UNIVERSIDAD DE ALMERÍA

ESCUELA POLITÉCNICA SUPERIOR Y FACULTAD DE CIENCIAS EXPERIMENTALES

INGENIERÍA INFORMÁTICA

DroidEngine2D

Motor para el desarrollo de videojuegos 2D para Android

El Alumno:
Miguel Vicente Linares

Director:
Antonio Leopoldo Corral Liria

Almería, Julio de 2014

Agradecimientos

A mi familia, y en especial a mis padres y mi hermano por su apoyo desde siempre y por su paciencia.

A mi tutor, Antonio Corral, por el asesoramiento, por los consejos y por animarme en todo momento a terminar el proyecto.

A mis compañeros, en especial a Jorge García, Sergio Revueltas y Antonio Vicente, por el apoyo durante los 5 años de carrera.

A mis profesores, por enseñarme las bases de los conocimientos técnicos que me han permitido, entre otras cosas, desarrollar este proyecto. En particular a los profesores Manuel Torres, Vicente González, Joaquín Cañadas, Leo González y Julio Barón por aceptar amablemente formar parte del tribunal que evaluará este proyecto.

A cualquier persona que me haya ayudado, de forma directa o indirecta, a llevar a cabo este proyecto.

Gracias.

Índice de contenidos

1. Introducción ... 1
1.1. Introducción al desarrollo de videojuegos ... 1
1.2. Roles en un equipo de desarrollo de videojuegos ... 1
1.2.1. Ingenieros y programadores ... 2
1.2.2. Artistas ... 3
1.2.3. Diseñadores de juegos ... 3
1.2.4. Productores ... 4
1.2.5. Personal de gestión y apoyo ... 4
1.3. Motores de juegos ... 4
1.3.1. ¿Cómo surgieron los primeros motores de juegos? ... 4
1.3.2. ¿Qué es un motor de juegos? ... 5
1.3.3. Motores optimizados para un género de videojuegos ... 5
1.3.4. Arquitectura de un motor de juegos ... 9
1.3.5. El bucle del juego ... 18
1.4. Componentes de una aplicación para Android ... 22
1.4.1. Activities ... 23
1.4.2. Services ... 24
1.4.3. Content providers ... 25
1.4.4. Broadcast receivers ... 25
1.5. Motivación ... 25
1.6. Objetivos generales del proyecto ... 26
1.7. Planificación temporal ... 27
2. Estado del arte ... 31
2.1. Introducción ... 31
2.2. Revisión de varios motores de juegos ... 31
2.2.1. Unity ... 32
2.2.2. Unreal Engine ... 33
2.2.3. UDK ... 34
2.2.4. CryEngine ... 35
2.2.5. OGRE3D ... 35
2.2.6. Open Game Engine ... 36
2.2.7. LibGDX ... 36

2.2.8. AndEngine ... 37
2.3. Conclusiones ... 38
3. OpenGL ES ... 39
3.1. Introducción ... 39
3.2. ¿Qué es OpenGL ES? ... 39
3.3. Versiones de OpenGL ES ... 40
3.3.1. OpenGL ES 1.X ... 40
3.3.2. OpenGL ES 2.X ... 41
3.3.3. OpenGL ES 3.X ... 41
3.4. Sistemas de coordenadas en OpenGL ... 42
3.4.1. Object Coordinates ... 43
3.4.2. World Coordinates ... 44
3.4.3. Eye Coordinates ... 46
3.4.4. Clip Coordinates ... 46
3.4.5. Normalized Device Coordinates ... 48
3.4.6. Window Coordinates ... 49
3.5. Pipelines de rendering en OpenGL ES ... 49
3.5.1. Pipeline de rendering de función fija ... 50
3.5.2. Pipeline de rendering programable ... 55
3.6. Soporte de OpenGL ES en Android ... 59
3.7. ¿Por qué utiliza DroidEngine2D OpenGL ES 2.0? ... 60
3.8. Conclusiones ... 61
4. DroidEngine2D ... 63
4.1. Introducción ... 63
4.1.1. ¿Qué es DroidEngine2D? ... 63
4.1.2. Beneficios de DroidEngine2D ... 64
4.2. Descripción general del proyecto ... 65
4.2.1. Perspectiva del producto ... 65
4.2.2. Funciones del producto ... 65
4.2.3. Características de los usuarios ... 69
4.2.4. Restricciones ... 69
4.2.5. Suposiciones y dependencias ... 70
4.3. Desarrollo del proyecto ... 70
4.3.1. Características generales ... 71
4.3.2. Audio ... 78

4.3.3. Estados del juego ... 80
4.3.4. Gráficos ... 85
4.3.5. Entrada de usuario ... 102
4.3.6. Carga de recursos ... 110
4.3.7. Utilidades ... 110
4.4. Ejemplos de uso ... 113
4.4.1. Demos de características individuales ... 114
4.4.2. Demo de un minijuego desarrollado con DroidEngine2D ... 115
5. Conclusiones ... 119
5.1. Conclusiones generales ... 119
5.2. Posibles mejoras y trabajos futuros ... 120
6. Bibliografía ... 123
6.1. Referencias ... 123
Apéndice A. BMFont ... 131
A.1. Introducción ... 131
A.2. ¿Qué es BMFont? ... 131
A.3. Obtener BMFont ... 131
A.4. Generar fuentes compatibles con DroidEngine2D ... 132
Apéndice B. TexturePacker ... 135
B.1. Introducción ... 135
B.2. ¿Qué es TexturePacker? ... 135
B.3. Obtener TexturePacker ... 135
B.4. Generar atlas compatibles con DroidEngine2D ... 136
Apéndice C. Diagrama de clases de DroidEngine2D ... 141
C.1. Descripción ... 141

1

Introducción

1.1. Introducción al desarrollo de videojuegos

En la actualidad (Julio, 2014), la industria del videojuego está en constante crecimiento y tiene una presencia bastante notable en el mercado global, superando en ingresos a las industrias del cine y la música. Los videojuegos pueden ser desarrollados para multitud de plataformas, entre las que podemos destacar principalmente: ordenadores, consolas de sobremesa, consolas portátiles, móviles y tablets, y navegadores web.

El desarrollo de un videojuego es un proceso complejo que requiere la implicación de profesionales de diversos perfiles, los cuales se explican más detalladamente a continuación.

1.2. Roles en un equipo de desarrollo de videojuegos

En un equipo de desarrollo de videojuegos normalmente colaboran profesionales de las siguientes, aunque no únicas, 5 disciplinas [1]: ingenieros y programadores, artistas, diseñadores de juegos, productores, y personal de gestión y apoyo (marketing, asesoramiento legal, personal administrativo, etc.).

En equipos de desarrollo grandes, es común que cada persona se especialice en un área concreta, mientras que en equipos pequeños, normalmente los roles no están tan diferenciados y una misma persona puede adoptar varios roles durante el desarrollo.

Cada una de las disciplinas mencionadas, a su vez engloba a profesionales de distintos perfiles.

A continuación se explica, a modo de resumen, el papel de cada una de estas disciplinas en el desarrollo de videojuegos. También se mencionan los distintos perfiles profesionales que se pueden encontrar en cada una de ellas.

Los distintos perfiles que se explican a continuación son los más destacados de los que normalmente se pueden encontrar en un estudio de desarrollo de videojuegos grande, que normalmente cuenta con presupuestos bastante elevados.

En equipos de desarrollo pequeños no están presentes todos los perfiles, y en muchas ocasiones cada persona adopta varios roles. También es común que algunos de estos perfiles no formen parte del equipo de forma permanente, sino que se contraten sus servicios para proyectos concretos o en etapas concretas del desarrollo de un proyecto.

1.2.1. Ingenieros y programadores

Los ingenieros y programadores diseñan e implementan el software del juego y las herramientas necesarias para el desarrollo del mismo [1].

Es común que los ingenieros y programadores se dividan en tres grupos en función de las tareas de las que se encargan:

- Los **desarrolladores de herramientas**, que diseñan e implementan herramientas que faciliten la labor de los demás en el desarrollo del juego. Estas herramientas pueden ser muy complejas, como editores de escenarios 3D, o más simples, como scripts que permitan automatizar otras tareas para reducir el tiempo de desarrollo.
- Los **desarrolladores del motor de juegos**, que diseñan e implementan el motor del juego. Este perfil solo se encuentra en estudios de desarrollo grandes, que desarrollan su propio motor o realizan modificaciones a motores comerciales porque éstos no se adaptan completamente a sus necesidades.
- Los **desarrolladores de las mecánicas del juego**, que se encargan de implementar la lógica del propio juego, utilizando el motor y las herramientas creadas por otros.

Normalmente, en las grandes compañías de desarrollo de videojuegos, los ingenieros se encargan en mayor medida del diseño de la arquitectura del software y de planificar y coordinar el trabajo de los programadores. Aunque también contribuyen escribiendo código, sobre todo en las partes en las que se requiere un mayor nivel de especialización.

Como se puede deducir de lo expuesto anteriormente, en el campo de la programación en concreto, se requieren conocimientos en multitud de disciplinas, como por ejemplo: gráficos y uso de APIs gráficas como OpenGL o DirectX, físicas, inteligencia artificial, audio, lógica del juego, scripting, comunicación en red (para juegos multijugador online o mantener tablas de puntuaciones online, por ejemplo), procesamiento de la entrada de usuario, creación de herramientas que faciliten el trabajo del equipo de desarrollo, etc. Por tanto, lo normal es que en el desarrollo de un videojuego complejo participen varios programadores, cada uno especializado en una o varias áreas.

1.2.2. Artistas

Los artistas se encargan de producir todo el contenido gráfico y el audio de un videojuego [1].

En el proceso de desarrollo de un videojuego se pueden encontrar artistas de múltiples perfiles:

- Los **artistas de concepto**, que se encargan de crear bocetos del juego y de los personajes para que el equipo tenga una visión de cómo quedará el producto final.
- Los **modeladores 3D**, que producen los modelos (geometría tridimensional) de todos los objetos del mundo 3D virtual. Normalmente entre los modeladores 3D se distinguen dos especialidades, los que se encargan de modelar los objetos en primer plano (personajes, vehículos, armas, etc.) y los que se encargan de modelar los objetos en segundo plano (terreno, edificios, etc.).
- Los **artistas de texturas**, que crean texturas 2D que se aplican a los modelos 3D para conseguir detalles y realismo.
- Los **artistas 2D**, que se encargan de diseñar y crear elementos 2D, como elementos de la interfaz de usuario, logotipos, etc.
- Los **artistas de iluminación**, que se encargan de colocar fuentes de luz en el mundo virtual y juegan con los colores, la intensidad y la dirección de la luz para mejorar el impacto visual y emocional del juego.
- Los **animadores**, que se encargan de dotar de movimiento a los personajes y los objetos del juego.
- Los **actores de captura de movimiento**, que se encargan de realizar los movimientos de los personajes del juego. Estos movimientos son capturados y utilizados por los animadores como base para crear las animaciones de los personajes.
- Los **diseñadores de sonido**, se encargan de diseñar y producir y mezclar la música y efectos de sonido del juego.
- Los **actores de voz**, que se encargan de proporcionar voces a los personajes del juego.
- Los **compositores**, que se encargan de componer bandas sonoras específicamente diseñadas para el juego.

1.2.3. Diseñadores de juegos

El trabajo del diseñador de juegos es diseñar la parte interactiva del juego, lo que se conoce normalmente como las mecánicas del juego [1].

Los diseñadores se encargan de crear la historia del juego, determinar cómo componer el entorno, definir la personalidad de los personajes, definir las acciones que pueden realizar los personajes, etc.

En muchos casos, los diseñadores no solo definen los conceptos para que los programadores los implementen en el juego, sino que ellos mismos se encargan también de implementar comportamientos en el juego. Para ello, normalmente utilizan lenguajes de scripting de alto nivel.

En algunos equipos se contratan escritores, que ayudan a los diseñadores a plasmar sus ideas y a componer la historia del juego y escribir líneas de diálogo.

1.2.4. Productores

El rol del productor en los estudios de desarrollo de videojuegos se define de distintas formas según el estudio. Normalmente el productor se encarga de gestionar la planificación temporal del desarrollo y actúa como jefe de recursos humanos. En otras compañías, los productores actúan como diseñadores de juego senior, o como enlace entre el equipo de desarrollo y la unidad de negocio de la compañía [1].

1.2.5. Personal de gestión y apoyo

Este grupo engloba a todos los demás profesionales que contribuyen al desarrollo y comercialización de un videojuego [1]. Incluye a todo el personal de gestión, marketing, administrativos, soporte técnico, etc.

1.3. Motores de juegos

En esta sección se hará un breve repaso de cómo surgieron los primeros motores de juegos, y qué son los motores de juegos tal y como se conocen en la actualidad (Julio, 2014). Además se explicarán los principales componentes o subsistemas que suelen estar presentes en los motores de juegos actuales (Julio, 2014).

1.3.1. ¿Cómo surgieron los primeros motores de juegos?

El término “motor de juegos” apareció a mediados de la década de 1990 a causa de la forma en la que estaba diseñada la arquitectura interna de *Doom* (1993), el famoso videojuego desarrollado por *id Software* [1 y 2].

La arquitectura interna de *Doom* estaba diseñada de forma que se podía distinguir una separación razonablemente bien definida entre los componentes del núcleo del sistema (sistema de rendering, sistema de detección de colisiones, sistema de audio, etc.) y los componentes que estaban fuertemente ligados al propio juego (recursos gráficos, sonidos, escenarios del mundo virtual del juego, implementación de las mecánicas del juego, etc.) [1].

El valor de realizar esta separación del código del juego en dos partes diferenciadas se hizo evidente cuando los desarrolladores comenzaron a crear juegos y convertirlos en juegos nuevos simplemente cambiando la parte del código referente al juego, sin tener que reescribir el sistema de rendering, el sistema de audio, etc.

Posteriormente se fueron agregando nuevas características a los motores de juegos, como por ejemplo, la posibilidad de utilizar lenguajes de scripting de alto nivel para definir las mecánicas del juego, mientras que el propio motor suele estar implementado en lenguajes compilados de más bajo nivel como por ejemplo C o C++.

A finales de la década de 1990, varios estudios de videojuegos desarrollaban su propio motor de juegos. Estos estudios comenzaron a vender licencias del uso de sus motores a otros desarrolladores como una vía secundaria de generación de ingresos [1].

1.3.2. ¿Qué es un motor de juegos?

Un motor de juegos es un sistema diseñado para facilitar la creación y el desarrollo de videojuegos.

Está formado por un conjunto de componentes reutilizables que proporcionan al desarrollador de videojuegos gran parte de la funcionalidad básica que necesita para desarrollar un videojuego. Esto supone una gran ventaja, puesto que de esta forma el programador puede centrarse en escribir el código relacionado con la lógica del juego y no tiene que preocuparse, por ejemplo, de cómo comunicarse con la GPU para renderizar (anglicismo derivado del verbo *render*, que se usa en el contexto de los gráficos por ordenador) un elemento en pantalla, puesto que esa funcionalidad la proporciona el motor.

Un buen motor de juegos es normalmente extensible y puede ser utilizado para crear multitud de juegos diferentes sin necesidad de hacer grandes modificaciones al motor.

1.3.3. Motores optimizados para un género de videojuegos

Normalmente, cada motor de juegos está optimizado para un género de videojuegos concreto.

Esto se debe a que la complejidad de crear un motor que esté optimizado para todos los géneros de videojuegos hace que ésta tarea sea prácticamente imposible.

Como se ha mencionado en apartados anteriores, hay componentes o sistemas que se repiten en la mayoría de videojuegos, por ejemplo, la gestión a bajo nivel de la entrada de usuario, o un sistema de reproducción de audio, o de rendering.

Sin embargo, hay otros componentes que son comunes a todos los juegos de un género concreto, pero que en otros géneros no son necesarios.

Otra cuestión a tener en cuenta son las optimizaciones realizadas para un género de videojuegos concreto. Si un componente está optimizado par aun género en concreto, es posible que si se utiliza para un género distinto, dicha optimización, en lugar de mejorar el rendimiento, haga que éste disminuya.

También hay que tener en cuenta que hay estudios de videojuegos que únicamente desarrollan juegos de un género en concreto. Estos estudios generalmente preferirán utilizar un motor que esté optimizado para juegos del género que ellos desarrollan antes que utilizar un motor que intente abarcar más géneros pero que no esté tan optimizado para sus necesidades.

A continuación se exponen los principales géneros de videojuegos, así como los requisitos que debe cumplir un motor de juegos que vaya a ser utilizado para cada uno de los mismos [1].

1.3.3.1. Videojuegos de disparos en primera persona

Los videojuegos de disparos en primera persona, también conocidos como FPS (*First-Person Shooter*), son un género de videojuegos y subgénero de los videojuegos de disparos en los que el jugador observa el mundo desde la perspectiva del personaje protagonista [3].

Los videojuegos de este género son algunos de los más difíciles de crear, por limitaciones tecnológicas. Esto se debe a que normalmente se intenta proporcionar al jugador la ilusión de que está inmerso en un mundo realista y muy detallado. No es una sorpresa que muchas de las grandes innovaciones tecnológicas de la industria de los videojuegos aparecieran al crear juegos de este género [1].

Las tecnologías en las que se suelen centrar los videojuegos de disparos en primera persona son:

- Rendering eficiente de mundos virtuales muy grandes.
- Controles de cámara y mecánicas de apuntado sensibles y precisas.
- Animaciones detalladas y de gran calidad de los brazos y armas del jugador.
- Gran variedad de armas de mano para el jugador.
- Control de movimiento y colisiones con el entorno no demasiado estricto, lo cual puede dar la impresión al jugador de que el personaje está flotando.
- Inteligencia artificial y animaciones de gran calidad para los personajes enemigos y aliados.
- Capacidades multijugador a pequeña escala (soportando normalmente hasta 64 jugadores por partida).

Normalmente, el sistema de rendering utilizado en este tipo de juegos se optimiza según el entorno que se está renderizando. Por ejemplo, los juegos que representan interiores (edificios, mazmorras, etc.) suelen utilizar un sistema de rendering optimizado mediante el método de partición binaria del espacio, también llamado BSP (*Binary Space Partitioning*), o un sistema de rendering basado en portales.

Sin embargo, los videojuegos de disparos en primera persona cuyos escenarios representan principalmente exteriores, utilizan técnicas de optimización del rendering muy distintas, como la eliminación de objetos ocultos (*Occlusion Culling*), o la división previa del escenario en sectores y comprobación en tiempo de ejecución de los sectores visibles para el jugador.

1.3.3.2. Videojuegos de plataformas y otros juegos en tercera persona

Los videojuegos de plataformas son un género de videojuegos cuya mecánica principal consiste en guiar a un avatar saltando sobre plataformas que se encuentran suspendidas en el aire [4].

En términos de requisitos tecnológicos, los juegos de plataformas normalmente se pueden agrupar con los videojuegos de disparos en tercera persona y los videojuegos de acción o aventura en tercera persona.

Los videojuegos en tercera persona tienen mucho en común con los videojuegos de disparos en primera persona, pero hacen más énfasis en las habilidades del personaje principal y sus animaciones.

Algunas de las tecnologías en las que este tipo de juegos se centra específicamente son [1]:

- Plataformas móviles, escaleras, cuerdas, etc.
- Puzles integrados en el entorno.
- Cámara en tercera persona que sigue al personaje principal. Esta cámara puede ser controlada por el jugador.
- Un sistema complejo de colisiones de cámara que asegure que cuando el jugador rote la cámara, no atraviese objetos.

1.3.3.3. Videojuegos de lucha

Los videojuegos de lucha son un género de videojuegos en el que el jugador controla un personaje que se enfrenta en combate a corta distancia con uno o varios oponentes. Los combates suelen consistir en varios asaltos que tienen lugar en un ring o una arena [5].

Tradicionalmente los desarrolladores de videojuegos de este género han centrado sus esfuerzos en mejorar la tecnología relacionada con los siguientes aspectos [1]:

- Un amplio conjunto de animaciones de lucha.
- Detección de colisiones precisa.
- Sistema de gestión de entrada de usuario capaz de detectar combinaciones de botones complejas.

Y en la actualidad (Julio, 2014), los juegos de lucha en 3D además requieren:

- Shaders que permitan renderizar piel con aspecto realista. Esto incluye efectos como la simulación de sudor en la piel, por ejemplo.
- Animaciones de gran calidad.
- Simulación de los movimientos del pelo y la ropa basada en físicas.

Como el mundo 3D de estos juegos es pequeño y la cámara está centrada en la acción en todo momento, estos juegos generalmente no necesitan aplicar técnicas de optimización de rendering como la subdivisión espacial del escenario o la eliminación

de objetos ocultos. Tampoco se espera que necesiten modelar la propagación del audio, por ejemplo.

1.3.3.4. Videojuegos de carreras

Los videojuegos de carreras son un género de videojuegos en el que el jugador participa en una competición con cualquier tipo de vehículo [6].

Los videojuegos de carreras normalmente centran todos detalles gráficos en los vehículos, el circuito y los alrededores del mismo. Y, en caso de que en el juego se pueda ver al piloto, como es el caso de los videojuegos de carreras de karts, también se dedica bastante esfuerzo a renderizar correctamente al piloto y sus animaciones.

Algunos de los requerimientos tecnológicos de este género de videojuegos son [1]:

- Se utilizan algunos trucos para renderizar el fondo del juego. Los elementos que están fuera del circuito y se ven a lo lejos se suelen renderizar como imágenes en 2 dimensiones en lugar de utilizar objetos 3D complejos.
- El circuito se suele dividir en sectores bidimensionales, que se utilizan para optimizar el rendering (determinación de zonas visibles) y para facilitar la implementación de la inteligencia artificial de los vehículos no controlados por el jugador, además de para resolver otros problemas técnicos.
- La cámara suele seguir al vehículo del jugador, simulando una perspectiva en tercera persona, aunque también puede estar dentro del vehículo, simulando la conducción en primera persona.
- Si en el circuito hay túneles u otros lugares cerrados, es necesario asegurarse de que la cámara no colisiona con la geometría del escenario.

1.3.3.5. Videojuegos de estrategia en tiempo real

En los videojuegos de estrategia en tiempo real, también conocidos como RTS (*Real-Time Strategy*), los jugadores controlan unidades (individuos o grupos) y estructuras, lo que les permite asegurar zonas del mapa y destruir los recursos de sus oponentes [7].

En este tipo de juegos, el jugador normalmente no puede cambiar el ángulo de la cámara para ver a mayor distancia. Esta restricción permite a los desarrolladores aplicar varias optimizaciones al motor de rendering.

Algunas de las características a nivel tecnológico de los juegos de estrategia en tiempo real son las siguientes [1]:

- Las unidades tienen relativamente pocos detalles a nivel gráfico. De esta forma el juego puede renderizar grandes cantidades de unidades simultáneamente.
- El escenario en el que tiene lugar el juego se suele representar con un heightmap. Se puede encontrar más información sobre heightmaps en [8].
- El jugador normalmente puede controlar sus unidades y construir nuevas estructuras sobre el terreno.

- La interacción del usuario es normalmente mediante clicks de ratón o selecciones de área, además de a través de menús y otros comandos.

1.3.3.6. Videojuegos multijugador masivos en línea

Un juego multijugador masivo en línea, también llamado MMOG (*Massively Multiplayer Online Game*) es un juego multijugador que es capaz de soportar un gran número de jugadores conectados simultáneamente (de miles a cientos de miles). La mayoría de estos juegos suelen tener un mundo persistente (existe durante largos períodos de tiempo, más allá de la sesión de un jugador), aunque no siempre tiene por qué ser así [9].

Los requisitos gráficos de este tipo de juegos suelen ser inferiores a los de otros juegos multijugador no masivos, debido a las grandes dimensiones del mundo virtual y la gran cantidad de jugadores que tienen que soportar simultáneamente.

1.3.3.7. Conclusiones

Existen multitud de géneros de videojuegos y cada uno tiene unos requisitos tecnológicos diferentes. En los apartados anteriores se han expuesto algunos de los más destacados, aunque hay muchos más.

Esto explica por qué tradicionalmente los motores de juegos se han creado pensando en un género de videojuegos concreto. Sin embargo, también hay un conjunto muy grande de tecnologías y requisitos que son comunes a todos los géneros de videojuegos, especialmente si el motor va destinado a una plataforma concreta.

Con el paso del tiempo, al disponer de hardware más potente, muchas de las optimizaciones que se han aplicado tradicionalmente no tienen un impacto tan grande en el rendimiento final, y en muchos casos son prescindibles. Esto hace que las diferencias que se podían encontrar entre motores pensados para géneros de videojuegos distintos debidas a optimizaciones para un género concreto comiencen a desaparecer, de forma que comienza a ser posible reutilizar el mismo motor de juegos para distintos géneros de videojuegos, e incluso para distintas plataformas hardware.

1.3.4. Arquitectura de un motor de juegos

Un motor de juegos, al igual que cualquier otro sistema software, suele estar diseñado utilizando una arquitectura basada en capas. Normalmente las capas superiores dependen de las capas inferiores, pero no al contrario.

A continuación, en la *Figura 1*, se puede observar de forma esquematizada la arquitectura de un motor de juegos. No todos los motores de juegos tienen todas las capas que se muestran en dicha figura. Existen motores de juegos más simples, que solo incluyen algunas de las capas que aparecen a continuación, y motores más complejos que incluyen alguna capa más aparte de las que se pueden ver en la *Figura 1*.

En la *Figura 1*, las capas correspondientes al motor son las que no están sombreadas. Como se puede apreciar, el motor de juegos es un sistema que se ejecuta sobre el

sistema operativo, hace uso de librerías o SDKs (*Software Development Kits*) externos y sirve de base para los subsistemas de juegos concretos.

