities/activity-lifecycle, Última fecha de acceso: 08/07/2021.

[30] Autor: Dianne Hackborn, Título: The Android 3.0 Fragments API, 2011, [Online] Disponible: https://android-developers.googleblog.com/2011/02/android-30-fragments-api.html, Última fecha de acceso: 08/07/2021.

[31] Autor: Google, Título: Diseño para latencia reducida, 2021, [Online] Disponible: https://source.android.com/devices/audio/latency/design, Última fecha de acceso: 08/07/2021.

[32] Autor: Google, Título: ThreadPoolExecutor, 2021, [Online] Disponible: https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor, Última fecha de acceso: 08/07/2021.