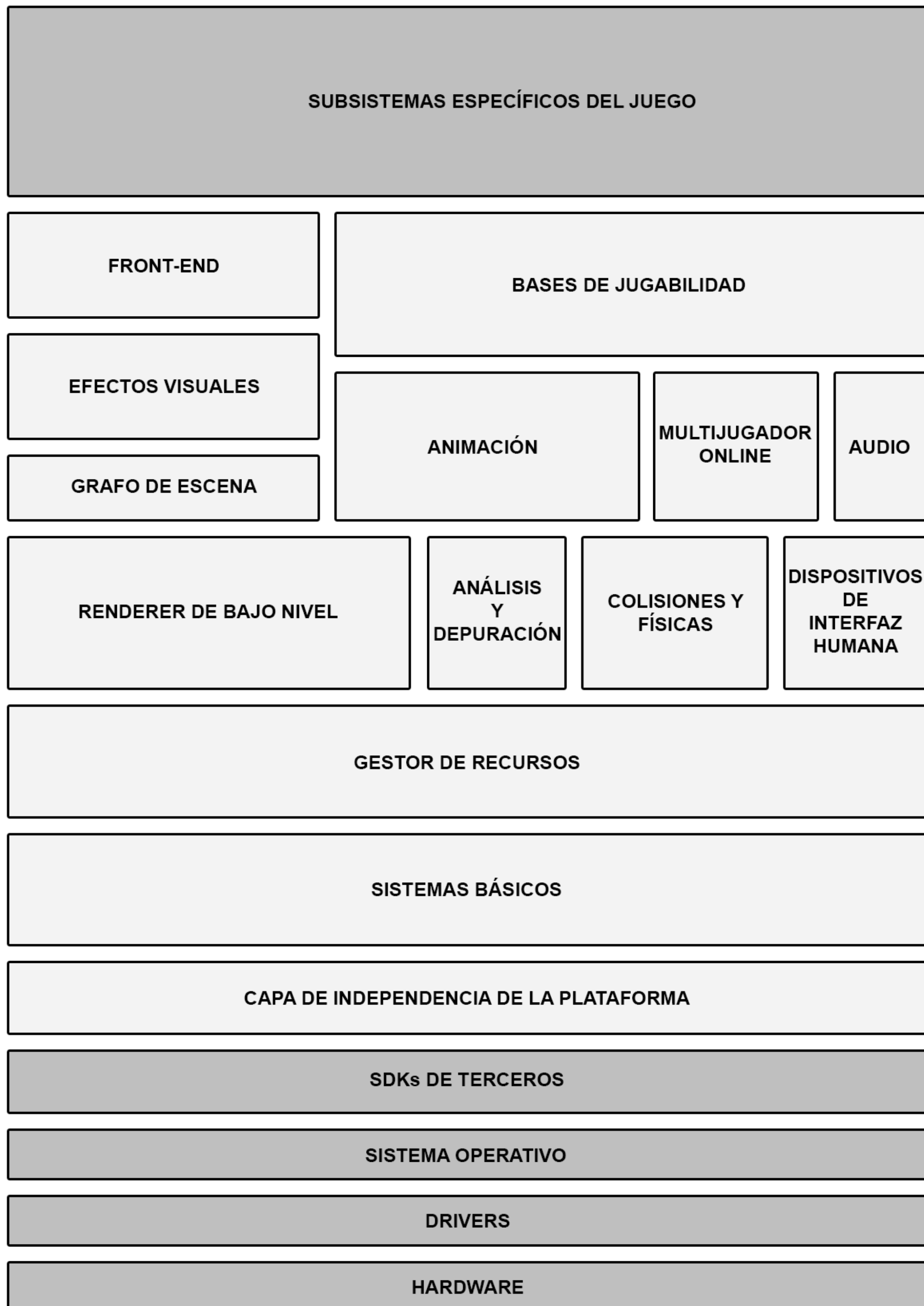

Figura 1. Arquitectura de un motor de juegos.

A continuación se explica en más detalle cada una de las capas que aparecen en la *Figura 1* [1], desde la capa de más abajo hacia arriba.

1.3.4.1. Hardware

La capa de hardware representa el dispositivo (ordenador, consola, smartphone, tablet…) en el que se ejecutará el videojuego.

1.3.4.2. Drivers

Los drivers, o controladores de dispositivo, son componentes software de bajo nivel que proporcionan una interfaz software para los dispositivos hardware, permitiendo al sistema operativo u otras aplicaciones acceder a funciones del hardware sin necesidad de tener que conocer el dispositivo hardware con el que se están comunicando. Este software generalmente lo proporcionan los vendedores de hardware, ya que los drivers son normalmente específicos para cada componente hardware.

1.3.4.3. Sistema operativo

El sistema operativo (*Operating System*, en inglés) es un software que tiene tres objetivos principales: proporcionar una capa de abstracción de los recursos hardware del sistema, gestionar el reparto de recursos del sistema (CPU, memoria física, espacio de disco, comunicaciones, etc.) entre las distintas aplicaciones que se ejecutan sobre el mismo, y proporcionar servicios al software de aplicación (semáforos, memoria compartida, etc.) [10].

Algunos ejemplos de sistemas operativos son Microsoft Windows, Mac OS, Linux, iOS y Android; aunque existen muchos más.

1.3.4.4. SDKs de terceros y middleware

La mayoría de motores de juegos hacen uso de SDKs (*Software Development Kits*) de terceros y middleware.

La interfaz pública proporcionada por un SDK se suele llamar API (*Application Programming Interface*).

Los suelen utilizarse librerías externas de todo tipo, algunos ejemplos son los siguientes [1]:

- Librerías de estructuras de datos y algoritmos: STL, Boost.
- Librerías gráficas: OpenGL, DirectX.
- Librerías de colisiones y simulación de físicas: Havok, PhysX, Edge.
- Librerías de animación de personajes: Granny, Havok Animation, Edge.
- Librerías de inteligencia artificial: Kynapse.

1.3.4.5. Capa de independencia de la plataforma

Muchos motores de juegos necesitan poder funcionar en más de una plataforma hardware [1]. Las grandes compañías de desarrollo de videojuegos normalmente desarrollan sus videojuegos para que sean compatibles con múltiples plataformas. De esta forma pueden hacer llegar sus productos a un mercado mucho más grande.

Normalmente los estudios de desarrollo de videojuegos que no lanzan sus productos en más de una plataforma hardware son los grandes estudios que tienen contratos de exclusividad con alguna plataforma en concreto, o bien estudios muy pequeños que no tienen recursos para lanzar sus productos en más de una plataforma. Aunque este último caso cada vez es menos común gracias a la existencia de motores de juegos multiplataforma a precios cada vez más asequibles.

El objetivo de la capa de independencia de la plataforma es principalmente actuar como interfaz entre el núcleo del motor de juegos y las capas inferiores. Esto facilita que el motor de juegos pueda ser compatible con otras plataformas.

Si un motor de juegos está diseñado de forma que incluya una capa de independencia de la plataforma, para hacer que sea compatible con una plataforma nueva, tan solo habría que modificar esta capa para que cuando el motor se ejecute en dicha plataforma utilice librerías compatibles con dicha plataforma y haga llamadas al sistema compatibles con la misma.

1.3.4.6. Sistemas básicos

Todo motor de juegos, y todos los sistemas software en general, necesitan un conjunto de utilidades sobre las que se construye el propio sistema. Estas utilidades son los sistemas básicos del motor.

Algunos ejemplos de sistemas básicos pueden ser [1]:

- Gestión de módulos.
- Gestión de memoria.
- Funciones matemáticas.
- Manejo de cadenas de caracteres.
- Utilidades de depuración y monitorización de rendimiento.
- Utilidades de logging.
- Parsers (XML, CSV, etc.).
- Configuración del motor.
- Carga asíncrona de archivos.

1.3.4.7. Gestor de recursos

El gestor de recursos proporciona una interfaz o conjunto de interfaces que permiten acceder a todos los recursos del juego [1]. Algunos motores de juegos, hacen esto de

forma centralizada, a través de una clase *ResourceManager*, mientras que otros son más abiertos en este sentido y permiten al programador utilizar un gestor de recursos propio o varios gestores para distintos tipos de recurso.

1.3.4.8. Renderer de bajo nivel

El renderer de bajo nivel es el subsistema del motor de juegos que se encarga de representar el juego en pantalla. Este subsistema trabaja con primitivas de bajo nivel y generalmente no tiene en cuenta si los elementos que se intentan representar están dentro del campo de visión de la cámara. Su objetivo principal es representar conjuntos de primitivas geométricas básicas lo más rápido posible [1].

Los componentes del renderer de bajo nivel cooperan para recolectar las primitivas geométricas (también llamados paquetes a renderizar) como mallas, listas de líneas, listas de puntos y cadenas de texto, entre otras muchas cosas, y renderizarlas tan rápido como sea posible.

El renderer de bajo nivel normalmente proporciona también una abstracción del *viewport* (rectángulo 2D sobre el que se proyecta la escena 3D hacia la posición de una cámara virtual [11]) mediante una matriz de proyección.

Este sistema también gestiona el estado del hardware de rendering y los *shaders* (programas que se encargan de aplicar colores y efectos a primitivas geométricas [12]) del juego mediante su sistema de materiales y el sistema de iluminación (en juegos 3D).

1.3.4.9. Grafo de escena

El renderer de bajo nivel renderiza toda la geometría que recibe, sin tener en cuenta si dicha geometría será visible o no.

En juegos en los que la escena está compuesta por pocos elementos, puede ser viable renderizar todos los elementos en pantalla sin que impacte en el rendimiento del juego. Sin embargo, cuando el mundo virtual es muy grande, es necesario aplicar una serie de optimizaciones que permitan renderizar solo los elementos indispensables.

Algunas de estas optimizaciones son la eliminación de caras traseras de los polígonos (*back-face culling*), eliminación de objetos que la cámara no puede ver (*frustum culling*), etc.

Si la escena está compuesta por pocos elementos, puede ser viable recorrer todos los elementos comprobando uno a uno si es visible o no, y renderizar solo los que son visibles. Por el contrario, si la escena es muy grande, hay que aplicar métodos de subdivisión espacial, o particionado del espacio, más avanzados.

A la subdivisión espacial a veces se le llama grafo de escena, aunque técnicamente el grafo de escena es una estructura de datos [1].

Los grafos de escena permiten saber rápidamente si un objeto es visible o no sin tener que recorrer todos los objetos de la escena

Algunas estructuras de datos que se suelen utilizar con este propósito son [13]:

- Árboles de partición binaria del espacio (*BSP trees*).
- Árboles cuaternarios (*Quadtrees*).
- Árboles octales (*Octrees*).
- Árboles KD (*K-D trees*).
- Jerarquías de volúmenes delimitadores (*Bounding Volume Hierarchies*).

Se puede encontrar información más detallada sobre este aspecto en [14].

1.3.4.10. Efectos visuales

Muchos motores de juegos modernos soportan una amplia gama de efectos visuales. Algunos de ellos son:

- Sistemas de partículas (para representar humo, salpicaduras, fuego, etc.).
- Light mapping.
- Sombras dinámicas.
- Efectos a pantalla completa (post-procesado). Estos efectos incluyen, desenfoques, resplandores, filtros de color, etc. Se puede encontrar más información sobre estos efectos en [15].

1.3.4.11. Front-End

La mayoría de juegos 3D utilizan gráficos 2D superpuestos en la escena 3D. Estos gráficos incluyen:

- El HUD (*Heads-Up Display*).
- Menús dentro del juego.
- Interfaz de usuario que permita al usuario configurar su personaje y tareas similares.

Estos gráficos constituyen el front-end del juego y se suelen renderizar en perspectiva ortográfica tras haber renderizado la escena 3D. Aunque también se pueden utilizar objetos 3D.

1.3.4.12. Análisis y depuración

Normalmente los motores de juegos incluyen algunas herramientas que permiten a los desarrolladores analizar el rendimiento de un videojuego.

Algunos ejemplos de estas herramientas son [1]:

- Herramientas de medición del tiempo de ejecución del código.
- Herramientas que permitan mostrar las estadísticas de depuración en pantalla mientras el juego se está ejecutando.

- Herramientas que permitan volcar los datos de depuración en un archivo de texto o una hora de cálculo para que puedan ser analizados posteriormente.
- Herramientas que permitan medir la cantidad de memoria que está siendo usada por el motor y por el juego.
- Herramientas que permitan almacenar los datos del uso de la memoria en archivos en cualquier momento.
- Herramientas que permitan mostrar mensajes de depuración durante la ejecución. Estas herramientas también deberían permitir al desarrollador clasificar los mensajes de depuración en distintas categorías y mostrar solo las que desee.

Las herramientas mencionadas son las más comunes, aunque hay muchas más y normalmente cada desarrollador crea herramientas propias para medir el rendimiento de su juego en caso de que las herramientas existentes no se adapten a sus necesidades.

1.3.4.13. Colisiones y físicas

La detección de colisiones es muy importante en la mayoría de los videojuegos. Si no, los objetos podrían atravesarse mutuamente y sería imposible interactuar con el mundo virtual de forma razonable [1].

Algunos videojuegos incluyen además simulaciones dinámicas realistas o semi-realistas. A esto en la industria del videojuego se le llama "sistema de físicas" aunque *mecánica de sólidos rígidos* sería un término más adecuado (más información sobre la *mecánica del sólido rígido* en [16]), ya que normalmente solo se simula el movimiento de cuerpos sólidos y las fuerzas que provocan que dicho movimiento ocurra.

Actualmente (Julio, 2014) muy pocas compañías de videojuegos producen su propio motor de físicas. En lugar de eso, normalmente se integra un SDK externo en el motor, o se omite la parte de físicas para que el desarrollador de videojuegos integre el motor que prefiera.

Algunos de los motores de físicas más populares son Havok y PhysX. Aunque hay otras alternativas de código abierto como ODE (*Open Dynamics Engine*).

También se pueden encontrar motores de físicas 2D. El más popular es Box2D.

1.3.4.14. Animación

La mayoría de juegos necesitan un sistema de animaciones, ya sea para representar personajes humanos, animales, o robots, entre otras cosas.

Se pueden distinguir varias técnicas de animación utilizadas en videojuegos [1]:

- Animación de sprites o texturas.
- Animación de jerarquías de cuerpos sólidos.
- *Skeletal animation.*
- Animación por vértice.

La técnica más utilizada a día de hoy, sobre todo en videojuegos 3D es la *Skeletal animation*. Consiste en definir un conjunto de huesos para un modelo 3D, de forma que cuando los huesos se muevan, los vértices del modelo 3D se mueven con ellos. Esta técnica también se puede aplicar a juegos 2D.

La animación de sprites o texturas también se utiliza bastante, sobre todo en videojuegos 2D.

El sistema de animación normalmente depende del sistema de físicas, puesto que las animaciones de los personajes se eligen en función de su movimiento y su interacción con los demás objetos del juego.

1.3.4.15. Dispositivos de interfaz humana

Los dispositivos de interfaz humana, también llamados HID (*Human Interface Devices*) son los que utiliza el usuario para comunicarse directamente con la máquina (ratón, teclado, joystick, pantalla táctil, etc.).

El componente HID de un motor de juegos se diseña de forma que abstraiga los detalles de bajo nivel de la comunicación del motor con el hardware. De esta forma el desarrollador de videojuegos no necesita preocuparse de cómo se lleva a cabo la comunicación con el hardware en una plataforma determinada [1].

Por ejemplo, en el caso de que un desarrollador de videojuegos necesite que el usuario utilice un mando para interactuar con el juego, tan solo debe tener que preocuparse de si un botón está pulsado o no. No tiene por qué tener que gestionar a bajo nivel la comunicación con el hardware.

1.3.4.16. Audio

En muchas ocasiones no se le da importancia al sistema de audio, pero la realidad es que es tan importante como el sistema de rendering. Un sistema de audio sofisticado puede hacer que un videojuego sea mucho mejor de lo que sería con un sistema de audio simple.

Un sistema de audio simple permite reproducir pistas de audio y efectos de sonido. Sin embargo, un motor de audio sofisticado engloba muchas más cosas. Por ejemplo, permite reproducir audio 3D, provocando en el jugador una sensación de inmersión mucho mayor.

1.3.4.17. Multijugador online

Con el objetivo de dar soporte a los juegos multijugador online, los motores de juegos suelen tener un componente que se encarga de gestionar conexiones, emparejamientos, replicación del estado del juego, etc.

1.3.4.18. Bases de jugabilidad

El término *jugabilidad*, o el término más utilizado comúnmente, *gameplay*, hacen referencia a las acciones que tienen lugar en el juego, las reglas que gobiernan el mundo

virtual, las habilidades de los personajes (mecánicas de los personajes), y los objetivos de los jugadores.

El gameplay normalmente se implementa en el mismo lenguaje en el que está implementado el motor, o bien utilizando un lenguaje de scripting.

La mayoría de motores de juegos incluyen una capa de abstracción que proporciona un conjunto de componentes sobre los que la lógica específica del juego se puede implementar.

Algunos de esos componentes pueden ser [1]:

- Un **modelo base del mundo del juego y modelos base de los objetos del juego**, que se utilizan para representar el mundo virtual y los objetos que lo componen.
- Un **sistema de eventos**, que se puede utilizar para que los objetos del juego se comuniquen entre ellos mediante envío de mensajes.
- Un **sistema de scripting**, que permita implementar la lógica del juego (gameplay) en un lenguaje de scripting. Normalmente los motores que incluyen soporte para scripting proporcionan una API que permite acceder a los componentes de bajo nivel del motor utilizando el lenguaje de scripting. Hay motores de juegos que utilizan como lenguaje de scripting JavaScript, Python, C#, etc. Mientras que otros utilizan un lenguaje de scripting propio, como es el caso de Unreal Engine, que utiliza UnrealScript.
- Unas **bases de inteligencia artificial**, que aunque tradicionalmente no se han considerado parte del motor de juegos por estar fuertemente ligada al juego, hay componentes que se pueden implementar de forma genérica y que se están comenzando a incluir en algunos motores de juegos. También se pueden utilizar motores de inteligencia artificial externos como Kynapse.

1.3.4.19. Subsistemas específicos del juego

Sobre la capa de bases de jugabilidad y el resto de componentes del motor de juegos, los programadores de gameplay y diseñadores de juegos cooperan para implementar el juego.

Los subsistemas del juego son diversos y normalmente específicos de cada juego, por lo que no tienen cabida dentro del motor de juegos.

Normalmente la capa de bases de jugabilidad es la última capa del motor de juegos y esta capa sería la capa correspondiente al juego en sí.

1.3.5. El bucle del juego

El bucle del juego, también conocido como *game loop*, es uno de los componentes más básicos de un videojuego y se encarga de llamar periódicamente a las funciones de actualización y renderizado del juego. Su principal objetivo es mantener una frecuencia de refresco que sea lo más constante posible. Los motores de juegos encapsulan la gestión del bucle del juego para que el programador de videojuegos no tenga que preocuparse de implementarlo.

1.3.5.1. Frecuencia de refresco

La frecuencia de refresco, o *frame rate*, es el número de veces que se actualiza el juego en una unidad de tiempo. La frecuencia de refresco de un videojuego se suele medir en FPS (*Frames Per Second*).

Normalmente los desarrolladores intentan que los videojuegos funcionen a 60 FPS, y en caso de no poder alcanzar dicho valor por limitaciones del dispositivo, intentan que los videojuegos funcionen a 30 FPS. Y, aunque también hay casos en los que se implementa una frecuencia de refresco variable, dichos casos son bastante menos comunes.

La principal razón por la que se intenta mantener la frecuencia a 60 FPS y no otro valor, es porque la mayoría de pantallas tienen una frecuencia de refresco de 60Hz y renderizar el juego a una frecuencia mayor a la frecuencia de refresco de la pantalla no tiene un impacto notable en la forma en la que el usuario percibe las animaciones y los movimientos de los objetos del juego.

Si en lugar de pensar en FPS se piensa en milisegundos, decir que un juego se actualiza a 30 FPS, equivale a decir que cada frame se procesa en aproximadamente 33ms; y en caso de que se actualice a 60 FPS, cada frame se procesa en aproximadamente 16ms.

La ventaja de que el videojuego se ejecute a 60 FPS es que las animaciones son más suaves. No obstante, muchos desarrolladores, sobre todo los que desarrollan para consolas, prefieren sacrificar frecuencia de refresco para poder permitirse utilizar gráficos de mayor calidad.

En la siguiente figura se puede ver la diferencia entre actualizar un elemento a 30 FPS y actualizarlo a 60 FPS. La figura representa las distintas posiciones que iría tomando un cuadrado que se desplaza a velocidad constante en el eje X durante un periodo de tiempo de 100ms.

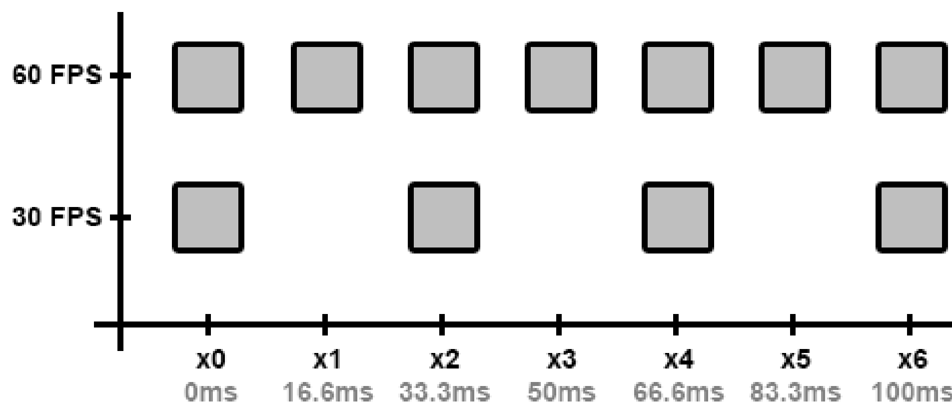

Figura 2. Desplazamiento en el eje X de un cuadrado a 30 y 60 FPS durante 100ms.

Como se puede observar en la figura anterior, las actualizaciones a 60 FPS aplican un desplazamiento menor en el eje X, pero son el doble de frecuentes. Esto hace que visualmente, el movimiento sea más fluido. A menor número de FPS, según la velocidad de desplazamiento, puede dar la sensación de que el objeto se desplaza a saltos.

1.3.5.2. Tipos de bucles de juego

En esta sección se explican algunos de los tipos de bucles de juego más utilizados en la actualidad (Julio, 2014) [17 y 18]. Concretamente el bucle simple, el de paso fijo, el de paso semi-fijo y, por último, una mejora del bucle de paso semi-fijo.

Existen otros tipos de bucle, como el de paso variable y otros bucles más avanzados que independizan la actualización de las físicas del juego. Se puede encontrar información adicional en la referencia [18].

1.3.5.2.1. Bucle de juego simple

Se podría decir que el bucle de juego más simple de todos es el que se ilustra en el siguiente diagrama de estados. Sin embargo este tipo de bucle tiene un gran problema, y es que las etapas de actualización y renderizado se ejecutarían tan frecuentemente como sea posible, por lo que si se ejecuta el juego en dos dispositivos distintos, en el más potente se actualizaría con mayor frecuencia. Por este motivo, no es recomendable utilizar este tipo de bucle.

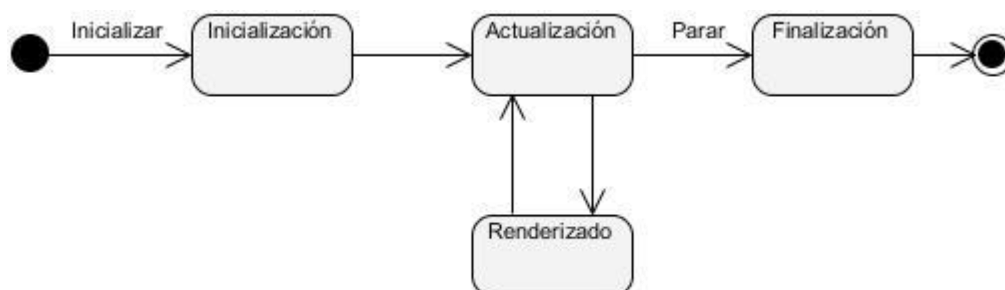

Figura 3. Diagrama de estados de un bucle de juego simple.

1.3.5.2.2. Bucle de juego de paso fijo

Para mejorar el bucle anterior surgen los bucles de paso fijo (*fixed time step*), que introducen un nuevo estado de espera. En este tipo de bucle, inicialmente se define el tiempo ideal debe dedicarse a cada frame (por ejemplo, 33ms para conseguir 30 FPS). Posteriormente, para cada ciclo de actualización y renderizado, se mide el tiempo consumido por dicho ciclo, y se calcula la diferencia entre ambos de la forma:

$$t_e = t_i - t_c$$

Donde t_e es el tiempo de espera, t_i es el tiempo ideal que debe tardar un ciclo de actualización y el renderizado, y t_c es el tiempo real que tarda un ciclo concreto de actualización y renderizado.

El tiempo de espera calculado es el tiempo que el hilo duerme antes de comenzar a actualizar el siguiente frame.

De esta forma se consigue que la frecuencia de actualización siempre sea la misma, suponiendo que el dispositivo en el que se ejecuta el juego sea capaz de mantener la frecuencia especificada.

En el siguiente diagrama de estados se puede observar el funcionamiento del bucle de paso fijo.

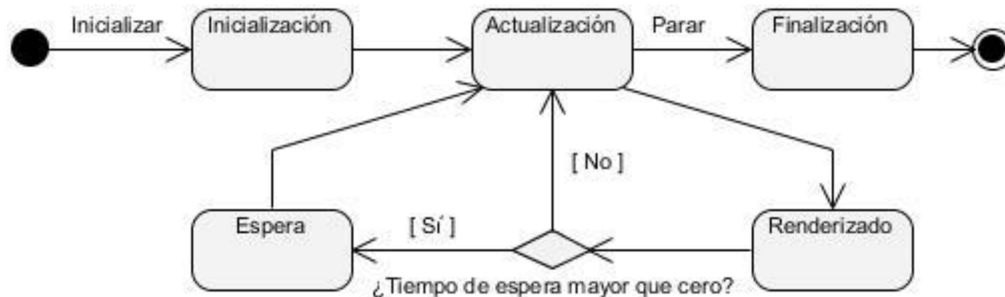

Figura 4. Diagrama de estados de un bucle de juego de paso fijo.

El principal beneficio que aporta este tipo de bucles es que la frecuencia de actualización depende del tiempo en lugar de depender de la potencia del dispositivo en el que se ejecuta el juego. Por otro lado, la principal desventaja es que si el dispositivo no es lo suficientemente potente como para mantener la frecuencia de refresco especificada, el juego funcionará más lento de lo esperado.

1.3.5.2.3. Bucle de juego de paso semi-fijo

El principal problema del bucle de paso fijo, como se ha comentado en el apartado anterior, surge cuando el dispositivo no es lo suficientemente potente como para mantener la frecuencia de actualización especificada, en cuyo caso el juego funcionaría más lento de lo esperado.

Esto se resuelve parcialmente con los bucles de paso semi-fijo (*semi-fixed time step*). Este tipo de bucle funciona de forma similar al bucle de paso fijo. La diferencia es que en caso que un frame tarde más tiempo de lo esperado, en lugar de retrasar la actualización del siguiente frame durante un ciclo completo, en este tipo de bucle es posible realizar varias actualizaciones seguidas sin renderizar. De esta forma el juego se

sigue actualizando a la frecuencia especificada, aunque se renderice a una frecuencia inferior.

En la siguiente figura se puede observar un ejemplo de funcionamiento de este tipo de bucle. Los bloques marcados con la letra 'A' representan actualizaciones, los bloques marcados con la letra 'R' representan renderizados y los bloques más oscuros representan tiempos de espera.

En el ejemplo se puede observar que el primer frame se ejecuta de forma normal, es decir, se lleva a cabo una actualización, un renderizado y el tiempo restante hasta el tiempo ideal por frame, el hilo duerme. En el segundo frame se puede observar que la etapa de renderizado tarda más de lo esperado, por lo que no hay tiempo de espera tras renderizar. Posteriormente, en el tercer frame se puede observar que, al haber superado el segundo frame el tiempo ideal por frame, tan solo se actualiza el juego y se introduce un tiempo de espera hasta completar el tiempo ideal por frame. Y por último, el cuarto frame se desarrolla de la forma esperada.

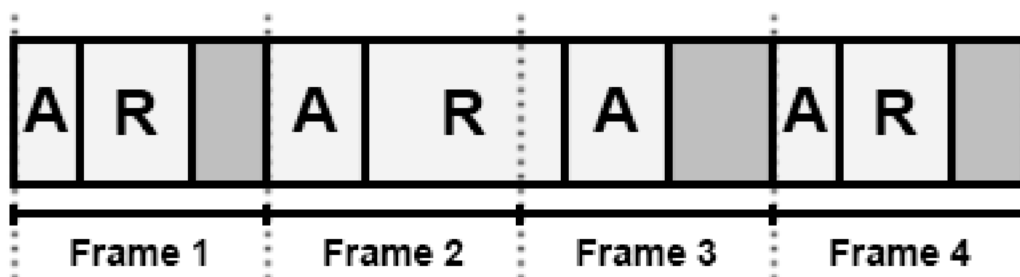

Figura 5. Ejemplo de funcionamiento de un bucle de paso semi-fijo.

En el siguiente diagrama de estados se puede observar el funcionamiento del bucle de paso semi-fijo.

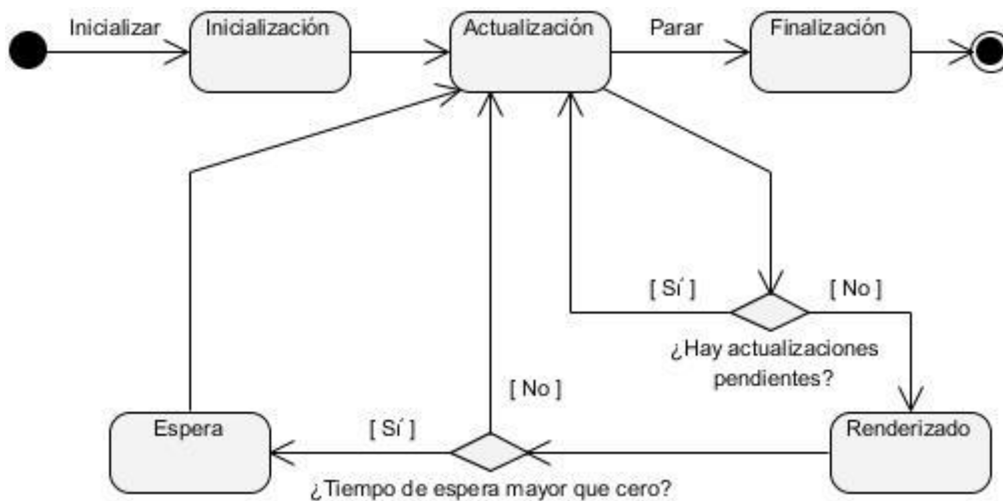

Figura 6. Diagrama de estados de un bucle de juego de paso semi-fijo.

1.3.5.2.4. Bucle de juego de paso semi-fijo mejorado

Un problema que puede acarrear el bucle de paso semi-fijo es que, si el juego está funcionando muy lento, no se renderice ningún frame. Esto se puede resolver fijando un límite para el número de frames que se pueden saltar sin renderizar.

En el siguiente diagrama de estados se puede observar el funcionamiento del bucle de paso semi-fijo con la mejora descrita en el párrafo anterior.

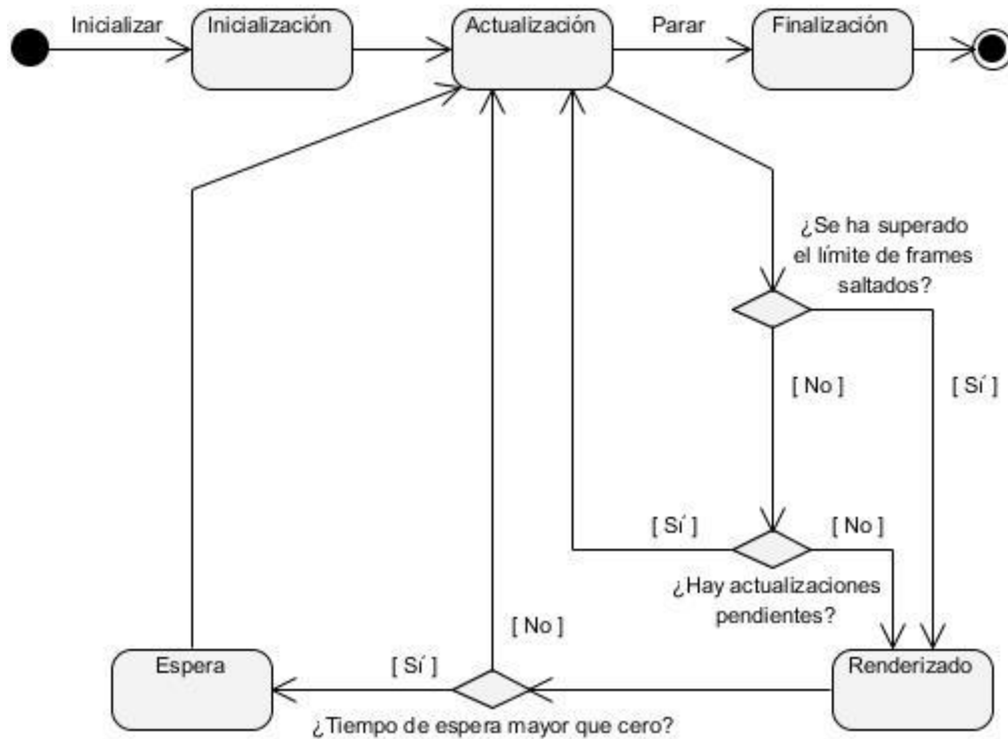

Figura 7. Diagrama de estados de un bucle de paso semi-fijo mejorado.

1.4. Componentes de una aplicación para Android

Tal y como queda reflejado en el título de este documento, la plataforma objetivo del proyecto desarrollado es Android. Por tanto, para facilitar el entendimiento de algunas de las explicaciones que se exponen en el capítulo 4, en este apartado se proporciona una visión general de los componentes que se pueden utilizar en una aplicación para Android.

Existen 4 tipos de componentes de aplicación [19]:

- Activities.
- Services.
- Content providers.
- Broadcast receivers.

A continuación se explican brevemente estos cuatro componentes, haciendo un poco más de hincapié en las activities y en su ciclo de vida, puesto que comprender su

funcionamiento es de vital importancia para entender algunas explicaciones expuestas en el capítulo 4.

1.4.1. Activities

Una activity es un componente que proporciona una pantalla con una interfaz gráfica con la que el usuario puede interactuar para hacer algo como enviar un correo electrónico, hacer una foto, o simplemente visualizar un mapa. A cada activity se le proporciona una ventana en la que dibujar su interfaz. La ventana de la activity puede ocupar toda la pantalla del dispositivo o bien ocupar solo una parte de la pantalla [19 y 20].

Una aplicación puede consistir en una o varias activities. Normalmente se especifica una activity de la aplicación como la activity "principal", que es la que se le presenta al usuario cuando ejecuta la aplicación.

Una activity puede a su vez lanzar otras activities para realizar otras acciones, por ejemplo, en una aplicación de correo electrónico, podría haber una activity que contiene una lista de correos recibidos y, al pulsar un elemento de la lista, se lanzaría otra activity en la que el usuario podría leer el correo seleccionado.

La aplicación mantiene internamente una pila de activities. Cada vez que una activity es lanzada, se inserta en la pila: la que hubiera antes en primer plano pasa a segundo plano y la nueva activity pasa a estar en primer plano. Y cuando la activity que está en primer plano se saca de la pila, se destruye y la siguiente activity de la pila pasa a primer plano. Si al destruirse una activity la pila queda vacía, se cierra la aplicación.

Con el objetivo de permitir ejecutar acciones cuando la activity cambia de un estado concreto a otro, se diseñó lo que se conoce como el ciclo de vida de la activity. Cuando la activity se lanza, se ejecutan, en orden, los métodos `onCreate()`, `onStart()` y `onResume()`. Llegado este punto, la activity está en primer plano, visible para el usuario. Si otra activity pasa a primer plano, se llama al método `onPause()` de la activity. En este momento pueden ocurrir 3 cosas: o el usuario vuelve a la activity, en cuyo caso se volvería a llamar a `onResume()`, o el sistema operativo necesita memoria y mata el proceso, en cuyo caso la activity debería crearse de nuevo, o la activity deja de estar visible, en cuyo caso se llama a `onStop()`.

Tras llamar a `onStop()`, la activity no es visible y se pueden dar 3 situaciones:

1. Si la activity se estaba destruyendo se llama a `onDestroy()` y finalmente se destruye.
2. Si la activity simplemente estaba en segundo plano porque no estaba en la cima de la pila de activities, al volver a la cima y, por tanto, a primer plano, se llama a `onRestart()` y posteriormente a `onStart()` y `onResume()`.
3. Si otra aplicación necesita memoria y no hay memoria suficiente disponible, el sistema operativo mata el proceso.

En el siguiente diagrama se puede ver de forma visual el ciclo de vida de la activity [20] explicado anteriormente.

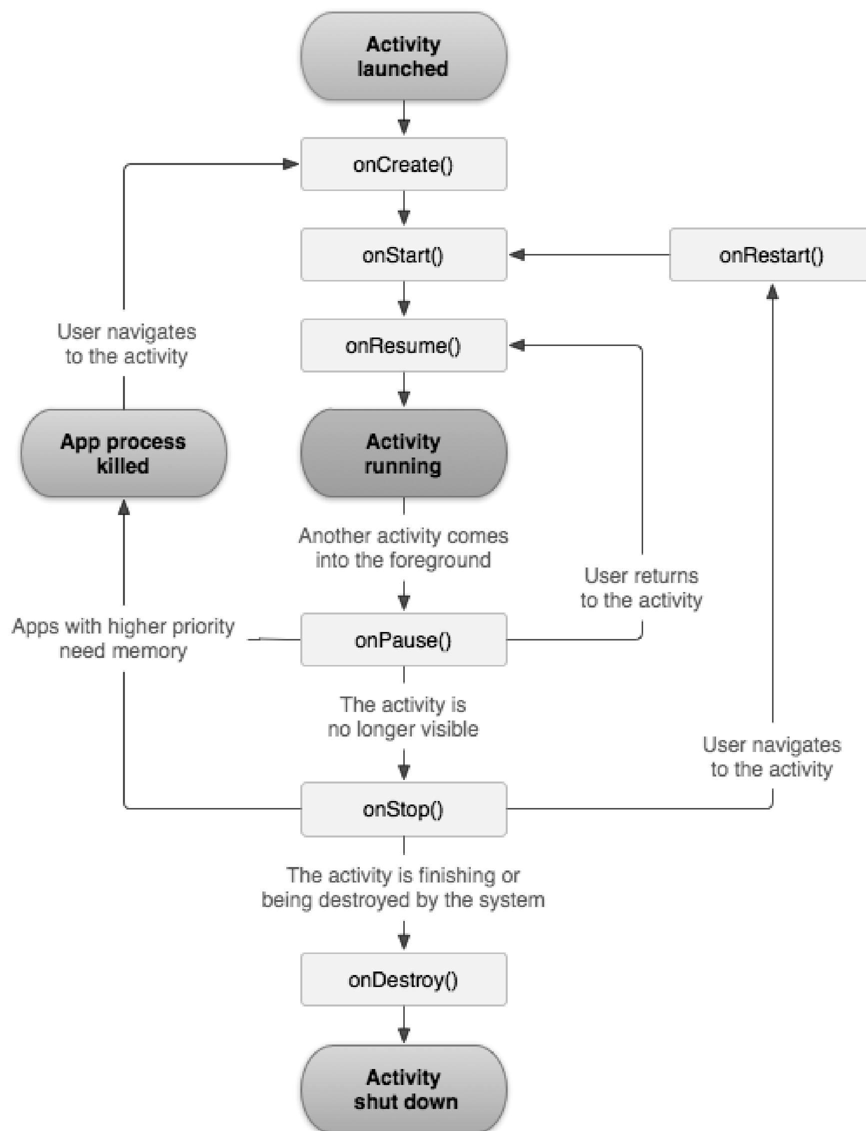

Figura 8. Ciclo de vida de la activity.

1.4.2. Services

Un service, o servicio, es un componente que puede llevar a cabo operaciones de larga duración en segundo plano. Los services no tienen interfaz de usuario [19 y 21].

Un service puede ser lanzado por otros componentes de aplicación, y seguirá ejecutándose en segundo plano aunque el usuario cambie de aplicación. Además, otro componente puede enlazarse al service para interactuar con él, e incluso establecer una comunicación entre procesos o IPC (*Inter-Process Communication*).

Algunos ejemplos de uso de servicios pueden ser gestionar comunicaciones de red, reproducir música, hacer operaciones de larga duración sobre archivos, etc. Todo esto en segundo plano.

1.4.3. Content providers

Los content providers, o proveedores de contenido, gestionan el acceso a un conjunto de datos estructurado. Encapsulan los datos y proporcionan mecanismos para asegurar la seguridad de los mismos [19 y 22].

Los content providers son la interfaz estándar que conecta los datos de un proceso con el código que se está ejecutando en otro proceso distinto. Los content providers se utilizan para acceder a los datos de una aplicación desde otras aplicaciones, e incluso pueden modificar dichos datos. Un ejemplo es el content provider que proporciona el propio sistema Android para obtener los contactos del dispositivo: cualquier aplicación, con los permisos adecuados, puede hacer consultas en la lista de contactos utilizando este content provider.

Además, los content providers se pueden utilizar para leer y escribir datos que sean privados de una aplicación. No siempre se utilizan para datos compartidos entre aplicaciones.

1.4.4. Broadcast receivers

Los broadcast receivers son componentes que permiten a una aplicación responder ante eventos del sistema [19]. Algunos ejemplos de estos eventos pueden ser: un evento que anuncia cuando se apaga la pantalla, un evento que anuncia que el nivel de la batería está bajo, un evento que anuncia cuando se arranca el sistema, etc.

Los broadcast receivers tampoco tienen interfaz de usuario, aunque es una práctica común crear una notificación en la barra de estado del sistema indicando al usuario que un evento concreto ha ocurrido. No obstante, no siempre es necesario mostrar una notificación cuando el broadcast receiver recibe un evento.

Estos componentes normalmente hacen una cantidad mínima de trabajo. Lo más común es que si la tarea que debe realizarse al recibir un evento determinado es una tarea pesada, se lance un servicio y se ejecute la tarea en dicho servicio. La misión principal de los broadcast receivers es permitir que la aplicación sea notificada de eventos del sistema.

1.5. Motivación

La decisión de desarrollar un motor de juegos 2D para Android se ve motivada principalmente por el interés del autor del proyecto por el desarrollo de videojuegos, y el desarrollo de tecnología relacionada con el desarrollo de videojuegos en general; además del interés por poner en práctica los conocimientos adquiridos durante la carrera.

La elección de Android como plataforma se debe por una parte al interés del autor por realizar un proyecto para plataformas móviles, entre las que destacan iOS y Android; y por otra parte, de estas 2 plataformas se ha escogido Android por 2 motivos: en primer lugar, el autor no dispone de ningún dispositivo Apple para desarrollar o hacer pruebas y, aunque actualmente (Julio, 2014) existen formas de desarrollar y probar aplicaciones para iOS en otros sistemas operativos, el proceso es más tedioso; y en segundo lugar, está más familiarizado con java que con Objective-C o Swift, por lo que comenzar a desarrollar para Android resulta más sencillo.

1.6. Objetivos generales del proyecto

A continuación se exponen, de forma resumida, los principales objetivos definidos para este proyecto. En el capítulo 4 se expondrán estos objetivos con mayor detalle y se explicará cómo se ha llevado a cabo cada uno de ellos.

1. Desarrollar un motor de juegos 2D para Android que posea, al menos, las características que se exponen a continuación.
    - Gestión automática del bucle del juego y sincronización entre el hilo de la lógica del juego y el hilo del renderer.
    - Integración del motor con el ciclo de vida de las aplicaciones Android.
    - Reproducción de efectos de sonido y pistas de audio.
    - Gestión de entrada del usuario: táctil, teclas físicas y acelerómetro.
    - Carga de recursos desde archivos: imágenes y pistas de audio.
    - Gestión de escenas, pantallas o estados de un juego.
    - Gestión de texturas y atlas de texturas.
    - Definición de animaciones.
    - Definición de materiales para especificar cómo se mostrará un elemento en pantalla (color, textura, etc.).
    - Implementación de un renderer basado en OpenGL ES 2.0 que permita:
        - Representación de rectángulos utilizando los materiales predefinidos o materiales implementados por cualquier usuario del motor.
        - Representación de texto.
        - Batch rendering (representación de múltiples elementos en una sola llamada a la función `glDrawElements` de OpenGL).
2. Elaborar documentación del motor en la que se explique el proceso de desarrollo del motor, así como las características del mismo, detallando especialmente las partes más complejas.
3. Desarrollar un conjunto de demos que muestren las principales funciones o características del motor y sirvan como ejemplos de uso del mismo, complementando a la documentación.

1.7. Planificación temporal

Este proyecto se ha realizado durante los años 2012-2014, mientras el autor se encontraba cursando 4º y 5º de Ingeniería Informática en la Universidad de Almería. La larga duración del proyecto se debe principalmente a que su desarrollo se ha tenido que compaginar con 4º y 5º de carrera. Por lo que ha habido meses en los que el proyecto ha estado totalmente parado.

A continuación se describe, desde el inicio, cómo se ha ido desarrollando el proyecto hasta llegar a su estado actual (Julio, 2014).

En Julio de 2012, el autor comenzó a interesarse por el desarrollo de videojuegos para Android y comenzó a investigar sobre motores de videojuegos existentes para dicha plataforma.

Posteriormente se interesó por el funcionamiento interno de un motor de juegos y comenzó a investigar sobre sus distintos componentes y cómo implementarlos, principalmente a modo de hobby.

A lo largo del verano de 2012 el autor creó un proyecto llamado GameEngine en el que desarrolló la base de lo que en la actualidad (Julio, 2014) es DroidEngine2D.

GameEngine comenzó como un conjunto de prototipos y experimentos en los que el autor iba poniendo en práctica los conocimientos adquiridos sobre el desarrollo de componentes de un motor de juegos.

Este proyecto estuvo en desarrollo desde Julio de 2012 hasta Octubre-Noviembre de 2012 y por esas fechas contaba con las siguientes características:

- Actualización del juego y rendering en hilos distintos. Se utilizaba un doble buffer para enviar los objetos desde el hilo del juego al hilo del renderer para que fuesen renderizados en pantalla.
- Renderer basado en OpenGL ES 1.0 que solo permitía renderizar rectángulos con textura (sprites).
- Gestión muy básica de la entrada de usuario.
- Reproducción de música y efectos de sonido.

En Noviembre de 2012, el autor comenzó a interesarse por aprender el funcionamiento básico de OpenGL ES 2.0, ya que la versión 1.0 estaba quedándose obsoleta y más del 90% de los dispositivos Android que había en el mercado por esas fechas ya soportaban la versión 2.0.

Entonces comenzó a leer sobre el funcionamiento de OpenGL ES 2.0 y poco tiempo después, se dispuso a poner en práctica los conocimientos adquiridos.

El código de GameEngine estaba muy desordenado muchas partes habría que reescribirlas desde cero al cambiar de versión de OpenGL ES. Por tanto decidió crear un proyecto nuevo en el que poder ir experimentando con OpenGL ES 2.0.

En Diciembre de 2012, el proyecto que había comenzado como un prototipo de renderer basado en OpenGL ES 2.0 tenía las mismas funcionalidades que el renderer de

GameEngine, e incluso algunas más, como por ejemplo la capacidad de renderizar rectángulos con rotación.

Sin embargo, aún había muchas partes que se podían mejorar y optimizar. El principal problema que tenía el renderer es que se enviaban los datos de la geometría a la GPU cada vez que se quería renderizar un rectángulo. Esto es altamente ineficiente porque el envío de geometría a la GPU es bastante costoso.

Una forma de ahorrar envíos de geometría a la GPU, y por tanto, de mejorar el rendimiento del motor gráfico es utilizar técnicas de *sprite batching* (empaquetado de sprites), que se explicará más adelante.

Durante Diciembre de 2012, el autor estuvo investigando sobre *sprite batching* y cómo implementarlo utilizando OpenGL ES 2.0. Y a principios de Enero de 2013 ya tenía dicha optimización implementada. Esta optimización permitía enviar hasta 32 rectángulos a la GPU en una sola llamada, reduciendo el número de envíos de geometría a la GPU drásticamente.

Desde Noviembre de 2012, GameEngine quedó bastante aparcado, aunque se fue actualizando de forma muy esporádica. Estas actualizaciones eran principalmente para arreglar fallos, ya que estaba siendo utilizado en AlarMind, un proyecto inicialmente desarrollado para la asignatura *Programación de Red*, impartida por Antonio Leopoldo Corral Liria en el curso 2012-2013.

En Febrero de 2013, se creó un nuevo proyecto, AndroidEngine, que pretendía reutilizar todo lo posible de GameEngine, sustituyendo todo el sistema de rendering por el sistema basado en OpenGL ES 2.0 implementado en los meses anteriores.

En AndroidEngine también se actualizaba el juego en un hilo y se renderizaba en otro hilo distinto, sin embargo se eliminó el doble buffer porque, por la forma en la que estaba diseñada la sincronización entre ambos hilos, no solo no era necesario, sino que provocaba una disminución en el rendimiento del motor.

La parte de reproducción de audio y la gestión de entrada de usuario permanecieron intactas al migrarlas al nuevo proyecto, mientras que la parte de rendering se reescribió por completo.

En Marzo de 2013, AndroidEngine era superior o igual en todos los aspectos a GameEngine, por lo que GameEngine dejó de usarse en AlarMind y comenzó a usarse AndroidEngine en su lugar. En este punto, GameEngine quedó abandonado por completo, mientras que AndroidEngine siguió desarrollándose.

Posteriormente el proyecto AndroidEngine se renombraría a DroidEngine2D.

Durante el mes de Marzo de 2013 se implementó la posibilidad de crear animaciones y se rediseñaron la forma de gestionar los eventos de entrada de usuario y la forma de inicializar el motor. Además se implementó un gestor de texturas y se arreglaron multitud de fallos.

En Abril y Mayo de 2013 se agregó la posibilidad de capturar valores del acelerómetro y se mejoró la gestión de entrada de usuario en general. Además se agregaron mecanismos de depuración de las llamadas a OpenGL y se mejoró el rendimiento del renderer.

En Julio y Agosto de 2013 se reestructuró por completo la gestión de shaders y se agregaron shaders que permitían renderizar conjuntos de rectángulos con textura y opacidad, con textura y color, con color únicamente y con textura únicamente. Además se agregó una clase Graphics, que funciona de forma similar a la clase Graphics de AWT/Swing y hace más cómodo el uso del motor.

En los meses posteriores, hasta Diciembre de 2013, se tradujo toda la documentación del código (javadoc) a inglés y se implementó un gestor de escenas, la posibilidad de renderizar texto utilizando OpenGL ES 2.0, y un sistema de materiales que facilitaba bastante el uso del motor.

Además se comenzó a desarrollar en paralelo un proyecto de demos técnicas del motor, que muestra las principales funcionalidades del motor de forma aislada y también contiene un minijuego de prueba que sirve de ejemplo de cómo utilizar DroidEngine2D para crear un videojuego 2D.

Por último, desde Enero de 2014 hasta el 15 de Marzo de 2014, se rediseñó por completo la gestión de estados del juego (previamente llamada gestión de escenas) y el procesamiento de la entrada de usuario. Anteriormente se procesaba de forma asíncrona, y tras ser rediseñado, los eventos siguen recibiéndose de forma asíncrona, pero se encolan y se procesan de forma síncrona, evitando posibles errores de sincronización por parte del usuario del motor de juegos.

Además se mejoraron las herramientas de depuración del renderer agregando un contador de llamadas a *glDrawElements* por frame.

En la última etapa del desarrollo, el autor se centró principalmente en las demos técnicas del motor, puesto que los objetivos propuestos en cuanto a características del motor en sí, ya se encontraban cubiertos.

Por último desde Abril de 2014 en adelante, el autor se centró en terminar las demos técnicas del motor y elaborar la documentación del mismo.

2

Estado del arte

2.1. Introducción

Actualmente (Julio, 2014) existen motores de juegos de muchos tipos. Algunas de las clasificaciones que se pueden hacer son las siguientes:

- Motores de código abierto o de código cerrado.
- Motores con licencias comerciales o gratuitas.
- Motores 2D o 3D.
- Motores con editor gráfico o sin editor gráfico.
- Motores monoplataforma o multiplataforma.

2.2. Revisión de varios motores de juegos

A continuación se hará un breve repaso de algunos de los motores de juegos más destacados en la actualidad (Julio, 2014), resumiendo brevemente sus principales características y las distintas licencias de uso que ofrece cada uno. Además se mencionarán algunos de los videojuegos desarrollados con cada uno de los distintos motores para que sea más sencillo hacerse una idea del tipo de trabajos que se pueden desarrollar con cada motor.

2.2.1. Unity

Unity es el motor de videojuegos multiplataforma que más auge ha tenido en los últimos años.

Posee su propio entorno integrado de desarrollo (IDE), que cuenta con un editor gráfico en el que se pueden diseñar los escenarios del juego.

Permite crear juegos 2D y 3D para multitud de plataformas. A continuación se expone una lista de las plataformas [23] que soporta este motor a la fecha de redacción de esta memoria:

- Windows.
- Windows Store.
- Mac.
- Linux.
- Unity Web Player.
- Facebook.
- iOS.
- Android.
- Windows Phone 8.
- BlackBerry.
- Wii U.
- PlayStation 3.
- PlayStation 4.
- PlayStation Vita.
- Xbox 360.
- Xbox One.

Una característica importante de Unity es su *Asset Store*. El Asset Store de Unity es una tienda online donde los desarrolladores pueden adquirir recursos (assets) o herramientas para sus juegos. Hay recursos y herramientas, tanto gratuitas como de pago.

Otras características que han contribuido a la expansión del uso de Unity en los últimos años son las siguientes [23]:

- Es un motor muy fácil de utilizar y tiene una curva de aprendizaje bastante rápida.
- Cuenta con una documentación bastante buena a la que se puede acceder de forma gratuita desde su web.
- Permite desarrollar de forma eficiente, ya que el entorno está diseñado para que su uso sea lo más intuitivo posible.

- El entorno es bastante flexible, lo que permite a cada desarrollador adaptarlo a su forma de trabajar.
- Permite desarrollar juegos rápidamente utilizando scripts para programar las mecánicas del juego. Los scripts se pueden escribir en JavaScript, C# o Boo.
- Existe una licencia gratuita, que permite crear juegos para Windows, Mac, Linux, Android, iOS, BlackBerry, web y Windows Store.
- Las licencias de pago se pueden obtener en 2 formatos, mediante pago único, o mediante suscripción mensual.

En cuanto a las licencias disponibles, podemos encontrar dos, *Unity Free* y *Unity Pro*.

La licencia *Unity Free* permite crear juegos para Windows, Mac, Linux, Android, iOS, BlackBerry, web y Windows Store.

Si se superan los 100.000$ de recaudación anuales con los juegos realizados utilizando la licencia *Unity Free*, es obligatorio adquirir la licencia *Unity Pro*.

La licencia *Unity Pro* agrega múltiples características nuevas y optimizaciones, permitiendo crear juegos más sofisticados.

Además, con la licencia Unity Pro es posible adquirir licencias individuales por plataforma. Por ejemplo, Android Pro, iOS Pro, etc. Y también es posible adquirir licencias para consolas, aunque para ello es necesario ponerse en contacto con Unity Technologies. Estas licencias no se pueden comprar a través de la web de Unity.

Algunos de los juegos desarrollados con Unity son: Temple Run 2, Forced, Deus Ex: The Fall, Gone Home, Wasteland 2, Oddworld: New 'n' Tasty y Dead Trigger 2.

2.2.2. Unreal Engine

Unreal Engine es un motor de juegos desarrollado por Epic Games. Se ofrece bajo suscripción mensual con restricciones mínimas.

Una vez suscrito, la licencia permite al desarrollador acceder al entorno de desarrollo y al código fuente del motor y las herramientas, que está escrito en C++.

Además tendrá acceso a algunos recursos, como por ejemplo, los siguientes [24]:

- Assets, como mapas por defecto, contenido de iniciación para construir niveles, etc.
- Plantillas para juegos, que se pueden utilizar como base para construir un juego del estilo que se desee.
- Juegos completos para PC y dispositivos móviles que sirven como ejemplos de juegos de estrategia y de disparos en primera persona realizados con el motor.
- Otros contenidos de demostración del motor, como demos de funcionalidades específicas, juegos pequeños, etc.

Los desarrolladores pueden realizar todas las modificaciones que deseen al código fuente del motor para que se adapte a sus necesidades. Los desarrolladores que modifiquen el código del motor podrán enviar una petición a Epic Games para que sus

cambios sean incluidos en el motor a través de GitHub. Sin embargo, no puede hacer público dicho código [24].

La suscripción de Unreal Engine permite a los desarrolladores crear juegos para las siguientes plataformas:

- Windows.
- Mac.
- iOS.
- Android.

También se puede utilizar para crear juegos para consolas, pero esa característica no se incluye en la suscripción. Para ello hay que ponerse en contacto con Epic Games para adquirir una licencia personalizada que permita desarrollar para consolas utilizando Unreal Engine.

Al igual que Unity, también posee una tienda virtual desde la que los desarrolladores pueden descargar plantillas para juegos, juegos de ejemplo, etc.

La licencia de uso de Unreal Engine especifica que los desarrolladores deberán pagarle a Epic Games el 5% de todos los ingresos que generen los juegos desarrollados utilizando Unreal Engine [24].

Algunos de las sagas de videojuegos más conocidas desarrolladas con Unreal Engine son: Unreal Tournament, Deus Ex, BioShock, Gears of War, Mass Effect, Borderlands y Dishonored.

2.2.3. UDK

UDK (*Unreal Development Kit*) es un kit de desarrollo que permite crear juegos utilizando Unreal Engine [25].

A diferencia de Unreal Engine, UDK no requiere suscripción, aunque este kit tiene varias limitaciones con respecto a Unreal Engine. Algunas de ellas son las siguientes:

- En la actualidad (Julio, 2014), UDK permite crear juegos únicamente en UnrealScript (lenguaje de scripting creado específicamente para Unreal Engine), aunque en versiones posteriores será posible utilizar C++.
- No permite acceder al código fuente del motor ni las herramientas.
- No se puede utilizar para desarrollar juegos para consolas.

Existen 2 tipos de licencia de UDK: gratuita y comercial [26].

La licencia gratuita solo permite utilizar UDK con fines educativos o para uso no comercial. Para uso comercial es necesario adquirir la licencia comercial de UDK. Esta licencia permite comercializar juegos creados con UDK sin restricciones hasta que éstos superen los 50.000$ en ingresos. Una vez superados estos ingresos, los desarrolladores deberán pagar a Epic Games un 25% de los ingresos generados desde dicho momento.

2.2.4. CryEngine

CryEngine es un motor de juegos desarrollado por Crytek que ha adquirido popularidad en los últimos años en el género de los videojuegos de disparos en primera persona.

Este motor comenzó siendo una demo de tecnología que realizaba Crytek para Nvidia. Pero posteriormente, cuando en Crytek se dieron cuenta del potencial del proyecto, convirtieron la demo en un juego (*Far Cry*), y crearon la primera versión de CryEngine [27].

CryEngine inicialmente solo permitía desarrollar juegos para PC. Pero desde la versión 3 (2009), permite crear juegos para PC, PlayStation 3 y Xbox 360.

En 2013, Crytek anunció que la cuarta generación de su motor soportaría además PlayStation 4, Xbox One y Wii U.

Al igual que Unity y Unreal Engine, CryEngine también cuenta con un editor gráfico. Además, cuenta con un modo WYSIWYP (*What You See Is What You Play*), que permite ejecutar el juego simultáneamente en PC, Xbox 360 y PlayStation 3, lo que facilita bastante el trabajo a los desarrolladores de juegos multiplataforma [28].

CryEngine se puede utilizar bajo 2 tipos de licencia: comercial y no comercial.

La licencia *Pro* permite comercializar los videojuegos desarrollados utilizando CryEngine y, dependiendo del caso, es posible incluso que Crytek colabore con el equipo de desarrollo para implementar en CryEngine nuevas características que pueda necesitar dicho equipo.

Por otro lado, la licencia *Free* (gratuita), solo puede utilizarse con fines educativos o con fines no comerciales.

Algunos de los juegos más conocidos creados con CryEngine son: Far Cry, Crysis, Aion: The tower of Eternity y Ryse: Son of Rome.

2.2.5. OGRE3D

OGRE (*Object-oriented Graphics Rendering Engine*) es un motor gráfico 3D de código abierto para PC, Mac, Linux, iOS y Android [29].

Está escrito en C++, pero en la actualidad (Julio, 2014) existen wrappers en Python (Python-Ogre), Java (Ogre4j) y .NET (MOGRE).

OGRE no es un motor de juegos propiamente dicho, sino un motor gráfico. Para crear un juego utilizando este motor, hay que combinarlo con otras librerías, ya que no incluye características como reproducción de audio, gestión de la entrada de usuario, detección de colisiones, etc.

A pesar de ello, este motor es bastante utilizado actualmente (Julio, 2014). Una de las ventajas que tiene respecto a otros motores más complejos es que, al ser de código abierto, es posible modificarlo para que se adapte a las necesidades de cada desarrollador. Además, al ser de código abierto puede servir como punto de partida para estudiar cómo se implementa un motor gráfico en C++.

Algunos de los juegos más conocidos que utilizan OGRE3D son: Torchlight, Torchlight II y Zombie Driver. Además, OGRE3D se utiliza como componente en motores de código abierto, como OGE (*Open Game Engine*).

2.2.6. Open Game Engine

Open Game Engine (OGE) es un proyecto de código abierto cuyo objetivo es crear un motor de videojuegos completo que incluya un motor de rendering, físicas, gestión de la entrada del usuario, detección de colisiones, scripting, etc. [30].

Todos los componentes y subsistemas utilizados en OGE están basados exclusivamente en librerías de código abierto. Y, en concreto, utiliza OGRE3D como motor gráfico [29].

Este motor además cuenta con un editor gráfico que permite diseñar niveles con mayor facilidad que utilizando OGRE3D de forma independiente.

Open Game Engine está disponible bajo 2 licencias. Por un lado, se puede obtener de forma gratuita bajo la licencia LGPL. Y por otro lado, también es posible obtener una licencia de pago que permite su uso con una única restricción, agregar una notificación visible para el usuario final que indique que el juego utiliza OGE.

2.2.7. LibGDX

LibGDX es un motor de juegos multiplataforma que permite desarrollar juegos 2D y 3D [31].

A continuación se puede ver una lista de todas las plataformas que soporta [32]:

- Windows.
- Linux.
- Mac OS X.
- Android (2.2+).
- BlackBerry.
- iOS.
- Java Applet.
- JavaScript / WebGL (Chrome, Safari, Opera, Firefox, Internet Explorer a través de Google Chrome Frame).

Este motor es de código abierto y está licenciado bajo la licencia Apache 2.0. Y, aunque el proyecto lo inició Mario Zechner, actualmente (Julio, 2014), no solo acepta, sino que anima a la comunidad a contribuir al mismo [31].

También cuenta con una documentación bastante detallada y un gran soporte por parte de la comunidad, además de una gran cantidad de ejemplos y juegos de código abierto desarrollados utilizando este motor. Esto hace que sea bastante sencillo comenzar a desarrollar videojuegos con libGDX.

El autor de libGDX (Mario Zechner) publicó un libro en 2011 titulado *Beginning Android Games*, en el que explica cómo crear videojuegos desde cero. En este libro podemos encontrar bastantes explicaciones de cómo están diseñados algunos de los componentes de libGDX, aunque en muchos ejemplos no se haga mención directa a la librería [33].

Además, existen otros libros que explican más detalladamente como desarrollar juegos multiplataforma utilizando libGDX. Por ejemplo, *Learning Libgdx Game Development*, escrito por Andreas Oehlke [34].

Uno de los juegos más populares desarrollado con libGDX es Ingress.

2.2.8. AndEngine

AndEngine es un motor de videojuegos 2D para Android basado en OpenGL y está desarrollado por Nicolas Gramlich [35].

Al igual que libGDX, está licenciado bajo la licencia Apache 2.0 y cuenta con bastante soporte por parte de la comunidad.

Actualmente (Julio, 2014) cuenta con 2 versiones principales: GLES1, que utiliza OpenGL ES 1.0 y GLES2, que utiliza OpenGL ES 2.0. La primera ha dejado de ser actualizada puesto que ya no se recomienda utilizar OpenGL ES 1.0, sin embargo hay multitud de tutoriales basados en esta versión que aún son de utilidad de cara a aprender a utilizar este motor.

AndEngine además cuenta con un conjunto de extensiones que permiten agregar funcionalidad adicional al núcleo del motor en función de las necesidades de los desarrolladores que hagan uso del mismo. Mediante estas extensiones se pueden agregar, entre otras, las siguientes características [36]:

- Realidad aumentada.
- Creación de fondos de escritorio animados.
- Multijugador.
- Multi-touch.
- Físicas 2D.
- Testing mediante Robotium.
- Scripting.
- Texturas en formato vectorial.
- Atlas de texturas de TexturePacker.
- Mapas en formato TMX.

Algunos juegos desarrollados con AndEngine son: Bunny Shooter, Cow and Pig Go Home y Greedy Spiders.

2.3. Conclusiones

En este capítulo se ha hecho un breve repaso de algunos de los motores de juegos que hay actualmente (Julio, 2014) en el mercado. Se han revisado motores muy conocidos y que son utilizados principalmente por grandes empresas, como Unreal Engine y CryEngine; motores muy accesibles y que son utilizados tanto por grandes empresas como por empresas pequeñas y aficionados, como Unity; y otros motores menos conocidos que los mencionados anteriormente, pero que también son bastante utilizados, como libGDX y AndEngine.

Como se puede deducir de la información expuesta en este capítulo, no existe un motor de juegos en el mercado que se adapte por completo a las necesidades de todos los desarrolladores.

Hay motores muy completos, como Unity, que permiten desarrollar prácticamente cualquier tipo de juego utilizando scripts, sin embargo no permite desarrollar en lenguajes de más bajo nivel como C/C++. Para los desarrolladores con esas necesidades existen motores como Unreal Engine y CryEngine. Por otro lado, si lo que se pretende es crear, por ejemplo, un juego muy simple en 2D para Android, es probable que utilizar motores como Unity no sea lo más recomendable. Para estos casos existen motores menos pesados, como libGDX y AndEngine.

Y otra opción que no se ha mencionado en este capítulo, es la posibilidad de que una empresa cree su propio motor de juegos, con las ventajas y desventajas que eso conlleva. Algunas empresas que utilizan sus propios motores de juegos son Naughty Dog, Kojima Productions, Rockstar Games y EA Digital Illusions CE (DICE).

3

OpenGL ES

3.1. Introducción

El sistema de rendering de DroidEngine2D está basado en OpenGL ES 2.0. En este capítulo se explica qué es OpenGL ES y las distintas versiones que existen actualmente (Julio, 2014).

Posteriormente se explicarán algunos conceptos básicos de OpenGL como los distintos sistemas de coordenadas con los que hay que tratar y los dos tipos de pipeline de rendering que utiliza OpenGL ES: el pipeline de rendering de función fija (OpenGL ES 1.X) y el pipeline de rendering programable (a partir de OpenGL ES 2.0).

Por último, se hará un estudio del soporte de cada una de las versiones de OpenGL ES en Android y se expondrán las principales razones por las que se ha optado por la utilización de OpenGL ES 2.0 en DroidEngine2D.

3.2. ¿Qué es OpenGL ES?

OpenGL ES (*OpenGL for Embedded Systems*) es una API gratuita y multiplataforma que permite representar gráficos 2D y 3D en sistemas embebidos, como consolas, smartphones y tablets. Es en un subconjunto de la API de OpenGL para escritorio.

OpenGL ES crea una interfaz de bajo nivel flexible y potente entre el software y el hardware. Incluye perfiles para sistemas de coma flotante y de coma fija, además de la especificación EGL, que permite utilizar OpenGL con los sistemas de ventanas nativos específicos de cada plataforma [37].

3.3. Versiones de OpenGL ES

Actualmente (Julio, 2014) están disponibles las siguientes versiones de OpenGL ES:

- OpenGL ES 1.0.
- OpenGL ES 1.1.
- OpenGL ES 2.0.
- OpenGL ES 3.0.
- OpenGL ES 3.1.

A continuación se exponen las características más destacadas de cada una de las versiones principales de OpenGL ES (1.X, 2.X, 3.X).

3.3.1. OpenGL ES 1.X

OpenGL ES 1.0 y OpenGL ES 1.1 son las versiones de OpenGL ES compatibles con las GPUs de pipeline fijo. Estas versiones utilizan por tanto un pipeline de rendering fijo.

La especificación de OpenGL ES 1.0 [38] fue creada a partir de la especificación de OpenGL 1.3, y se centra en activar el rendering vía software utilizando aceleración por hardware básica.

La especificación de OpenGL ES 1.1 [39] agrega, entre otras, las siguientes características [40]:

- Está definida a partir de OpenGL 1.5.
- Agrega soporte para buffer objects, que proporcionan un mecanismo que los clientes pueden utilizar para reservar, inicializar y renderizar directamente desde la memoria del servidor (GPU).
- Agrega generación automática de mipmaps.
- Mejora el procesamiento de texturas.
- Permite crear planos de recorte definidos por el usuario.
- Agrega point sprites, lo que permite representar partículas de forma más eficiente, sin tener que utilizar rectángulos.
- Agrega consultas estáticas y dinámicas del estado del contexto de OpenGL.
- Define un mecanismo que permite renderizar texturas escribiendo los píxeles de una o varias texturas en una región rectangular especificada de la pantalla.

Además, OpenGL ES 1.1 cuenta con un pack de extensiones formado por un conjunto de extensiones opcionales. El pack de extensiones agrega las siguientes extensiones:

- Cube maps para representar entornos más realistas.
- Capacidad para utilizar los colores de cualquier textura para la función de combinar texturas.

- Nuevo wrap mode: GL_MIRRORED_REPEAT.
- Extensiones de blending para que sea posible especificar los factores de blending RGB y alfa para las operaciones de blending que requieren factores fuente y destino y proporcionan una ecuación independiente para RGB y alfa.
- Extiende las funciones StencilOp para que soporten los modos INCR_WRAP y DECR_WRAP.
- Aumenta el tamaño mínimo de la paleta de matrices de 9 a 32, lo que permite reducir el número de llamadas a glDrawElements y glDrawArrays.
- Soporta FrameBuffer Objects (FBOs), que mejoran la gestión de la superficie de representación, permitiendo renderizar elementos en otros buffers que no sean los que le proporciona el sistema de ventanas al contexto de OpenGL.

3.3.2. OpenGL ES 2.X

OpenGL ES 2.0 está pensado para aprovechar las ventajas que ofrecen las GPUs de pipeline programable. Combina una versión de GLSL (*OpenGL Shading Language*) con la ya conocida API de OpenGL ES 1.1, a la que se le eliminan todas la funcionalidades relacionadas con el pipeline de funciones fijas y se le agregan funciones programables en su lugar [41].

Las novedades de la especificación de OpenGL ES 2.0 [42] con respecto a la especificación de OpenGL ES 1.1 son las siguientes:

- Está definida tomando como referencia la especificación de OpenGL 2.0.
- Utiliza GLSL ES (*OpenGL Shading Language for Embedded Systems*) 1.0 [43], cuya especificación está definida tomando como referencia la especificación de GLSL 1.20. Agrega las funciones básicas de shading utilizadas en OpenGL 2.0, pero adaptadas para sistemas embebidos.
- El pipeline programable sustituye al pipeline fijo de OpenGL ES 1.X.
- Agrega más opciones de precisión (las mismas que están disponibles en GLSL 1.20).
- Incluye soporte para FBOs (*Frame Buffer Objects*) sin necesidad de utilizar extensiones.
- No es 100% compatible hacia atrás con OpenGL ES 1.1, aunque las características eliminadas se pueden reemplazar con shaders (programas que normalmente se ejecutan en la GPU y se utilizan para aplicar transformaciones geométricas y colores a los píxeles de una imagen).

3.3.3. OpenGL ES 3.X

La especificación pública de OpenGL ES 3.0 [44] fue publicada en Agosto de 2012. Esta versión está basada en OpenGL 3.X y es compatible hacia atrás con OpenGL ES 2.0, lo que significa que todas las características de OpenGL ES 2.0 están disponibles en

OpenGL ES 3.0 y todo el código que dependa de OpenGL ES 2.0 debería seguir funcionando al actualizar a la versión 3.0.

OpenGL ES 3.0 incluye algunas características nuevas con respecto a OpenGL ES 2.0. Algunas de las más destacadas son las siguientes:

- Soporte para instanciación de geometría, lo que permite renderizar múltiples copias de la misma malla en la escena en una sola llamada [45].
- Soporte para 4 o más render targets. Un render target es una característica de las GPUs que permite que una escena 3D sea renderizada sobre un buffer intermedio en lugar de renderizar directamente sobre el frame buffer o el back buffer. Los render targets se pueden utilizar para aplicar efectos adicionales sobre una imagen antes de mostrarla (post-procesamiento) [46].
- Formatos de compresión de texturas estándar (ETC2 / EAC), lo que permite eliminar la necesidad de tener que proporcionar las texturas en varios formatos para que la aplicación decida cual usar en función de la plataforma.
- Soporta GLSL ES 3.00 [47], que es un subconjunto de GLSL 3.3.
- Mejoras sustanciales en el mapeado de texturas y en los formatos de textura aceptados.
- Agrega más tipos de buffer object como los Uniform Buffer Objects y los Pixel Buffer Objects.

Por otro lado, la especificación de OpenGL ES 3.1 [48] fue publicada en Marzo de 2014 y agrega principalmente las siguientes características:

- Soporta GLSL ES 3.10 [49].
- Agrega compute shaders, o shaders de cómputo, que permiten ejecutar programas no relacionados con rendering en la GPU.
- Permite compilar vertex y fragment shaders de forma independiente.
- Permite que la GPU dibuje objetos que se encuentran definidos en un buffer en la memoria de la GPU en lugar de estar en la memoria de la CPU (*indirect drawing*). Esto permite que los compute shaders creen objetos directamente en la memoria de la GPU, reduciendo la frecuencia de comunicación con la CPU.

3.4. Sistemas de coordenadas en OpenGL

En el contexto de los gráficos 3D se utilizan múltiples sistemas de coordenadas durante el proceso de representación de objetos.

Normalmente, para representar un objeto, primero hay que definir las posiciones de sus vértices en un sistema de corrdenadas, y desde la definición de las posiciones de los vértices del objeto hasta que el objeto se muestra en pantalla, dichos vértices se van transformando de un sistema de coordenadas a otro hasta, finalmente, quedar definidos en el sistema de coordenadas de la ventana en la que se muestra la escena 3D.

Al trabajar con OpenGL en concreto se utilizan 6 sistemas de coordenadas distintos [50, 51 y 52], aunque con algunos de ellos no se trabaja de forma directa, sino que los utiliza OpenGL internamente.

- Coordenadas de modelado o coordenadas del objeto (*object coordinates*).
- Coordenadas mundiales (*world coordinates*).
- Coordenadas de visualización o coordenadas de cámara (*eye coordinates*).
- Coordenadas de proyección (*clip coordinates*).
- Coordenadas normalizadas de dispositivo (*normalized device coordinates*).
- Coordenadas de ventana (*window coordinates*).

En el siguiente diagrama se puede ver el orden en el que se utilizan estos sistemas de coordenadas en OpenGL. A continuación se explica el proceso en más detalle.

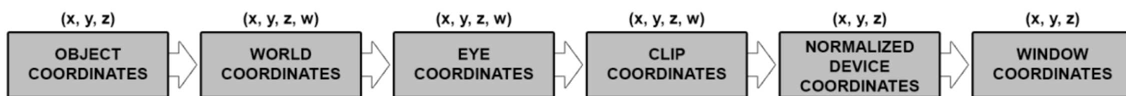

Figura 9. Orden de las transformaciones entre sistemas de coordenadas en OpenGL.

3.4.1. Object Coordinates

Este es el sistema de coordenadas local de cada objeto tridimensional. Siempre que se modela un objeto, se hace en este espacio de coordenadas. De ahí que estas coordenadas también se conozcan como coordenadas de modelado.

En este espacio de coordenadas cada vértice se define mediante 3 componentes (x, y, z), que se definen utilizando la unidad de medida del objeto (metros, pulgadas, etc.).

Por ejemplo, en la siguiente imagen se puede ver cómo se definirían los vértices de un cubo de 2x2x2 metros en este sistema de coordenadas. El cubo del ejemplo tiene su centro en el punto (0, 0, 0).

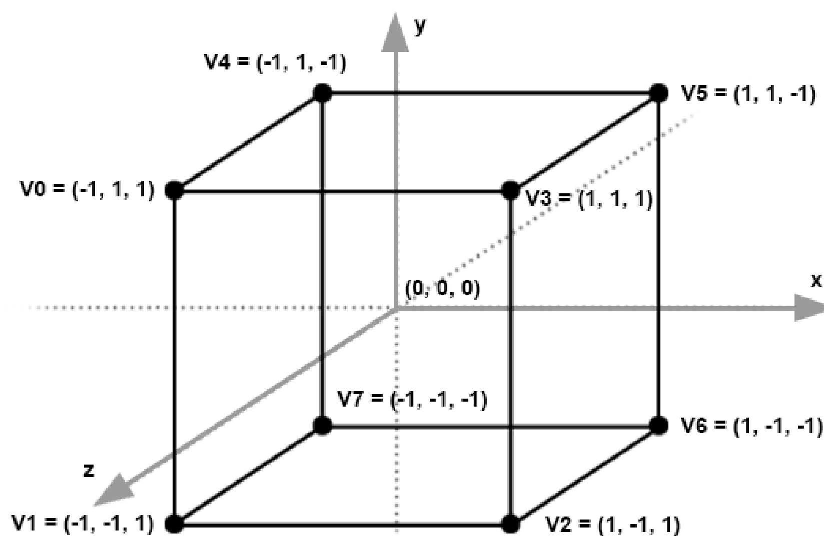

Figura 10. Cubo cuyos vértices están definidos utilizando el sistema de coordenadas de modelado.

Los vértices de los objetos se definen inicialmente en este sistema de coordenadas. Y posteriormente se transforman a coordenadas mundiales.

Las coordenadas mundiales, como se explica en el siguiente apartado, son coordenadas homogéneas, por tanto, antes de poder pasar las coordenadas de modelado a coordenadas mundiales, hay que hacer que las coordenadas de modelado también sean homogéneas. Para ello simplemente se añade una nueva coordenada *w* con valor 1.0. Las coordenadas de modelado homogéneas siempre son de la forma (x, y, z, 1.0).

3.4.2. World Coordinates

Este sistema de coordenadas también se conoce como coordenadas mundiales o coordenadas de escena y se utiliza para posicionar objetos en el mundo virtual. Para ello se aplican varias transformaciones a los vértices de los objetos (previamente definidos en coordenadas de modelado homogéneas). Hay 3 tipos de transformaciones: traslación, rotación y escalado. Y cualquier objeto se puede posicionar en la escena de la forma que se desee combinando estas transformaciones.

En este sistema de coordenadas, a diferencia del anterior, el punto (0, 0, 0, 0) no es el origen del objeto, sino el origen del mundo virtual o escena.

En el ejemplo de la siguiente imagen podemos observar 2 cubos que previamente se habían definido en coordenadas de modelado homogéneas, tal y como se explica en el apartado anterior, a los que se les ha aplicado una serie de transformaciones para que queden representados en coordenadas mundiales.

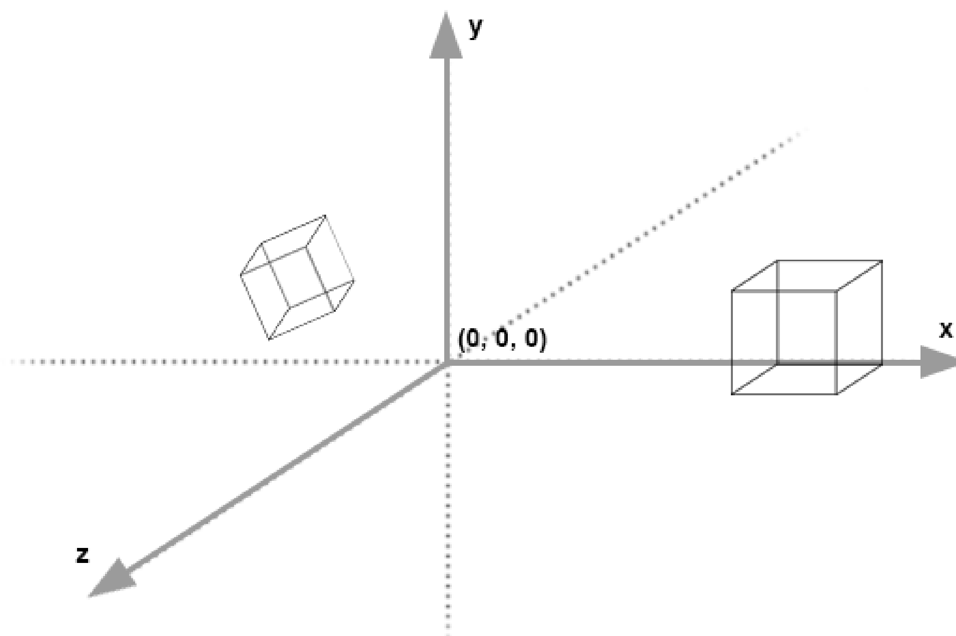

Figura 11. Cubos en coordenadas mundiales.

Con el objetivo de simplificar la explicación, se utilizarán las siguientes funciones de ejemplo:

```
traslacion(eje, distancia)
rotacion(eje, angulo)
escalaXYZ(factor)
```

Las transformaciones aplicadas al cubo que aparece a la derecha podrían ser las siguientes:

```
escalaXYZ(10)→traslacion(X, 30)
```

Las transformaciones aplicadas al cubo que aparece a la izquierda podrían ser las siguientes:

```
escalaXYZ(5)→rotacion(Y, 5)→rotacion(X, 45)→traslacion(Y, 20)
```

Todas estas transformaciones se realizan mediante productos de matrices. Normalmente se define una matriz de transformación homogénea que se utiliza para aplicar las transformaciones a todos los vértices del objeto. La matriz de transformación homogénea se obtiene al multiplicar las matrices de transformación individuales (matriz de traslación, matriz de rotación y matriz de escalado) en el orden en el que se desee aplicar las transformaciones. El orden es importante ya que el producto de matrices no es conmutativo.

Lo que se mencionaba en el apartado anterior sobre que las coordenadas de modelado debían ser coordenadas homogéneas se debe precisamente a que de esta forma se pueden multiplicar los vértices (vectores de 4 elementos) por la matriz de transformación homogénea, para pasar el objeto al sistema de coordenadas mundiales o de escena.

En las primeras versiones de OpenGL, la propia librería se encargaba de gestionar las matrices y para aplicar las transformaciones siempre había que seleccionar primero la matriz con la que se pretendía trabajar y posteriormente se llamaba a la función *glMulMatrix*()*, que acepta una matriz por parámetro y la multiplica por la matriz previamente seleccionada. Lo que hay que tener en cuenta al trabajar con OpenGL es que si, por ejemplo, la matriz seleccionada es la matriz C, y se hace una llamada a glMulMatrixf(M), internamente la operación que se realiza es C = C * M. Por tanto, las transformaciones hay que aplicarlas en orden inverso al orden natural.

Por ejemplo, anteriormente se ha mencionado que las transformaciones aplicadas al cubo que aparece a la izquierda en la imagen anterior podrían ser las siguientes:

```
escalaXYZ(5)→rotacion(Y, 5)→rotacion(X, 45)→traslacion(Y, 20)
```

En OpenGL se aplicarían en el siguiente orden:

```
traslacion(Y, 20)→rotacion(X, 45)→rotacion(Y, 5)→escalaXYZ(5)
```

Desde que se introdujo el pipeline de rendering programable, las matrices no las gestiona OpenGL internamente, sino que debe gestionarlas el programador a mano, por lo que es posible implementar funciones que hagan la multiplicación de matrices en el orden natural, aunque no se suele hacer, ya que quien trabaja habitualmente con

OpenGL está acostumbrado a aplicar las transformaciones en orden inverso y al cambiarlo podría generar confusión.

Por último, mencionar que la matriz de transformación homogénea utilizada para transformar los vértices de coordenadas de modelado a coordenadas mundiales normalmente se suele llamar “Model Matrix”. Esta matriz en las primeras versiones de OpenGL no se puede manipular directamente, sino que se trabaja con `GL_MODELVIEW`, que es la combinación de la “Model Matrix” y la “View Matrix” (M_{view} * M_{model}), la cual se explica más detalladamente en el siguiente apartado.

3.4.3. Eye Coordinates

Este sistema de coordenadas también se conoce como coordenadas de visualización o coordenadas de cámara. Y como su propio nombre indica, representa el espacio de coordenadas de la cámara.

Una vez que el objeto está posicionado en la escena, en coordenadas mundiales, para transformarlo al sistema de coordenadas de la cámara, tan solo hay que multiplicar todos los vértices por la “Model View Matrix”, que se calcula multiplicando la “View Matrix” por la “Model Matrix” (M_{view} * M_{model}). Esta es la matriz `GL_MODELVIEW` que se utiliza en OpenGL ES 1.X.

De esta forma, la escena quedará como si se estuviese mirando a través de una cámara. En este espacio de coordenadas, los vértices están posicionados relativos a la cámara.

En OpenGL la cámara se define mediante tres vectores.

- **Eye vector:** Indica la posición en la que se encuentra la cámara.
- **Center vector:** Indica la dirección hacia donde apunta la cámara.
- **Up vector:** Indica la orientación de la cámara.

La “View Matrix” se puede definir mediante la función *gluLookAt()* en las primeras versiones de OpenGL. Desde que se introdujo el pipeline de rendering programable (OpenGL ES 2.0), es necesario utilizar librerías externas o definir la matriz manualmente.

3.4.4. Clip Coordinates

Hasta ahora se han explicado todas las transformaciones necesarias para posicionar los vértices de los objetos en la escena de la forma en la que se desean ver (desde el punto de vista de la cámara). Sin embargo, los objetos deben verse en una pantalla 2D, por lo que es necesario proyectar las coordenadas de los vértices en un espacio de coordenadas tridimensional sobre una superficie bidimensional. Para ello hay que transformar las coordenadas de cámara explicadas en el apartado anterior a coordenadas de proyección o clip coordinates. Normalmente se elige uno de los dos tipos de proyección siguientes, aunque existen más:

- Proyección ortográfica.
- Proyección en perspectiva.

Para realizar la transformación, se define la matriz de proyección o "Projection Matrix". Esta matriz se define mediante las funciones *glOrtho()* o *glFrustum()*, dependiendo si se desea una proyección ortográfica o en perspectiva respectivamente.

Ambas funciones tienen los mismos parámetros, y son los siguientes:

- **Left:** Coordenadas del plano de recorte vertical izquierdo.
- **Right:** Coordenadas del plano de recorte vertical derecho.
- **Bottom:** Coordenadas del plano de recorte horizontal inferior.
- **Top:** Coordenadas del plano de recorte horizontal superior.
- **Near:** Distancia desde la cámara al plano de recorte frontal.
- **Far:** Distancia desde la cámara al plano de recorte trasero.

Todos estos planos de recorte mencionados forman el volumen de recorte, que se puede ver claramente en las siguientes figuras. Todos los vértices que se encuentren fuera del volumen de recorte no se verán en pantalla. Además, como se puede apreciar, el sistema de coordenadas está centrado en la cámara y es un sistema de coordenadas levógiro.

En la siguiente figura se puede ver el volumen de recorte de una escena cuando se utiliza la proyección ortográfica.

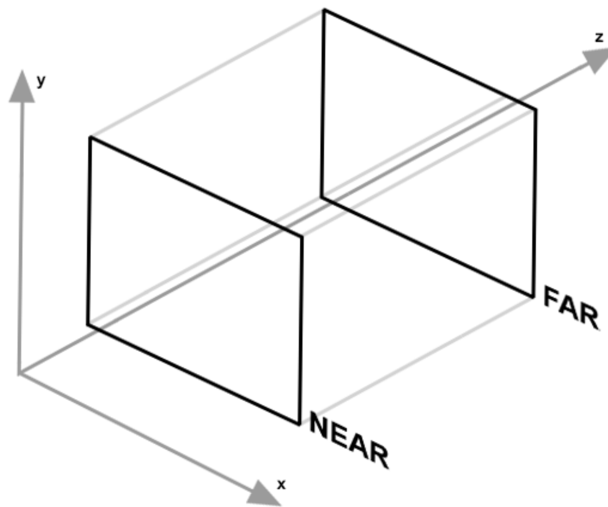

Figura 12. Proyección ortográfica.

En la siguiente figura se puede ver el volumen de recorte de una escena cuando se utiliza la proyección en perspectiva.

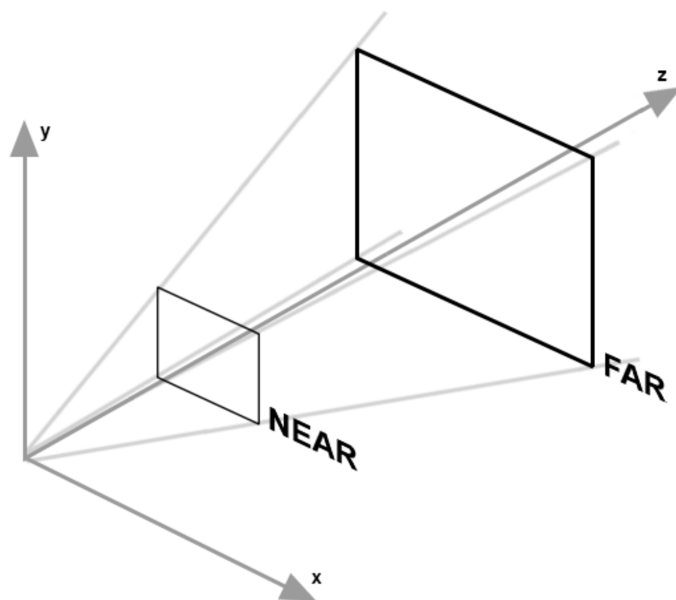

Figura 13. Proyección en perspectiva.

Tras definir la matriz de proyección, el siguiente paso es calcular la "Model View Projection Matrix", comúnmente abreviada como "MVP Matrix". Esta matriz se obtiene multiplicando la matriz de proyección por la "Model View Matrix" de la siguiente forma:

$$MVP = M_{projection} * M_{view} * M_{model}$$

Tras obtener esta matriz, basta con multiplicar los vértices de un objeto, definidos en coordenadas de modelado homogéneas, por la misma para pasar el objeto de coordenadas de modelado a coordenadas de proyección.

3.4.5. Normalized Device Coordinates

Una vez que se han calculado las coordenadas de proyección, se realiza la división de perspectiva para transformarlas a coordenadas normalizadas de dispositivo, que se encuentran en el rango [-1.0, 1.0].

Recordando lo explicado anteriormente, un punto en el sistema de coordenadas de proyección se especifica con 4 componentes (x, y, z, w).

La división de perspectiva consiste en lo siguiente:

$$(x_1, y_1, z_1) = ((x_0 / w), (y_0 / w), (z_0 / w))$$

Donde (x_1, y_1, z_1) es el punto original (x_0, y_0, z_0), representado en coordenadas normalizadas de dispositivo.

Esta operación la realiza OpenGL internamente, el programador no necesita preocuparse de realizarla de forma manual.

3.4.6. Window Coordinates

Por último, para que la escena pueda ser representada en pantalla, los objetos que la componen deben estar representados en el sistema de coordenadas de ventana. En este sistema de coordenadas, la unidad de medida son los píxeles y los puntos tienen 3 coordenadas (x, y, z), siendo (x, y) la posición en la ventana y *z* un valor entre 0.0 y 1.0 que representa la profundidad.

Para que OpenGL pueda transformar las coordenadas normalizadas de dispositivo a coordenadas de ventana, es necesario especificar el viewport (sección rectangular de la pantalla sobre la que se mostrará la escena) [11]. Para ello se utiliza la función *glViewPort()*, que tiene los siguientes parámetros:

- **x:** Coordenada X de la esquina inferior izquierda del viewport (en píxels).
- **y:** Coordenada Y de la esquina inferior izquierda del viewport (en píxels).
- **w:** Anchura del viewport (en píxels).
- **h:** Altura del viewport (en píxels).

3.5. Pipelines de rendering en OpenGL ES

Tal y como se ha mencionado en apartados anteriores, OpenGL ES 1.X implementa un pipeline de rendering de función fija, mientras que OpenGL ES 2.X y OpenGL ES 3.X implementan un pipeline de rendering programable. En esta sección se explican ambos pipelines de rendering.

La información expuesta en esta sección, excepto aquellas partes en las que se indique explícitamente, se ha extraído y sintetizado de las referencias bibliográficas [41], [52] y [53].

3.5.1. Pipeline de rendering de función fija

El pipeline de rendering de función fija se llama así debido al hecho de que todas las funciones que realiza OpenGL son fijas y no se pueden modificar. El programador no tiene control total sobre todos los aspectos del pipeline de rendering.

El siguiente diagrama muestra las distintas etapas del pipeline de función fija. Los nodos más oscuros representan las etapas del pipeline que se mantuvieron cuando surgió el pipeline programable, tal y como se explicará más adelante.

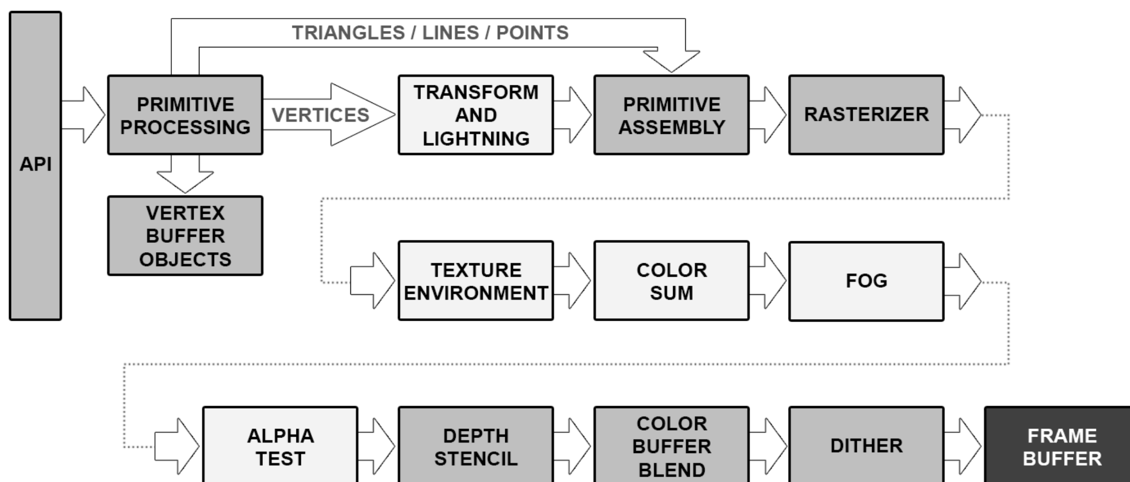

Figura 14. Pipeline de rendering de OpenGL ES 1.X.

A continuación se explica más detalladamente el diagrama anterior.

3.5.1.1. API

Esta caja representa a la interfaz que utiliza el programador para comunicarse con la GPU.

La primera etapa del rendering consiste en enviar datos de los vértices a la GPU. Los datos de los vértices representan los objetos 2D o 3D que se pretende renderizar.

Además de las posiciones de los vértices en el espacio, es posible enviar otros atributos, como colores, vectores normales, coordenadas de textura, y coordenadas de niebla. La lista de vértices con todos sus atributos que se envían a la GPU se llama normalmente *vertex stream*.

3.5.1.2. Primitive Processing

En esta etapa, los datos de los vértices se procesan para cada primitiva por separado. Las primitivas definen cómo debe interpretar OpenGL el vertex stream [54]. OpenGL ES 1.X soporta tres tipos de primitivas: puntos, líneas y triángulos.

Puntos

Para que OpenGL ES represente el vertex stream como puntos hay que utilizar la constante GL_POINTS. Los point sprites son cuadrados alineados a los ejes de la pantalla que se definen mediante una posición y un radio. Normalmente se utilizan para representar sistemas de partículas de forma eficiente, representando las partículas como puntos en lugar de rectángulos (un rectángulo en OpenGL se representa mediante 2 triángulos).

Líneas

Las primitivas de líneas soportadas por OpenGL ES son GL_LINES, GL_LINE_STRIP y GL_LINE_LOOP.

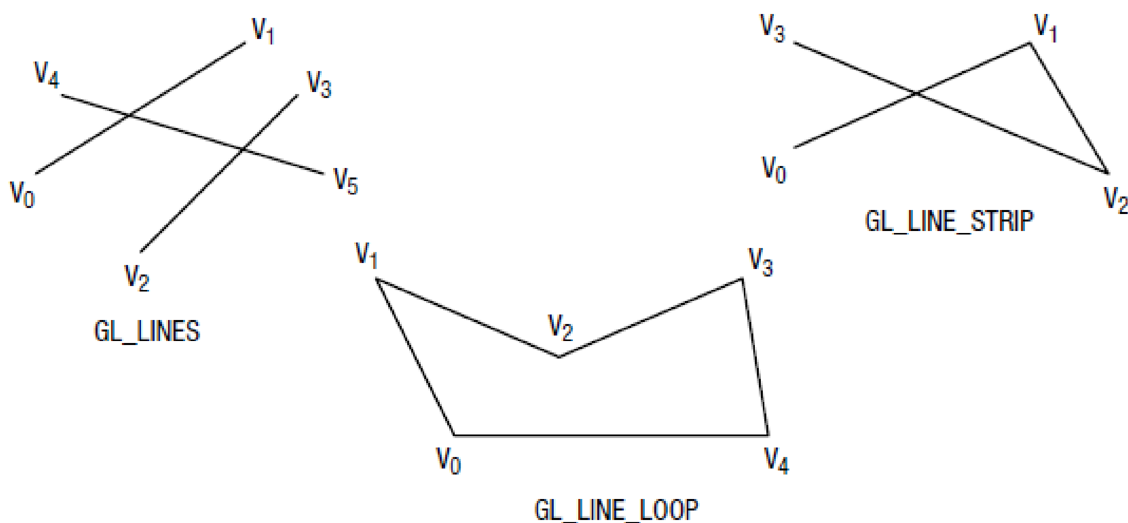

Figura 15. Tipos de primitivas de líneas.

GL_LINES trata cada par de vértices como un segmento de línea. Por ejemplo, si se especifican los vértices V0, V1, V2, V3, V4 y V5, dibujará 3 líneas: (V0, V1), (V2, V3) y (V4, V5).

GL_LINE_STRIP dibuja una serie de líneas conectadas en la que todos los vértices están conectados por un segmento de línea. El último vértice especificado no estará conectado al primero de la serie. Por ejemplo, si se especifican los vértices V0, V1, V2 y V3, dibujará 3 líneas: (V0, V1), (V1, V2) y (V2, V3).

GL_LINE_LOOP funciona igual que GL_LINE_STRIP salvo que en este caso, además se dibuja una línea que conecta el último vértice con el primero. Por ejemplo, si se especifican los vértices V0, V1, V2, V3 y V4, dibujará 5 líneas: (V0, V1), (V1, V2), (V2, V3), (V3, V4) y (V4, V0).

Triángulos

Las primitivas de triángulos soportadas por OpenGL ES son `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` y `GL_TRIANGLE_FAN`.

En OpenGL ES el método más común para describir objetos geométricos es utilizar mallas de triángulos. Por ejemplo, un rectángulo se definiría como una malla de 2 triángulos.

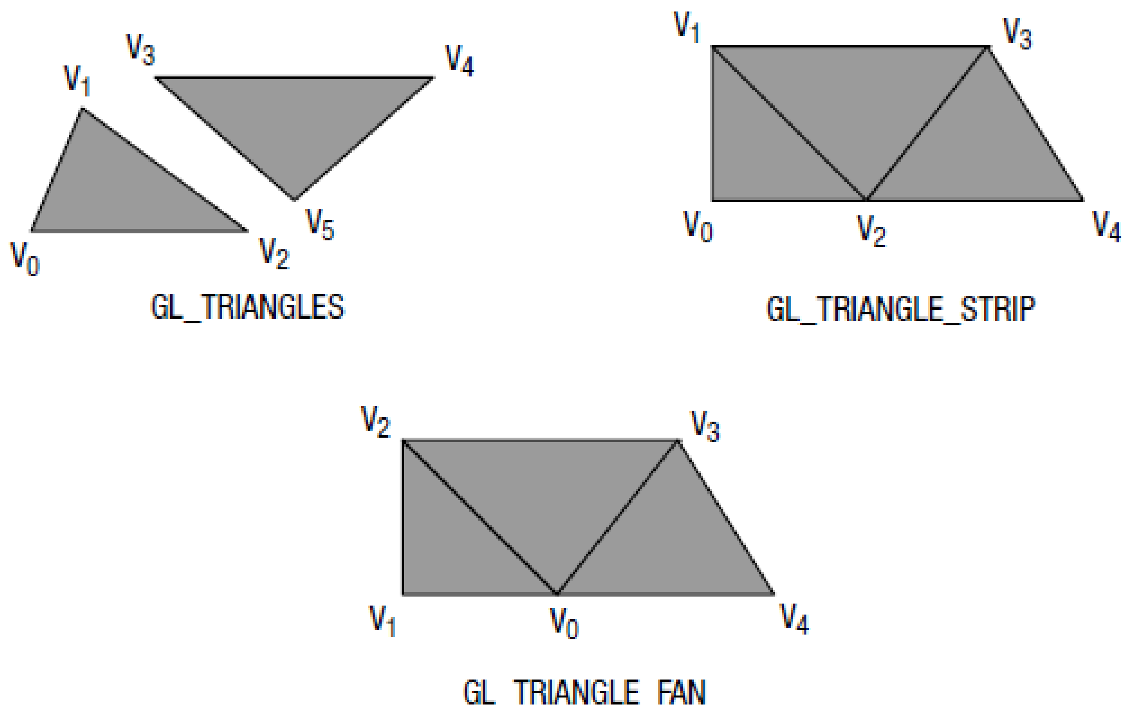

Figura 16. Tipos de primitivas de triángulos.

`GL_TRIANGLES` trata cada 3 vértices como un triángulo independiente. Con esta opción, OpenGL dibuja *n / 3* triángulos, donde *n* es el número de índices especificado en la llamada a *glDrawElements* o *glDrawArrays*. Por ejemplo, si se especifican los vértices V0, V1, V2, V3, V4 y V5, dibujará 2 triángulos: (V0, V1, V2) y (V3, V4, V5).

`GL_TRIANGLE_STRIP` dibuja una serie de triángulos conectados. El orden en el que se especifican los vértices es importante [55], ya que para cada nuevo vértice *n* que se especifique a partir del tercero, se construye un nuevo triángulo *T*. Si el vértice *n* es par, *T = [n - 1, n - 2, n]*, y si es impar, *T = [n – 2, n – 1, n]*. Por ejemplo, si se especifican los vértices V0, V1, V2, V3 y V4, dibujará 3 triángulos: (V0, V1, V2), (V2, V1, V3) y (V2, V3, V4).

`GL_TRIANGLE_FAN` también dibuja una serie de triángulos conectados, salvo que en este caso el primer vértice del vertex stream es común a todos los triángulos. Por ejemplo, si se especifican los vértices V0, V1, V2, V3 y V4, dibujará 3 triángulos: (V0, V1, V2), (V0, V2, V3) y (V0, V3, V4). Esta primitiva se puede utilizar para representar círculos.

3.5.1.3. Transform and Lightning

En esta etapa, a la que en algunos documentos se hacer referencia como T&L, cada vértice del vertex stream se transforma de forma individual al sistema de coordenadas de la vista usando la matriz `GL_MODELVIEW`. Tras transformar la posición y la normal del vértice, se computa la información de iluminación del mismo.

Como esta etapa se ejecuta una vez para cada vértice, no es capaz de aplicar iluminación por píxel. Esto es una limitación del pipeline de función fija y se resuelve con el pipeline programable (a partir de OpenGL ES 2.0).

3.5.1.3.1. Primitive Assembly

Durante esta etapa, OpenGL crea las primitivas especificadas a partir de los vértices y además comprueba si las primitivas se salen de la ventana de recorte (*viewport*) y efectúa los recortes que sean necesarios (*clipping*).

3.5.1.3.2. Rasterizer

En esta etapa se convierten las primitivas en fragmentos (*fragments*), que más adelante se convertirán en píxeles.

Un fragment está formado por los datos necesarios para generar un píxel. Algunos de estos datos son:

- Posición.
- Profundidad.
- Atributos interpolados (color, coordenadas de textura, etc.).
- Alfa.

Los atributos de los vértices se interpolan linealmente entre vértices para obtener los valores de dichos atributos para cada fragment individualmente.

3.5.1.4. Texture Environment

En esta etapa se aplican las texturas que se hubieran especificado previamente a través de la API a los fragments.

3.5.1.5. Color Sum

En esta etapa se puede agregar un color secundario a la geometría después de que las texturas hayan sido aplicadas.

Por ejemplo, si la iluminación se calcula antes de aplicar las texturas, el efecto resultante tras aplicarlas será que las texturas son más brillantes en las zonas donde incide la luz con más intensidad. Esta etapa hace posible, además agregar un color para simular que la luz es de dicho color.

3.5.1.6. Fog

En esta etapa, si se especifica a través de la API, se simula un desvanecimiento de la geometría. De esta forma da la sensación de que hay niebla en la escena.

Esto se utiliza sobre todo cuando se renderizan escenas que representan espacios muy grandes al aire libre. En estos casos, como la escena normalmente no cabe en el volumen de visualización (explicado anteriormente en el apartado 3.2.2.4), se vería un corte vertical en el plano de recorte trasero. Para disimular dicho corte, se simula niebla en las zonas más lejanas, de forma que la geometría se irá desvaneciendo gradualmente hasta el color de niebla especificado.

3.5.1.7. Alpha Test

Durante esta etapa, los fragments cuyo valor alpha esté por debajo de un cierto umbral se descartan. Esta etapa solo funciona si se activa utilizando `GL_ALPHA_TEST` y el frame buffer utiliza un modo de color que almacene valores de alpha, como el RGBA.

3.5.1.8. Depth Stencil

Durante el depth test, un fragment puede ser descartado si el frame buffer actual tiene un depth buffer (buffer de profundidad), el depth test está activado mediante `GL_DEPTH_TEST` y falla la comparación de profundidad. Por defecto, un fragment se descarta si el valor de profundidad es mayor que el valor de profundidad del fragment actual. Es decir, si hay un fragment en la posición (x, y) con profundidad 1 y se intenta renderizar otro fragment en la misma posición pero con profundidad 2, el segundo fragment se descarta.

El stencil test se realiza de forma similar. Se activa mediante `GL_STENCIL_TEST` y el valor de stencil del fragment se compara con el valor del stencil buffer que se encuentre en la misma posición, y si el test falla, el fragment se descarta. Es importante destacar que si el depth test falla para un fragment, el stencil test no se llega a realizar para dicho fragment. Esta característica sirve para tener más control sobre las zonas de la imagen que se muestran en pantalla [56].

3.5.1.9. Color Buffer Blend

Esta etapa se ejecuta si `GL_BLEND` está activado, y su objetivo es llevar a cabo la fusión de los colores de los fragments que se están renderizando y los colores del frame buffer. A esta operación se la denomina normalmente blending.

El blending consiste en combinar los colores de los fragments que ha procesado el fragment shader con los colores de los fragments que se encuentran en la misma posición de destino en el buffer donde los fragments procesados deben escribirse. Es decir, cuando se va a escribir en el buffer un fragment que se mostrará en la posición (x, y), si la opción `GL_BLEND` está activa, se fusiona el color de dicho fragment con el color del fragment del buffer que se fuese a mostrar en la posición (x, y). Esta fusión de colores se realiza de acuerdo a la ecuación de blending seleccionada, con los parámetros que se especifique [74].

Las ecuaciones de blending utilizadas en OpenGL ES son las siguientes: `GL_FUNC_ADD`, `GL_FUNC_SUBTRACT` y `GL_FUNC_REVERSE_SUBTRACT`. Se puede especificar qué función utilizar mediante `glBlendEquation(mode)`. Esto se puede encontrar explicado con mayor detalle en la referencia [75].

En cuanto a los parámetros de dichas funciones, se pueden especificar utilizando la función `glBlendFunc` de OpenGL. Estos parámetros son dos y especifican cómo se debe tratar el color fuente (el que ya estaba en el buffer) y cómo se debe tratar el color destino (el que devuelve el fragment shader). Esto, así como cada uno de los parámetros posibles, se puede encontrar explicado con mayor detalle en la referencia [76].

3.5.1.10. Dither

Esta etapa se ejecuta si `GL_DITHER` está activado, y su función es, en caso de estar utilizando una paleta de colores con pocos colores, simular una paleta de colores más variada combinando colores que se encuentren cercanos entre sí. Por ejemplo, si al representar un degradado los cambios de color son muy bruscos, es bastante probable que al activar el dithering los cambios sean más suaves.

3.5.2. Pipeline de rendering programable

El pipeline de rendering programable se llama así debido al hecho de que es posible programar parte de su comportamiento. El programador tiene mucho más control sobre la forma en la que se renderizan los objetos que en el pipeline de rendering fijo.

El siguiente diagrama muestra las distintas etapas del pipeline programable. Los nodos más oscuros representan las etapas del pipeline que existían previamente en el pipeline de función fijo (todas ellas explicadas en la sección anterior).

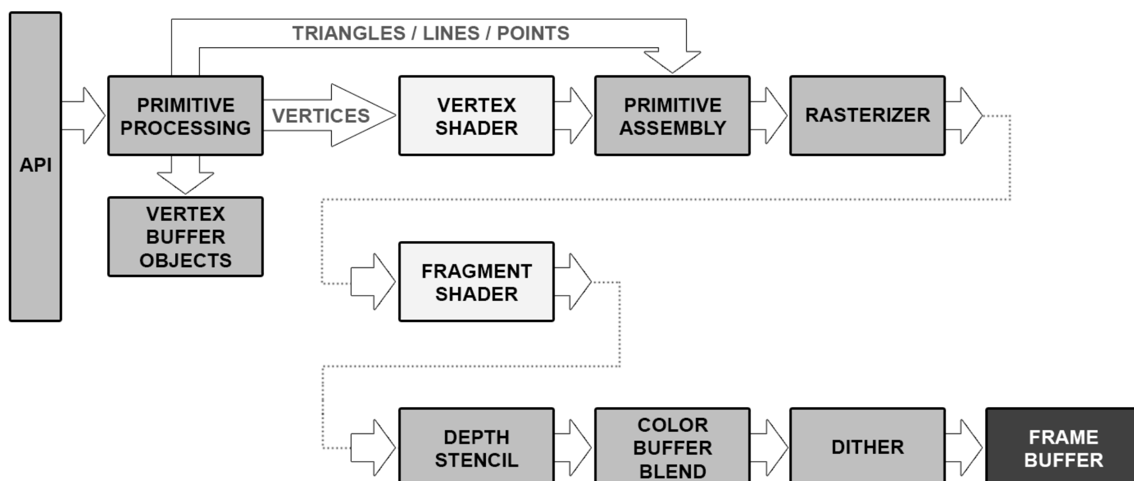

Figura 17. Pipeline de rendering de OpenGL ES 2.0.

Comparando los diagramas del pipeline de rendering de función fija (*Figura 14*) y el pipeline de rendering programable (*Figura 17*), se pueden observar los siguientes cambios:

- La etapa *Transform and Lightning* se ha sustituido por el *vertex shader* (explicado más adelante).
- Las etapas *Texture Environment, Color Sum, Fog* y *Alpha Test* se han sustituido por el *fragment shader* (explicado más adelante).

3.5.2.1. Vertex Shader

El vertex shader es un programa que se utiliza para operar sobre los vértices. Se ejecuta una vez por vértice y normalmente se utiliza principalmente para aplicar transformaciones a los vértices y para aplicar efectos de iluminación simples. Tal y como se hacía en la etapa *Transform and Lightning* del pipeline de función fija, pero con mayor flexibilidad, ya que el vertex shader lo puede implementar cada programador de la forma que mejor se adapte a sus necesidades.

La siguiente figura muestra las distintas variables a las que tiene acceso el vertex shader.

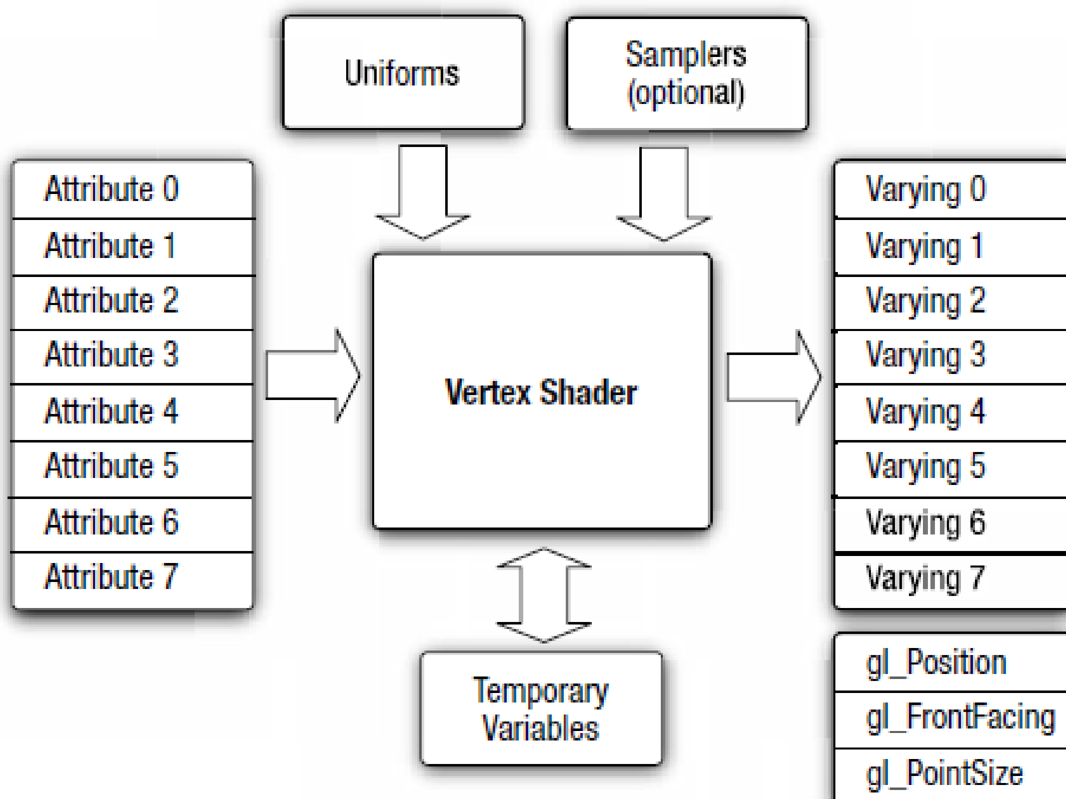

Figura 18. Vertex Shader de OpenGL ES 2.0.

En la figura anterior se pueden identificar varios tipos de variables:

- **Attributes:** Datos específicos de cada vértice.
- **Uniforms:** Datos constantes para todos los vértices.
- **Samplers:** Representan texturas.
- **Varying Variables:** De estas variables se obtienen las entradas del fragment shader, que se calculan durante la etapa de rasterización de primitivas. En el vertex shader se le asigna valor a una varying variable para un vértice determinado, posteriormente en la etapa de rasterización se interpola esta variable entre los vértices para obtener el valor de la variable para cada fragment y la variable con el valor interpolado será la entrada del fragment shader, que se explica en el siguiente apartado.
- **Variables predefinidas:** OpenGL ES 2.0 tiene algunas variables especiales predefinidas que se utilizan para especificar las salidas de los shaders. Las variables predefinidas del vertex shader son:
    - **gl_Position:** Se utiliza para especificar la posición del vértice en coordenadas de proyección (explicadas anteriormente en el apartado 3.2.2.4).
    - **gl_PointSize:** Se utiliza para definir el tamaño de un point sprite, en píxeles.
    - **gl_FrontFacing:** Esta variable especial, aunque no se modifica directamente en el vertex shader, se calcula basándose en las posiciones calculadas en el vertex shader y el tipo de primitiva.
- **Variables temporales:** Es posible definir variables temporales dentro del shader.

También existen constantes predefinidas como `gl_MaxVertexUniformVectors`, `gl_MaxVaryingVectors`, `gl_MaxVertexTextureImageUnits`, etc.

En OpenGL ES 2.0, el vertex shader tiene una serie de limitaciones, algunas de ellas se exponen a continuación:

- El número máximo de sentencias que puede contener el vertex shader puede variar de una implementación a otra y no hay forma de saber si un shader sobrepasa el máximo número de sentencias soportado.
- El número máximo de variables temporales soportado también puede variar de una implementación a otra y tampoco puede ser consultado desde el shader.
- No es posible declarar todas las variables tipo uniform que el programador desee. Sin embargo, existe un número mínimo de uniforms que todo hardware que implemente OpenGL ES 2.0 está obligado a soportar, y son 128 entradas `vec4` [43].

Se puede encontrar más información sobre el vertex shader y sus características y limitaciones en [43] y [52].

3.5.2.2. Fragment Shader

La función principal del fragment shader consiste en determinar el color de un fragment.

El fragment shader se ejecuta para todos los fragments, una vez por fragment.

Anteriormente, cuando se describían OpenGL ES 1.X y el pipeline de función fija, se mencionaba que con dicho pipeline era imposible aplicar la iluminación por píxel, sin embargo, en el pipeline programable sí que es posible aplicar dicha iluminación gracias al fragment shader, ya que ofrece al programador un control mucho más minucioso de las operaciones que se realizan a nivel de fragment.

La siguiente figura muestra las distintas variables a las que tiene acceso el fragment shader.

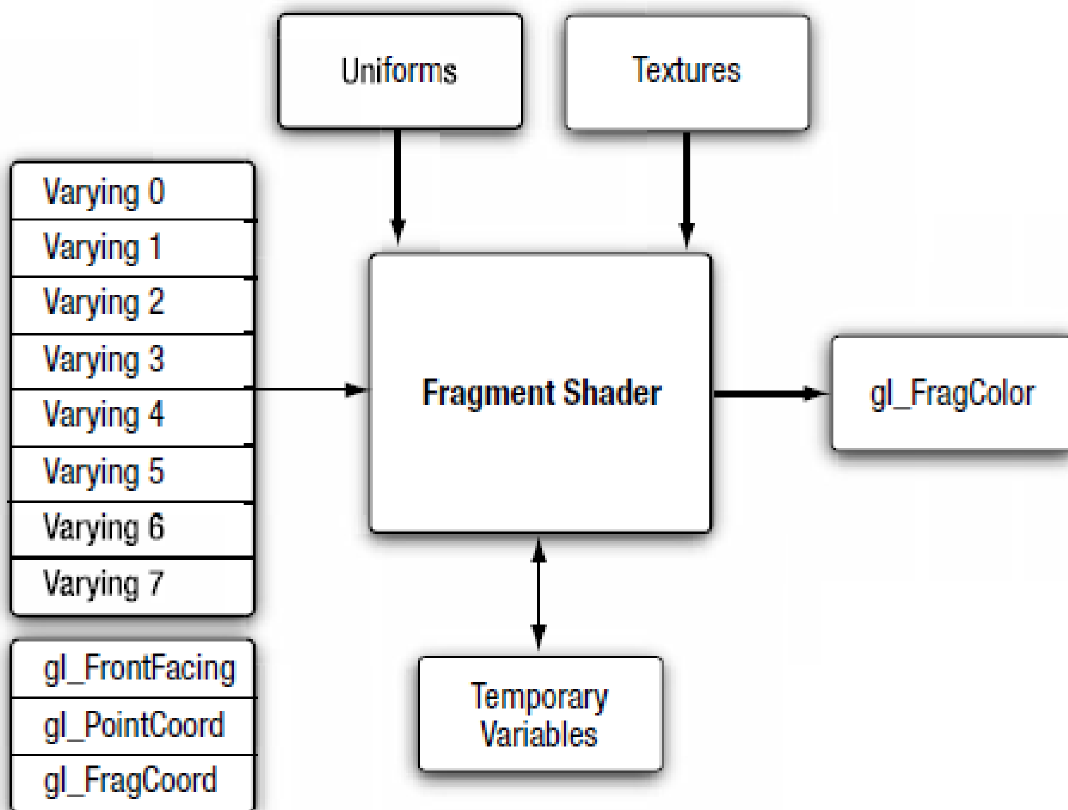

Figura 19. Fragment Shader de OpenGL ES 2.0.

En la figura anterior se pueden identificar varios tipos de variables:

- **Varying Variables:** Variables que contienen los valores asignados en el vertex shader, pero interpolados para cada fragment.
- **Uniforms:** Datos constantes para todos los fragments.
- **Textures:** Texturas a las que se accede mediante samplers.
- **Variables temporales:** Es posible definir variables temporales dentro del shader.

- **Variables predefinidas:**
  - **gl_FragColor:** Esta variable se utiliza para definir el color del fragment desde el fragment shader. Su valor se pasa después a las etapas posteriores del pipeline de rendering.
  - **gl_FragCoord:** Es una variable de solo lectura que almacena las coordenadas del fragment relativas a la ventana en el formato (*x, y, z, 1 / w*).
  - **gl_FrontFacing:** Es una variable de solo lectura que vale `true` si el fragment es parte de una primitiva que está orientada hacia delante, y `false` en caso contrario.
  - **gl_PointCoord:** Es una variable de solo lectura que se puede utilizar cuando se renderizan point sprites. Contiene las coordenadas de textura del point sprite, que se generan automáticamente en el rango [0.0, 1.0] en la etapa de rasterización.

Además desde el fragment shader es posible acceder a algunas constantes predefinidas. Como `gl_TextureImageUnits`, `gl_MaxFragmentUniformVectors` y `gl_MaxDrawBuffers`, entre otras.

En cuanto a las limitaciones del fragment shader en OpenGL ES 2.0, la más destacable es que solo es posible acceder a posiciones concretas de un array de uniforms mediante índices enteros constantes. No es posible indexar utilizando variables. Por lo demás, comparte las limitaciones de los vertex shaders mencionadas en el apartado anterior.

Al igual que se menciona en el apartado sobre los vertex shaders, se puede encontrar más información sobre el fragment shader, sus características y limitaciones en las referencias [43] y [52].

3.6. Soporte de OpenGL ES en Android

En Android hay dos formas de utilizar OpenGL ES, a través del NDK (*Native Development Kit*) o a través del SDK (*Software Development Kit*). La principal diferencia es que el NDK permite a los desarrolladores trabajar con C/C++, mientras que el SDK permite trabajar con Java.

Al utilizar OpenGL ES a través de la API proporcionada por el SDK de Android, éste internamente llama a la implementación en C de las funciones de OpenGL ES.

Android soporta OpenGL ES 1.0 desde sus comienzos, y OpenGL ES 1.1 desde la versión 1.6 (Donut). En cuanto a OpenGL ES 2.0, lo soporta desde la versión 2.2 (Froyo), sin embargo, no toda la API estaba disponible a través del SDK de Android 2.2. A algunas funciones de OpenGL ES 2.0 solo se podía acceder utilizando el NDK. La API completa no estuvo disponible en el SDK hasta Android 2.3 (Gingerbread).

Actualmente (Julio, 2014), la versión de OpenGL ES más alta soportada por Android es OpenGL ES 3.0. Esta versión tiene soporte solo a partir de Android 4.3 [57], que fue lanzada en Julio de 2013.

En la siguiente tabla se puede ver el porcentaje de dispositivos Android que soporta cada versión de OpenGL ES. Téngase en cuenta que si un dispositivo soporta una versión, implica también que soporta todas las anteriores [58].

Versión más alta soportada de OpenGL ES	Porcentaje de dispositivos
1.1	0.1%
2.0	83.6%
3.0	16.3%

Tabla 1. Porcentaje de dispositivos Android con soporte para cada versión de OpenGL en Junio de 2014.

De la tabla anterior podemos extraer que, en Junio de 2013, el 100% de los dispositivos Android soportan OpenGL ES 1.1, el 99.9% de los dispositivos soportan OpenGL ES 2.0 y el 16.3% de los dispositivos soportan OpenGL ES 3.0.

3.7. ¿Por qué utiliza DroidEngine2D OpenGL ES 2.0?

En Android hay principalmente dos formas de dibujar en pantalla los elementos de un juego 2D, y son las siguientes:

- Utilizar la clase `Canvas` proporcionada por el SDK de Android.
- Utilizar OpenGL ES.

El rendering basado en `Canvas` lo lleva a cabo la CPU mediante lo que se conoce como *software rendering*: todos los cálculos necesarios para determinar el color de cada píxel se llevan a cabo mediante software y los realiza la CPU.

Por otro lado, OpenGL ES utiliza la GPU y gran parte del proceso está automatizado mediante hardware, lo cual, sumado al gran poder de procesamiento de las GPUs, hace que sea más eficiente.

Para implementar el renderer de DroidEngine2D se optó por utilizar OpenGL ES en lugar de `Canvas` por ser el primero más eficiente, ya que hace uso de la GPU, como se ha explicado anteriormente. El rendering basado en `Canvas` puede ser suficiente para juegos que necesiten representar pocos sprites en pantalla, pero si un juego requiere representar un gran número de sprites, el rendering basado en GPU es mejor opción.

Una vez explicada la principal razón por la que se escogió OpenGL ES frente a `Canvas`, queda explicar por qué se escogió OpenGL ES 2.0 en lugar de OpenGL ES 3.0, que es la versión más reciente soportada por Android. Esto se debe a que cuando se comenzó a trabajar en el proyecto, a finales de 2012, la versión más alta soportada de OpenGL ES en Android era OpenGL ES 2.0. La versión 3.0 no tuvo soporte hasta Julio de 2013, cuando Google lanzó Android 4.3 [57]. Y el motivo por el que no se ha agregado soporte para OpenGL ES 3.0 posteriormente es que el porcentaje de usuarios que disponían de dispositivos con soporte para esta versión era extremadamente bajo.

De los datos expuestos en la *Tabla 1*, se puede extraer que en Junio de 2013 el 99.9% de los dispositivos Android soportaban OpenGL ES 2.0, mientras que un 16.3% soportaban además OpenGL ES 3.0, por lo que una posible mejora de cara a futuras versiones de DroidEngine2D puede ser introducir optimizaciones disponibles en OpenGL ES 3.0 para que los dispositivos con soporte para dicha versión puedan aprovecharlas.

3.8. Conclusiones

En este capítulo se ha explicado qué es OpenGL ES, así como cuáles son las distintas versiones existentes y las principales diferencias entre ellas.

Además se han explicado conceptos básicos de OpenGL, como los distintos sistemas de coordenadas por los que pasan los vértices de los objetos desde que se definen hasta que se muestran en pantalla, y los pipelines de rendering. De esta forma es posible hacerse una idea del funcionamiento de OpenGL y de las ventajas del pipeline de rendering programable frente al pipeline de rendering de función fija.

Tal y como se ha expuesto, OpenGL proporciona al programador bastante flexibilidad a la hora de renderizar objetos 3D, no obstante, la curva de aprendizaje es más lenta que la de otras soluciones no tan eficientes, como es el caso de `Canvas` en Android. Para juegos extremadamente simples, es posible que utilizar OpenGL no suponga una mejora visible en el rendimiento frente a utilizar rendering mediante CPU. Sin embargo, para un motor de juegos, como DroidEngine2D, que puede utilizarse para crear juegos más complejos, utilizar OpenGL es mejor opción.

En cuanto a implementar un motor de juegos para Android en la actualidad (Julio, 2014), la mejor opción para el sistema de rendering es utilizar OpenGL ES 2.0, ya que el 99.9% de dispositivos lo soportan. Adicionalmente se podrían implementar algunas optimizaciones que aporta OpenGL ES 3.0 para que los dispositivos que soporten dicha versión puedan beneficiarse de un mejor rendimiento, teniendo en cuenta que estos dispositivos representan solo un 16.3% del total, por lo que habría que mantener el motor compatible con los dispositivos que soporten únicamente hasta la versión 2.0.

4

DroidEngine2D

4.1. Introducción

En este capítulo se profundizará más en el proyecto. En primer lugar, se especificarán las principales funciones y características del proyecto, a qué tipo de usuarios va dirigido, etc. En segundo lugar, se explicará detalladamente el diseño y desarrollo del proyecto, haciendo hincapié en los sistemas o módulos más importantes. También se comentarán las distintas demos que se han realizado, en las que se puede comprobar el funcionamiento de DroidEngine2D en conjunto, así como de sus principales características de forma aislada.

Este capítulo puede ser de especial utilidad a cualquier desarrollador que desee utilizar DroidEngine2D o contribuir al proyecto, ya que en él se exponen sus principales características y además se explican algunas de las decisiones más importantes que se han tomado durante el diseño y el desarrollo del mismo.

4.1.1. ¿Qué es DroidEngine2D?

DroidEngine2D es un motor de juegos para Android que pretende facilitar la implementación de videojuegos en dos dimensiones para dicha plataforma.

El principal objetivo de este proyecto es aportar una herramienta software completa, de código abierto, que permita desarrollar videojuegos completos para Android de forma sencilla y que a su vez sea fácilmente extensible.

El código fuente del proyecto está licenciado bajo la licencia Apache 2.0 y se puede consultar online o descargar a través de la referencia [61]. Además, se ha generado la documentación completa de la API del motor (javadoc), la cual se puede consultar online en la referencia [62].

Y las demos del motor también están licenciadas bajo la licencia Apache 2.0 y se pueden obtener a través de la referencia [63].

4.1.2. Beneficios de DroidEngine2D

En este apartado se describen algunos de los principales beneficios que aporta DroidEngine2D a un programador de videojuegos.

En el capítulo anterior se explicaba de forma resumida el funcionamiento de OpenGL y concretamente de OpenGL ES 2.0, que es la versión que utiliza DroidEngine2D (por los motivos previamente expuestos en el apartado 3.7). Como se ha podido observar, OpenGL es una librería muy potente, pero la curva de aprendizaje es lenta.

Para poder trabajar con OpenGL ES 2.0, entre otras cosas, el programador necesita:

- Entender el funcionamiento del pipeline de rendering programable.
- Estar familiarizado con las matrices de transformación homogénea.
- Conocer los distintos sistemas de coordenadas.
- Saber cómo y cuándo transformar los vértices de un sistema de coordenadas a otro.
- Tener conocimientos de GLSL para poder implementar los shaders.

DroidEngine2D se ha diseñado de forma que el usuario del motor pueda crear videojuegos sin necesidad de conocer el funcionamiento de OpenGL. Aunque por otro lado, este motor también se ha diseñado para ser extensible, permitiendo a los usuarios que sí tengan conocimientos de OpenGL, por ejemplo, implementar sus propios shaders en caso de que los incluidos en el motor no se adapten completamente a sus necesidades.

El motor además gestiona los eventos de entrada de usuario, manteniendo internamente una cola de eventos que se procesan de forma secuencial en el hilo del juego, lo cual evita algunos problemas que pueden surgir al procesar los eventos asíncronos en el momento en el que se reciben. También proporciona mecanismos de gestión de estados del juego, recarga automática de texturas, reproducción de música y efectos de sonido, etc.

Además, este motor se integra perfectamente con el ciclo de vida de las activities de Android, de forma que, cuando la aplicación vuelve a primer plano tras haber estado en segundo plano, no sea necesario que el programador recargue manualmente las texturas, o gestione el hilo del juego y el hilo del renderer de forma manual, entre otras cosas.

En resumen, DroidEngine2D proporciona al programador una serie de facilidades a la hora de crear videojuegos en dos dimensiones para Android, permitiéndole centrarse principalmente en desarrollar la lógica del juego sin tener que tratar con otros aspectos de más bajo nivel.

4.2. Descripción general del proyecto

En esta sección se presenta una descripción a alto nivel del proyecto. Se expondrá la perspectiva del producto, las principales funciones que es capaz de llevar a cabo, las características esperadas de los usuarios del motor, las restricciones, dependencias y otros factores que afecten al desarrollo y uso del mismo.

4.2.1. Perspectiva del producto

DroidEngine2D, a diferencia de otros más complejos como Unity o Unreal Engine, no posee un editor gráfico. Los escenarios deben diseñarse a mano o haciendo uso de un editor externo. En este aspecto es más parecido a motores como libGDX, AndEngine y XNA, ya que no cuenta con interfaz gráfica de ningún tipo.

Esta librería depende del SDK de Android [65] y es compatible con todas las versiones de Android desde la versión 2.3 (Gingerbread) en adelante. Nota: actualmente (Julio, 2014) la última versión de Android disponible es la versión 4.4 (KitKat).

Además hace uso de la librería Collections [64], que es una librería de código abierto desarrollada por el autor de este proyecto. Esta librería proporciona algunas estructuras de datos diseñadas con el objetivo de minimizar el número de objetos que se crean después de que éstas hayan sido inicializadas, reduciendo el número de veces que se ejecuta el recolector de basura.

De las 2 dependencias mencionadas, el SDK de Android debe estar instalado en la máquina que el usuario del motor vaya a utilizar para desarrollar videojuegos haciendo uso del mismo, mientras que la librería Collections se distribuye junto al motor, por lo que no es necesario que el usuario del motor la descargue manualmente.

Es posible desarrollar videojuegos con DroidEngine2D en las plataformas en las que el SDK de Android es compatible, es decir, en Windows, Mac OS X y Linux [65].

4.2.2. Funciones del producto

En este apartado se exponen, en forma de lista, las principales funciones que realiza DroidEngine2D, organizadas en bloques o módulos. Más adelante se explicará cada bloque por separado en mayor detalle.

4.2.2.1. General

A continuación se exponen las principales funciones generales de DroidEngine2D.

- Bucle del juego.
    - Gestión automática del bucle del juego.
- Integración del motor con el ciclo de vida de las activities de Android.
    - Inicializar automáticamente el motor al crear la activity.

  - Pausar automáticamente el motor cuando la activity pasa a segundo plano.
  - Reanudar automáticamente el motor cuando la activity pasa a primer plano.
  - Liberar automáticamente todos los recursos cuando se destruye la activity.

4.2.2.2. Audio

A continuación se exponen las principales funciones de DroidEngine2D relacionadas con la reproducción de audio.

- Gestión de pistas de audio.
  - Reproducir pistas de audio desde archivos, sin cargarlas completamente en memoria.
  - Pausar y reanudar la reproducción de pistas de audio.
  - Parar la reproducción de pistas de audio.
  - Modificar el volumen de reproducción.
  - Reproducir la pista de audio una sola vez o en modo bucle.
  - Comprobar si es posible reproducir un archivo especificado.
- Gestión de efectos de sonido.
  - Cargar efectos de sonido en memoria y almacenarlos indexados por un identificador especificado.
  - Reproducir efectos de sonido previamente almacenados en memoria.
  - Liberar recursos de forma que no quede ningún efecto de sonido cargado en memoria.

4.2.2.3. Gestión de estados del juego

A continuación se exponen las principales funciones de DroidEngine2D relacionadas con la gestión de estados del juego.

- Crear estados del juego (menús, niveles, etc.).
- Registrar estados en el gestor de estados.
- Eliminar estados del gestor de estados.
- Gestión de la pila de estados activos.
  - Agregar estado a la pila de estados activos.
  - Eliminar uno o varios estados de la lista de estados activos.
  - Intercambiar el estado en la cima de la pila de estados activos por otro estado especificado.

    - Consultar el estado en la cima de la pila de estados activos.
- Registrar callbacks que se llaman cuando se produce un cambio en la pila de estados activos.

4.2.2.4. Gráficos

A continuación se exponen las principales funciones de DroidEngine2D relacionadas con la representación de elementos en pantalla, y con la gestión de elementos gráficos en general.

- Animaciones.
    - Reproducir animaciones.
    - Pausar y reanudar animaciones.
    - Reiniciar animaciones.
    - Reproducir las animaciones una sola vez o en modo bucle.
    - Definir callbacks que se llaman cuando ocurren determinados eventos en una animación (cuando se inicia, cuando cambia de frame, cuando se pausa, cuando se reanuda, cuando termina un ciclo, etc.).
- Materiales.
    - Definir materiales personalizados.
    - Utilizar materiales definidos por defecto.
        - Color.
        - Textura.
        - Textura con control de opacidad.
        - Textura con superposición de color RGBA.
        - Textura a la que se le puede alterar el tono, la saturación y el brillo.
- Renderers de materiales.
    - Definir renderers de materiales personalizados.
    - Utilizar renderers de materiales definidos por defecto. Los renderers de materiales definidos por defecto permiten representar conjuntos de rectángulos con cualquiera de los materiales definidos por defecto.
- Texto.
    - Representar texto en pantalla.
    - Fuentes.
        - Utilizar fuentes definidas en el formato de representación de BMFont.

        - Definir un cargador de fuentes personalizado para poder cargar fuentes definidas en el formato que se desee.

- Gestión de texturas.
    - Recargar todas las texturas automáticamente cuando se reinicializa el contexto de OpenGL.
    - Atlas de texturas.
        - Cargar atlas de texturas definidos con TexturePacker en formato XML.
        - Definir un cargador de atlas de texturas personalizado para poder cargar atlas de texturas definidos en el formato que se desee.

4.2.2.5. Entrada de usuario

A continuación se exponen las principales funciones de DroidEngine2D relacionadas con la gestión de la entrada de usuario.

- Gestión de la entrada de usuario en cada estado del juego de forma independiente.
    - Gestión de eventos de entrada táctil.
        - Encolar eventos de entrada táctil cuando se reciben.
        - Procesar en el hilo del juego, secuencialmente, los eventos de entrada táctil encolados.
        - Utilizar un procesador de eventos personalizado.
    - Gestión de eventos de entrada mediante teclas físicas del dispositivo.
        - Encolar eventos de entrada mediante teclas físicas del dispositivo cuando se reciben.
        - Procesar en el hilo del juego, secuencialmente, los eventos de entrada mediante teclas físicas del dispositivo encolados.
        - Utilizar un procesador de eventos personalizado o el procesador de eventos por defecto.
    - Captura de datos del acelerómetro.
        - Comenzar a capturar valores del acelerómetro.
        - Parar de capturar valores del acelerómetro.
        - Utilizar el listener por defecto, que permite acceder a las coordenadas del vector de aceleración.
        - Utilizar un listener personalizado.

4.2.2.6. Carga de recursos

A continuación se exponen las principales funciones de DroidEngine2D relacionadas con la carga de recursos.

- Lectura de archivos de imagen.
- Lectura de archivos de audio.

4.2.2.7. Utilidades

Además de las funciones mencionadas en los apartados anteriores, DroidEngine2D proporciona algunas utilidades:

- Tiempo.
    - Contar el tiempo transcurrido.
    - Facilitar la conversión entre unidades de tiempo proporcionando constantes para las unidades de tiempo más utilizadas.
- Vectores y matrices.
    - Realizar operaciones básicas con vectores de dos y tres coordenadas.
    - Realizar operaciones básicas con matrices de 4x4.
    - Trabajar con matrices de transformación homogénea.
- Android.
    - Detectar si el dispositivo soporta OpenGL ES 2.0.
    - Detectar la orientación por defecto del dispositivo.

4.2.3. Características de los usuarios

DroidEngine2D está destinado a desarrolladores con experiencia en la programación de aplicaciones en Java para Android. Además es recomendable tener conocimientos básicos sobre el desarrollo de videojuegos.

4.2.4. Restricciones

Con DroidEngine2D no es posible crear aplicaciones compatibles con las versiones de Android anteriores a Android 2.3 (Gingerbread). Esto se debe, en primer lugar, a que hasta la versión 2.2 (Froyo), no existía soporte para OpenGL ES 2.0 en el SDK de Android y, aun en esta versión, el soporte no era completo; y en segundo lugar, a que en la actualidad (Julio, 2014) solo el 0.8% de dispositivos utilizan Android 2.2 (datos de Junio de 2014 [58]).

4.2.5. Suposiciones y dependencias

Tal y como ya se ha mencionado anteriormente, DroidEngine2D es una librería que necesita que el SDK de Android esté instalado en la máquina que el usuario del motor vaya a utilizar para desarrollar videojuegos haciendo uso del mismo. El SDK de Android se puede obtener a través de la referencia [65].

El entorno de desarrollo utilizado para desarrollar este proyecto ha sido Eclipse [66] haciendo uso del plugin ADT (*Android Developer Tools*). Las últimas pruebas se han realizado utilizando la versión 22.6.3 (Abril, 2014), por lo que se recomienda utilizar dicha versión, aunque puede funcionar también en versiones anteriores. El plugin ADT se puede obtener a través de la referencia [67].

Se asume que el usuario del motor lo utilizará para desarrollar proyectos en Eclipse con el plugin mencionado anteriormente. No obstante, al ser un proyecto de código abierto, es posible adaptarlo para desarrollar proyectos en otros entornos de desarrollo si fuese necesario.

4.3. Desarrollo del proyecto

DroidEngine2D ha sido desarrollado siguiendo una metodología iterativa e incremental, basada en análisis, diseño, implementación y pruebas de cada una de las características de las mencionadas en el apartado 4.4.2.

En esta sección se describe detalladamente el proceso de análisis, diseño, implementación y pruebas de cada una de las características. En el apartado de análisis se identifica la funcionalidad necesaria para implementar la característica, en el apartado de diseño e implementación se detalla mejor cómo se ha planteado la solución para cubrir la característica en concreto y se exponen los detalles más relevantes de la implementación; y en el apartado de pruebas se comentan las principales pruebas que se han realizado para verificar que la característica implementada funciona correctamente y está lista para ser integrada en el motor.

El sistema de control de versiones utilizado para este proyecto es Git [68]. Y el modelo de branching (ramificación) utilizado ha sido *Git-Flow*. El funcionamiento completo de este modelo de branching se puede encontrar en la referencia [69]. En cuanto al servidor Git utilizado, inicialmente se utilizaba un servidor privado y posteriormente se decidió publicar el código del proyecto en GitHub [70].

La gestión de tareas se ha realizado con la aplicación *Personal Kanban Board*, que es una aplicación creada por el autor de este proyecto que permite organizar las tareas en columnas y visualizar todas las tareas en un tablero kanban [71] de forma que se pueda ver rápidamente el estado del proyecto.

A continuación se describe el desarrollo de cada una de las características del motor expuestas anteriormente en el apartado 4.4.2, haciendo más hincapié en las características más complejas o que han requerido un mayor esfuerzo a la hora de diseñarlas e implementarlas. Las características se han agrupado en bloques de características relacionadas.

4.3.1. Características generales

En este bloque se incluyen 2 características: la gestión automática del bucle del juego y la integración del motor con el ciclo de vida de las activities de Android.

4.3.1.1. Gestión del bucle del juego

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la gestión del bucle del juego.

4.3.1.1.1. Análisis

Tras hacer un repaso en la sección 1.3.5.2 de los bucles de juego más utilizados, se ha concluido que, de los tipos de bucle expuestos, el bucle de paso semi-fijo mejorado (explicado en el apartado 1.3.5.2.4) es el que permite una mejor gestión del flujo del juego. Como ya se ha comentado, existen bucles más complejos, pero, para un motor de juegos 2D como es DroidEngine2D, en principio con este tipo de bucle no debería haber ningún problema a la hora de implementar la mayoría de juegos.

Inicialmente, el bucle de paso semi-fijo mejorado parece una buena solución, sin embargo, debido al ciclo de vida de las activities de Android (explicado en el apartado 1.4.1), es conveniente pausar el hilo cuando la activity pasa a segundo plano, ya que no tiene mucho sentido consumir recursos del sistema actualizando el bucle del juego cuando esto ocurre. Por tanto, es necesario realizar unas pequeñas modificaciones al bucle de paso semi-fijo mejorado para que sea posible pausar el bucle cuando sea necesario.

En resumen, el bucle del juego debe poder: iniciarse, pausarse una vez iniciado, reanudarse tras la pausa y pararse para poder liberar recursos cuando se cierre la aplicación.

4.3.1.1.2. Diseño e implementación

Una cuestión a tener en cuenta es que, por defecto, todos los componentes de una aplicación para Android se ejecutan en el mismo hilo, el hilo principal de la aplicación (*main thread*).

El bucle del juego no puede ejecutarse en el hilo principal, puesto que al ser un bucle "infinito", bloquearía el hilo principal de la aplicación, provocando que ésta no respondiese. Por este motivo es necesario que el bucle del juego se ejecute en un hilo independiente.

Una segunda cuestión a tener en cuenta es que el contexto de OpenGL se gestiona por defecto en un hilo dedicado al rendering. Este hilo lo gestiona internamente la clase `GLSurfaceView` proporcionada por el SDK de Android y por defecto refresca a 60 FPS (*Frames Per Second*), lo cual fuerza a que cada frame deba actualizarse y renderizarse en 16 milisegundos aproximadamente.

No es posible implementar el tipo de bucle mencionado en el apartado anterior en el hilo del renderer, lo cual deja solo una opción: implementar el hilo del juego por un lado y utilizar el hilo del renderer únicamente para renderizar los elementos del juego. Esto supone tener que sincronizar ambos hilos de forma que si el hilo del juego está

actualizando los elementos del juego, el hilo del renderer debe esperar a que la actualización termine, y viceversa, para evitar una condición de carrera.

Actualmente (Julio, 2014) en DroidEngine2D se utilizan 2 hilos aparte del hilo principal de la aplicación. En el primer hilo se gestiona el bucle del juego y se actualiza la lógica del juego (clase `GameThread`). Y en el segundo hilo se realizan todas las llamadas a OpenGL y se renderizan los objetos del juego. Este hilo lo gestiona la clase `GLSurfaceView` proporcionada por el SDK de Android. A continuación se explica más detalladamente cómo funciona la sincronización entre ambos hilos.

Clase GameThread

La clase `GameThread` de DroidEngine2D implementa una versión modificada del bucle de juego de paso semi-fijo mejorado que se ha explicado anteriormente en el apartado 1.3.5.2.4. En este caso se ha agregado la posibilidad de pausar el hilo para poder bloquear el hilo sin destruirlo mientras la activity que contiene al juego está en segundo plano, en cuyo caso se pausa el motor y no se lleva a cabo ninguna actualización ni renderizado. Cuando la aplicación vuelve a primer plano, el hilo del juego se reanuda y los objetos del juego continúan en el mismo estado en el que estuvieran antes de la pausa. Esto se explicará más detalladamente cuando se explique la integración del motor con el ciclo de vida de la activity más adelante.

El diagrama de estados que ilustra su funcionamiento se expone a continuación.

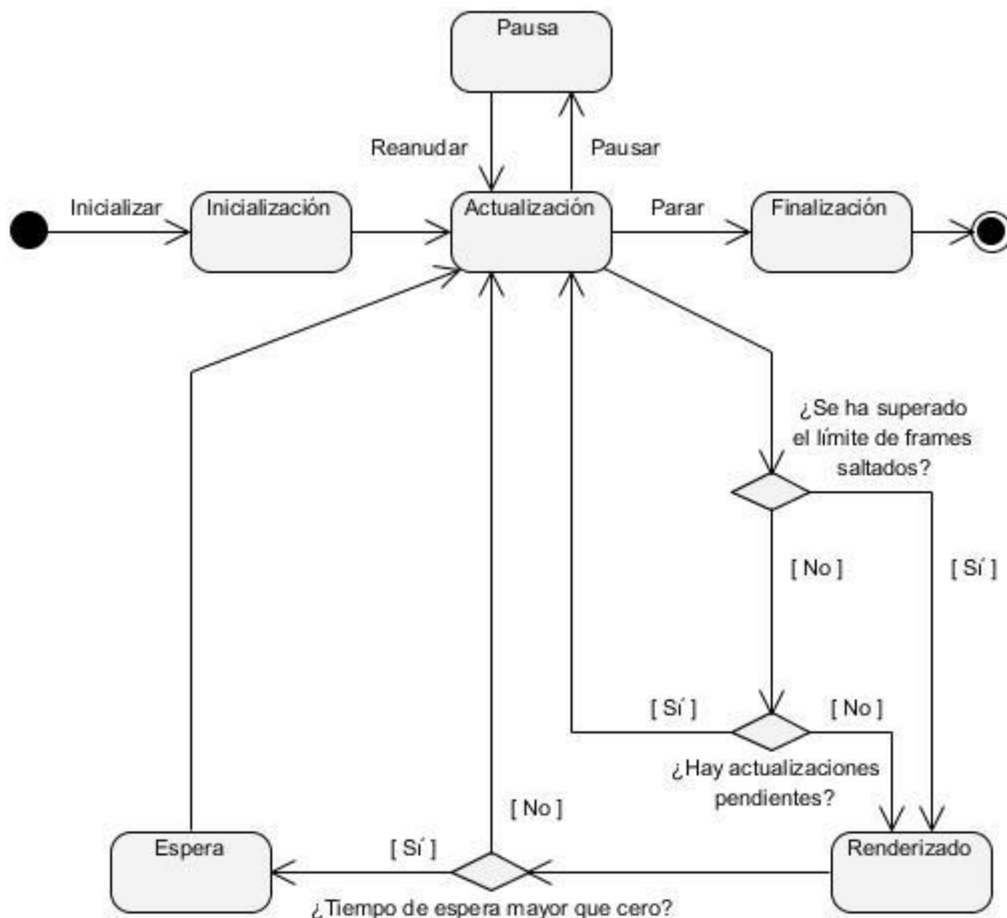

Figura 20. Diagrama de estados de un bucle de paso semi-fijo mejorado que permite ser pausado.

La actualización se realiza por defecto a 60 FPS, aunque es posible especificar otro valor en el momento en que se inicializa el motor. La única restricción es que sea un valor positivo que no supere la frecuencia de refresco de la pantalla del dispositivo.

Además, la clase `GameThread` proporciona los métodos `start()`, `pause()`, `resume()` y `terminate()`, que sirven para iniciar, pausar, reanudar y parar el hilo respectivamente.

Clase GLSurfaceView

La clase `GLSurfaceView` proporciona las siguientes características [59]:

- Gestiona una superficie de rendering que puede ser integrada en el sistema de views de Android.
- Gestiona la interfaz EGL, que es una interfaz entre las APIs de rendering de Khronos, como OpenGL ES, y el sistema de ventanas de la plataforma nativa. EGL gestiona el contexto gráfico, el enlace de buffers y la sincronización del rendering [60].
- Acepta un renderer proporcionado por el usuario.
- Realiza el rendering en un hilo independiente para evitar bloquear el hilo principal de la aplicación.
- Soporta 2 modos de rendering.
    - Rendering continuo, a 60 FPS.
    - Rendering bajo demanda.
- Proporciona métodos para depurar y obtener información acerca de errores en las llamadas a la API de OpenGL ES.

DroidEngine2D no trabaja con `GLSurfaceView` directamente, sino con `GLView`, una subclase de `GLSurfaceView` que actualmente (Julio, 2014) no agrega funcionalidad adicional, pero la documentación del SDK de Android recomienda crear una subclase en lugar de trabajar directamente con `GLSurfaceView`.

Sincronización entre el hilo del juego y el hilo del renderer

De los 2 modos de rendering soportados por `GLSurfaceView`, DroidEngine2D utiliza el rendering bajo demanda. Y la sincronización ente ambos hilos se lleva a cabo tal y como se explica a continuación.

Tras actualizar el juego, para pasar al estado de renderizado (ver el diagrama de la *Figura 20*), lo que hace `GameThread` es enviar una petición al hilo del renderer gestionado por `GLView` (subclase de `GLSurfaceView` proporcionada por DroidEngine2D) indicándole que ya puede renderizar los objetos del juego. Además, ambos hilos se sincronizan en cada frame de forma que no se pueda llevar a cabo una actualización durante el envío de datos de los objetos del juego a la GPU y no se pueda enviar datos de los objetos del juego a la GPU durante la actualización (exclusión mutua).

Como se puede extraer de la explicación anterior, la actualización y el envío de datos de los objetos del juego a la GPU, aunque se realizan en hilos distintos, se ejecutan de forma secuencial. La parte que se ejecuta de forma concurrente son las llamadas que realiza internamente `GLSurfaceView` para preparar la superficie de rendering y para intercambiar los buffers. En especial la llamada a `eglSwapBuffers()`, que bloquea el hilo del renderer durante varios milisegundos, especialmente en dispositivos menos potentes. Durante este tiempo en el que el hilo del renderer está bloqueado sí se permite actualizar el siguiente frame en el hilo del juego.

Métodos update y draw

Algunas clases, como las clases `AbstractGame` y `GameState`, poseen los métodos `update(float)` y `draw(Graphics)`. Estos métodos están pensados para que el usuario del motor pueda sobreescribirlos e implementar la actualización del juego en el método `update(float)` y el rendering en el método `draw(Graphics)`.

El método `draw(Graphics)` siempre se llamará desde el hilo del renderer, y por tanto, tiene acceso al contexto de OpenGL ES.

Ciclo de vida del juego

Una vez descrito el funcionamiento del bucle del juego anteriormente, en este apartado se expone el ciclo de vida del juego. Conocer el ciclo de vida del juego es fundamental para poder implementar un juego correctamente con DroidEngine2D.

En primer lugar, el usuario podrá definir un juego creando una subclase de `AbstractGame`, que dispone de una serie de métodos que se llaman automáticamente durante el ciclo de vida del juego. Estos métodos son `initialize()`, `update()`, `draw()`, `onEnginePaused()`, `onEngineResumed()` y `onEngineDestroyed()`.

La inicialización de los elementos del juego no debe llevarse a cabo en el constructor, sino en el método `initialize()`. El motivo de que la inicialización deba ir en dicho método es que en Android no se pueden obtener las dimensiones de un componente del layout de la activity hasta que dicho layout no se ha renderizado al menos una vez, y hasta que esto no ocurre no se ejecuta el método `initialize()` de `AbstractGame`. Por tanto, si se inicializan elementos del juego que necesiten acceder a las dimensiones del `GLView` antes de que se llame a `initialize()`, es posible que el `GLView` no esté inicializado del todo y sus dimensiones sean 0x0 píxeles.

Una vez inicializado, el juego estará en ejecución y los métodos `update()` y `draw()` se ejecutarán constantemente, cada uno en un hilo distinto, tal y como se ha descrito en apartados anteriores.

Si mientras el juego está en ejecución se pausa el bucle del juego, se llamará al método `onEnginePaused()` y al reanudarse el bucle del juego, se llamará a `onEngineResumed()` justo antes de reanudar la ejecución del juego.

Por último, si el bucle del juego termina, se llamará a `onEngineDestroyed()`.

Con todos estos callbacks, lo que se pretende es que el usuario del motor, al programar su juego pueda tener en cuenta estos eventos y gestionarlos de la forma que crea conveniente. Además, DroidEngine2D hace uso de algunos de estos métodos para tareas

como gestionar los estados del juego automáticamente, lo cual se explicará más adelante.

A continuación se expone un diagrama que describe el ciclo de vida del juego explicado anteriormente.

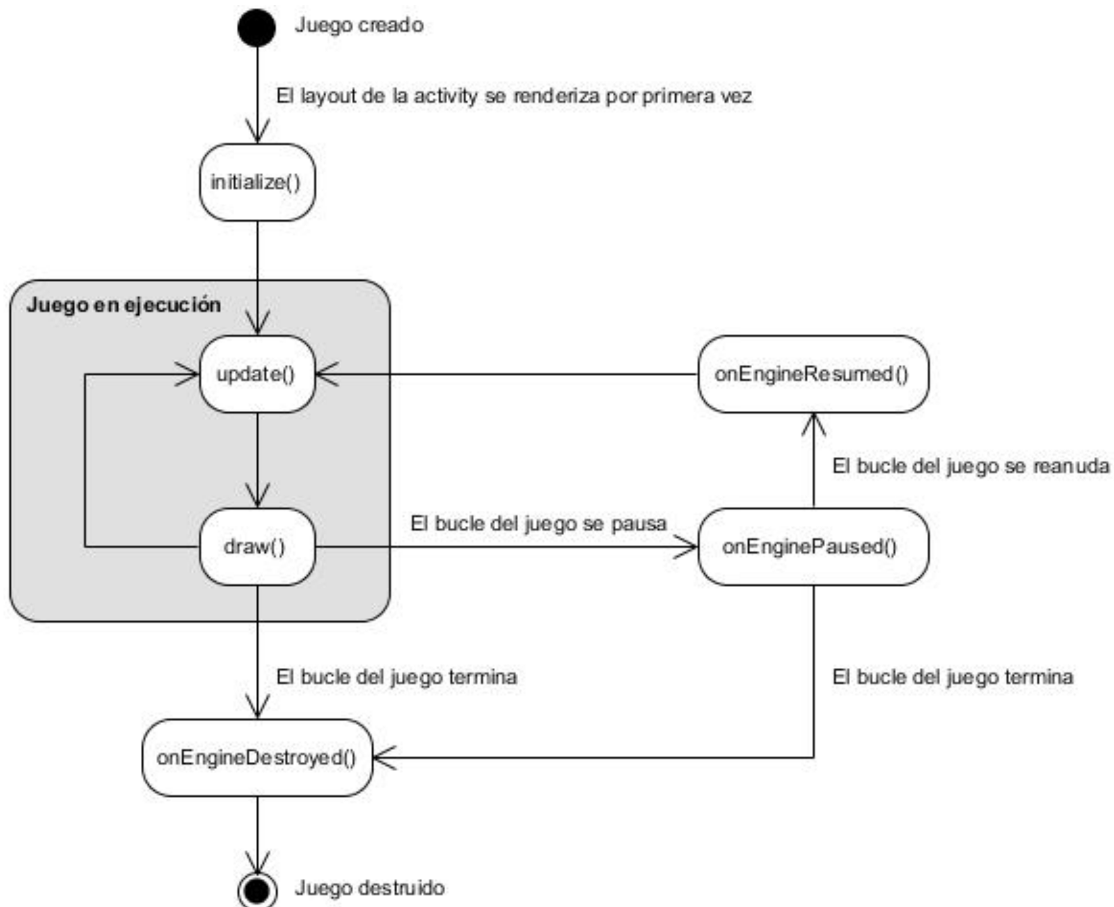

Figura 21. Ciclo de vida del juego.

4.3.1.1.3. Pruebas

Para verificar el funcionamiento correcto de la sincronización entre hilos se ha hecho uso de la clase `Log` proporcionada por el SDK de Android. Esta clase permite imprimir mensajes de depuración para comprobar que las llamadas se realizan en el orden correcto.

El comportamiento que se ha verificado es que las actualizaciones y los renderizados se ejecutan de forma alterna. Excepto si el bucle tiene que saltarse una o varias etapas de renderizado por cuestiones de rendimiento, como se explica en el apartado 1.3.5.2.4. En este caso se puede observar que si se salta, por ejemplo, una etapa de renderizado, se realizan 2 actualizaciones seguidas antes de la siguiente etapa de renderizado.

Además, con este mismo procedimiento se ha comprobado que el método `start()` inicia el bucle del juego, el método `pause()` pausa el hilo si estaba previamente corriendo, el método `resume()` reanuda el hilo si estaba pausado, y el método `terminate()` para el hilo y libera recursos.

4.3.1.2. Integración del motor con el ciclo de vida de las activities

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la integración del motor con el ciclo de vida de las activities de Android.

4.3.1.2.1. Análisis

DroidEngine2D debe integrarse con el ciclo de vida de las activities de Android (explicado en el apartado 1.4.1). La integración del motor con el ciclo de vida de la activity que lo contiene consiste en lo siguiente:

- Cuando se crea la activity, el motor se inicializa y pone en marcha el bucle del juego.
- Cuando se pausa la activity, el motor pausa el bucle del juego.
- Cuando se reanuda el funcionamiento de la activity tras la pausa, el motor reanuda el bucle del juego.
- Cuando se destruye activity, el motor para el bucle del juego y libera todos los recursos.

4.3.1.2.2. Diseño e implementación

Con el objetivo de cubrir los 4 casos expuestos en el apartado anterior, DroidEngine2D proporciona dos clases: `Engine` y `EngineActivity`.

Clase Engine

La clase `Engine` es la que representa al motor: se encarga de gestionar el hilo del juego y el hilo del renderer. De esta clase se pueden destacar los métodos `start()`, `pause()`, `resume()` y `destroy()`. Estos métodos permiten iniciar, pausar, reanudar y parar el motor liberando los recursos que tuviese reservados, respectivamente.

Una cuestión que ha habido que tener en cuenta es que al llamar al método `onPause()` de `GLSurfaceView` el hilo del renderer se destruye, y al llamar al método `onResume()` se crea un hilo nuevo con un contexto de OpenGL nuevo. Por tanto, es necesario recargar las texturas y los shaders al reanudar el funcionamiento tras la pausa, ya que al destruirse el contexto de OpenGL previo, todos los recursos asociados al mismo quedan liberados automáticamente.

Para integrar correctamente el motor con el ciclo de vida de la activity que lo contiene, es necesario crear una subclase de `Activity` que sobrescriba los métodos `onCreate()`, `onPause()`, `onResume()` y `onDestroy()`, de forma que desde estos métodos se llame a los métodos `engine.start()`, `engine.pause()`, `engine.resume()` y `engine.destroy()`, respectivamente.

En resumen, esta clase permite gestionar el ciclo de vida del hilo del juego (`GameThread`) y la superficie de rendering (`GLView`), de forma que, por ejemplo, al pausar el motor, se pausen ambos hilos, y al reanudarlo, se reanuden ambos.

Clase EngineActivity

Para evitar que el usuario del motor tenga que preocuparse de integrar el motor con el ciclo de vida de la activity de forma manual, DroidEngine2D proporciona también la clase `EngineActivity`, una subclase de `Activity` que permite al usuario especificar directamente los parámetros de inicialización del motor. Algunos ejemplos de parámetros de inicialización son la frecuencia máxima de refresco del juego, el máximo número de frames consecutivos en los que se puede actualizar sin renderizar en caso de que el rendimiento sea bajo, el renderer que se utilizará, etc.

Al utilizar `EngineActivity` en lugar de `Activity`, el usuario tan solo debe proporcionar un objeto tipo `Engine`, el identificador del layout que se desee utilizar para la activity (interfaz de la activity definida en XML), el identificador del `GLView` dentro de dicho layout, y la orientación en la que se representará el juego.

La configuración de la ventana de `EngineActivity` por defecto especifica que el juego se muestre a pantalla completa y que la pantalla no se apague aunque el usuario pase mucho tiempo sin interactuar con ella. No obstante, dicha configuración se puede modificar sobrescribiendo el método `setWindowFlags()`.

A continuación se muestra el diagrama de secuencia que ilustra la inicialización del motor cuando se llama al método `onCreate()` de `EngineActivity`.

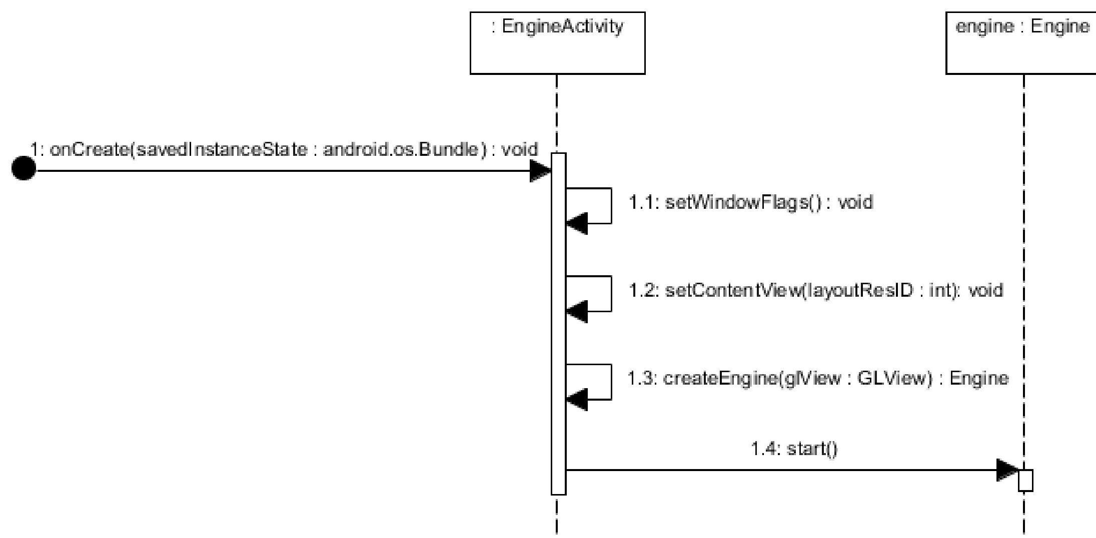

Figura 22. Diagrama de secuencia de la inicialización del motor utilizando EngineActivity.

Como se puede ver en la figura anterior, en el método `onCreate()` se configura la ventana, se asigna el layout que se va a utilizar para la interfaz y se crea el objeto `Engine`. Y por último se pone el motor en funcionamiento con la llamada al método `start()`.

Para los métodos `onPause()`, `onResume()` y `onDestroy()` no se proporciona diagrama de secuencia, puesto que simplemente llaman a `engine.pause()`, `engine.resume()` y `engine.destroy()`, respectivamente.

4.3.1.2.3. Pruebas

Para verificar que la integración del motor con el ciclo de vida de la activity, se ha utilizado, de nuevo, la clase `Log` proporcionada por el SDK de Android. Se colocaron mensajes de depuración en los métodos `onCreate()`, `onPause()`, `onResume()` y `onDestroy()`, de `EngineActivity` que, junto con los mensajes de depuración mencionados en las pruebas del bucle del juego, hacían posible comprobar que la integración era correcta.

Las pruebas se han efectuado de forma manual, ejecutando la aplicación, poniéndola en segundo plano, volviendo a ponerla en primer plano y, finalmente, cerrándola para asegurar que los recursos se liberan correctamente.

4.3.2. Audio

En este bloque se incluyen dos características: la gestión de pistas de audio y la gestión de efectos de sonido.

4.3.2.1. Gestión de pistas de audio

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la gestión de pistas de audio.

4.3.2.1.1. Análisis

Una característica bastante útil en un motor de juegos es la capacidad de reproducir pistas de audio desde archivos. Esto facilita al programador el uso de música de fondo en los videojuegos.

La gestión de pistas de audio, tal y como se ha definido para DroidEngine2D, consiste en lo siguiente:

- Reproducir pistas de audio desde archivos, sin cargarlas completamente en memoria.
- Pausar y reanudar la reproducción de pistas de audio.
- Parar la reproducción de pistas de audio.
- Modificar el volumen de reproducción.
- Reproducir la pista de audio una sola vez o en modo bucle.
- Comprobar si es posible reproducir un archivo especificado.

Además, hay que tener en cuenta que las pistas de audio se pueden especificar mediante una ruta relativa a la carpeta *assets* del proyecto Android o mediante una ruta dentro del sistema de archivos de Android.

4.3.2.1.2. Diseño e implementación

La gestión de pistas de audio se ha implementado en la clase `MusicPlayer`. Esta clase proporciona una serie de métodos que permiten realizar las acciones listadas en el apartado anterior.

Para implementar el reproductor de música, `MusicPlayer` utiliza internamente la clase `MediaPlayer`, que es la clase que proporciona el SDK de Android para reproducir archivos de audio y video directamente desde archivos, sin cargar el archivo completo en memoria [72].

La ventaja de utilizar `MusicPlayer` frente a utilizar directamente `MediaPlayer` es que `MusicPlayer` encapsula la gestión del estado del reproductor y facilita la comprobación de compatibilidad de archivos, entre otras cosas. Por tanto, permite al usuario realizar las mismas acciones, pero escribiendo menos líneas de código.

4.3.2.1.3. Pruebas

Para probar la gestión de pistas de audio, se ha implementado una demo que carga una pista de audio y permite realizar varias acciones básicas, como reproducirla, pausarla y aumentar y reducir el volumen. Esta demo carga la pista de audio de la carpeta *assets*.

La carga de pistas de audio desde rutas externas al proyecto es una funcionalidad que se desarrolló inicialmente porque era necesaria para AlarMind, que como ya se ha mencionado previamente, es una aplicación que cuenta con varios mini-juegos desarrollados con DroidEngine2D. Las pruebas de esta funcionalidad concreta se realizaron de forma manual durante el desarrollo de dicho proyecto.

4.3.2.2. Gestión de efectos de sonido

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la gestión de efectos de sonido.

4.3.2.2.1. Análisis

Como ya se ha explicado en el primer capítulo, los efectos de sonido, a diferencia de las pistas de audio son sonidos de muy corta duración. Por ejemplo, el sonido de un disparo, el sonido que se reproduce al recoger monedas, el sonido que se reproduce al pulsar un botón, etc.

Estos sonidos, al contrario que en el caso de la música de fondo, sí tiene sentido cargarlos en memoria. Principalmente porque se suelen reproducir con bastante frecuencia (por ejemplo, en el caso de sonidos de disparos). Cargar los sonidos desde disco cada vez que se necesita reproducirlos no es lo suficientemente eficiente. Además, al ser sonidos cortos no ocupan mucha memoria.

La gestión de efectos de sonido, tal y como se ha definido en DroidEngine2D, consiste en lo siguiente:

- Cargar efectos de sonido en memoria y almacenarlos indexados por un identificador especificado.
- Reproducir efectos de sonido previamente almacenados en memoria.
- Liberar recursos de forma que no quede ningún efecto de sonido cargado en memoria.

4.3.2.2.2. Diseño e implementación

La gestión de efectos de sonido se ha implementado en la clase `SoundManager`. Esta clase proporciona una serie de métodos que permiten realizar las acciones listadas en el apartado anterior.

La clase `SoundManager` utiliza internamente la clase `SoundPool`, proporcionada por el SDK de Android. Esta clase gestiona en memoria y reproduce efectos de sonido [73].

La clase `SoundPool`, al cargar un sonido desde un archivo en memoria, devuelve un identificador de dicho sonido, un entero. Sin embargo, por comodidad, DroidEngine2D utiliza como identificador la ruta del archivo de audio, por tanto es necesario que `SoundManager` gestione además un mapa que indexe los sonidos por ruta.

Para reproducir un sonido previamente cargado en memoria, tan solo hay que especificar la ruta de dicho archivo, y si está cargado en memoria, `SoundManager` lo reproducirá. La ruta debe ser una ruta relativa a la carpeta *assets*. No se pueden cargar archivos que no se encuentren en dicha carpeta.

En cuanto a la liberación de recursos, es posible liberar los recursos que ocupa un sonido concreto, o bien liberar todos los recursos que ocupan todos los sonidos registrados en el `SoundManager`.

4.3.2.2.3. Pruebas

Para probar la gestión de efectos de sonido, se ha implementado una demo que permite reproducir varios sonidos de la carpeta *assets*. De esta forma se puede verificar que los sonidos se reproducen correctamente

4.3.3. Estados del juego

En este bloque se explica la gestión de estados del juego, que consiste en poder definir estados, registrarlos en un registro de estados y activarlos o desactivarlos cuando el usuario desee.

4.3.3.1. Gestión de estados del juego

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la gestión de estados del juego.

4.3.3.1.1. Análisis

Una característica bastante útil de DroidEngine2D es la gestión de estados del juego. Para ello, es necesario, en primer lugar, que el usuario pueda definir estados del juego con facilidad.

Un estado del juego se puede ver como una pantalla del juego, aunque no tiene por qué ocupar la pantalla completa. Puede ser una ventana flotante que ocupa una región de la pantalla del dispositivo. Algunos ejemplos de estados del juego son el menú principal, un nivel concreto del juego, una pantalla de pausa o una pantalla de carga que se muestra mientras se inicializa un nivel, entre otros.

Los estados del juego deben integrarse con el ciclo de vida del juego. También deben poder cargarse en un registro de estados disponibles, de forma que cambiar de un estado a otro sea rápido al estar ambos cargados en memoria. Y deben poder activarse y desactivarse, de forma que solo los estados activos se muestren en pantalla, y solo en último estado que se ha activado se actualice.

4.3.3.1.2. Diseño e implementación

Para definir un estado del juego, el usuario debe crear una subclase de la clase `GameState` o de la clase `GameStateAdapter`.

Los estados del juego deben crearse en el método `initialize()` del juego que esté implementando el usuario, que será una subclase de `AbstractGame`. El motivo de esta decisión es que antes de llamar a este método no se puede asegurar que las dimensiones del `GLView` devueltas por los métodos `getWidth()` y `getHeight()` sean correctas, puesto que hasta que Android no haya renderizado el layout de la activity al menos una vez no es posible acceder a las dimensiones de los elementos de dicho layout.

Tras crear un estado, éste debe ser registrado en el `GameStateManager` del juego, y posteriormente podrá ser activado o desactivado cuando el usuario desee.

A continuación se explican las principales características y el funcionamiento de las clases `GameState`, `GameStateAdapter` y `GameStateManager`.

Clase GameState

La clase `GameState` proporciona un conjunto de métodos abstractos que el usuario del motor deberá implementar para crear sus propios estados del juego. En la siguiente tabla se recogen dichos métodos, junto con una explicación breve del objetivo de cada uno.

Método	Descripción
`update(float)`	En este método se debe implementar la actualización de la lógica del juego (por ejemplo, actualizar la posición de los objetos). Se llama periódicamente, una vez por frame, si el `GameState` está en la cima de la pila de estados activos del `GameStateManager`.
`draw(Graphics)`	En este método se deben ejecutar las funciones que permiten representar los elementos del juego en pantalla. Se llama periódicamente, una vez por frame, si el `GameState` está en la pila de estados activos del `GameStateManager`.
`onRegister()`	Este método se llama cuando el `GameState` se registra en el `GameStateManager`. Es un buen lugar para llevar a cabo la inicialización del estado del juego.
`onActivation()`	Este método se llama cuando el `GameState` es activado, es decir, cuando se introduce en la pila de estados activos.

onDeactivation()	Este método se llama cuando el GameState es desactivado, es decir, cuando se elimina de la pila de estados activos del GameStateManager.
onDispose()	Este método se llama cuando se destruye el GameState o cuando se elimina del registro del GameStateManager. Es un bien lugar para liberar recursos.
onPause()	Este método se llama cuando se pausa el GameStateManager, es decir, cuando se pausa el bucle del juego.
onResume()	Este método se llama cuando se reanuda el funcionamiento del GameStateManager tras una pausa, es decir, cuando se reanuda el bucle del juego tras una pausa.

Tabla 2. Métodos más relevantes de la clase GameState.

Clase GameStateAdapter

La clase GameStateAdapter es simplemente una subclase de GameState que proporciona una implementación vacía de los métodos abstractos de GameState y, por tanto, no hace nada por defecto.

El objetivo de GameStateAdapter es que el usuario pueda sobrescribir únicamente los métodos que necesite, reduciendo así el número de líneas de código que necesita escribir para implementar el juego.

Clase GameStateManager

La clase GameStateManager, como su propio nombre indica, se encarga de gestionar los estados del juego.

En primer lugar, el usuario debe registrar los estados del juego en el GameStateManager, proporcionando un identificador único para cada estado que permita acceder a dicho estado más adelante. Un buen lugar para registrar los estados del juego en el GameStateManager es el método initialize() de AbstractGame.

Para registrar un estado en el GameStateManager se puede utilizar el método registerGameState(int, GameState), que permite registrar un estado del juego con el identificador especificado; o bien registerGameState(int, GameState, boolean), que permite además especificar si se desea activar inmediatamente el estado que está siendo registrado.

Una vez registrados los estados del juego con un identificador único para cada estado, es posible eliminarlos del registro mediante el método unregisterGameState(int), especificando el identificador del estado que se desee eliminar.

En cuanto a la activación y desactivación de estados, se ha implementado haciendo uso de una pila. A esta pila se hace referencia como la pila de estados activos.

El funcionamiento de la pila de estados activos es muy simple: todos los estados que se encuentren en la pila serán representados en pantalla, en orden de inserción en la pila, y solo el estado que se encuentra en la cima de la pila será actualizado.

Para interactuar con la pila de estados activos, `GameStateManager` proporciona varios métodos que permiten al usuario realizar las operaciones básicas que necesita. Los métodos esenciales para activar y desactivar estados son los siguientes:

Método	**Descripción**
`pushActiveGameState(int)`	Inserta un `GameState`, previamente registrado en el `GameStateManager` con el identificador especificado, en la pila de estados activos.
`popActiveGameState()`	Elimina el `GameState` que se encuentra en la cima de la pila de estados activos y lo devuelve.
`peekActiveGameState()`	Devuelve el `GameState` que se encuentra en la cima de la pila de estados activos, eliminarlo de la misma.
`switchActiveGameState(int)`	Sustituye el `GameState` que se encuentra en la cima de la pila de estados activos por el `GameState` especificado mediante su identificador. Esta operación es equivalente a ejecutar `popActiveGameState()` y posteriormente `pushActiveGameState(int)`.

Tabla 3. Métodos básicos para la activación y desactivación de estados del juego.

En la tabla anterior se exponen las operaciones básicas. La clase `GameStateManager` proporciona además otros métodos que permiten desactivar varios estados a la vez, desactivar directamente todos los estados activos, etc.

También se ha implementado la posibilidad de registrar un listener para poder escuchar cuándo se produce un cambio en la pila de estados activos. Esta funcionalidad la utiliza internamente DroidEngine2D, aunque también puede ser útil al usuario en caso de que necesite dicha información.

Lo normal es que la pila solo contenga un estado activo, sin embargo, en algunas ocasiones es conveniente activar dos o más estados a la vez. Un ejemplo de esto puede ser una ventana flotante que indique el fin de la partida. En DroidEngine2D, este comportamiento se puede conseguir simplemente insertando en la pila de estados activos un `GameState` que represente la ventana de fin de partida correspondiente. La siguiente figura ilustra la pila de estados activos para este caso concreto que se acaba de describir. Como se puede ver, inicialmente había un estado que podría ser un nivel del juego, y sobre ese estado se inserta otro estado que representa la ventana flotante que indica al usuario que ha perdido, y oscurece levemente el resto de la pantalla, dejando visible el estado anterior.

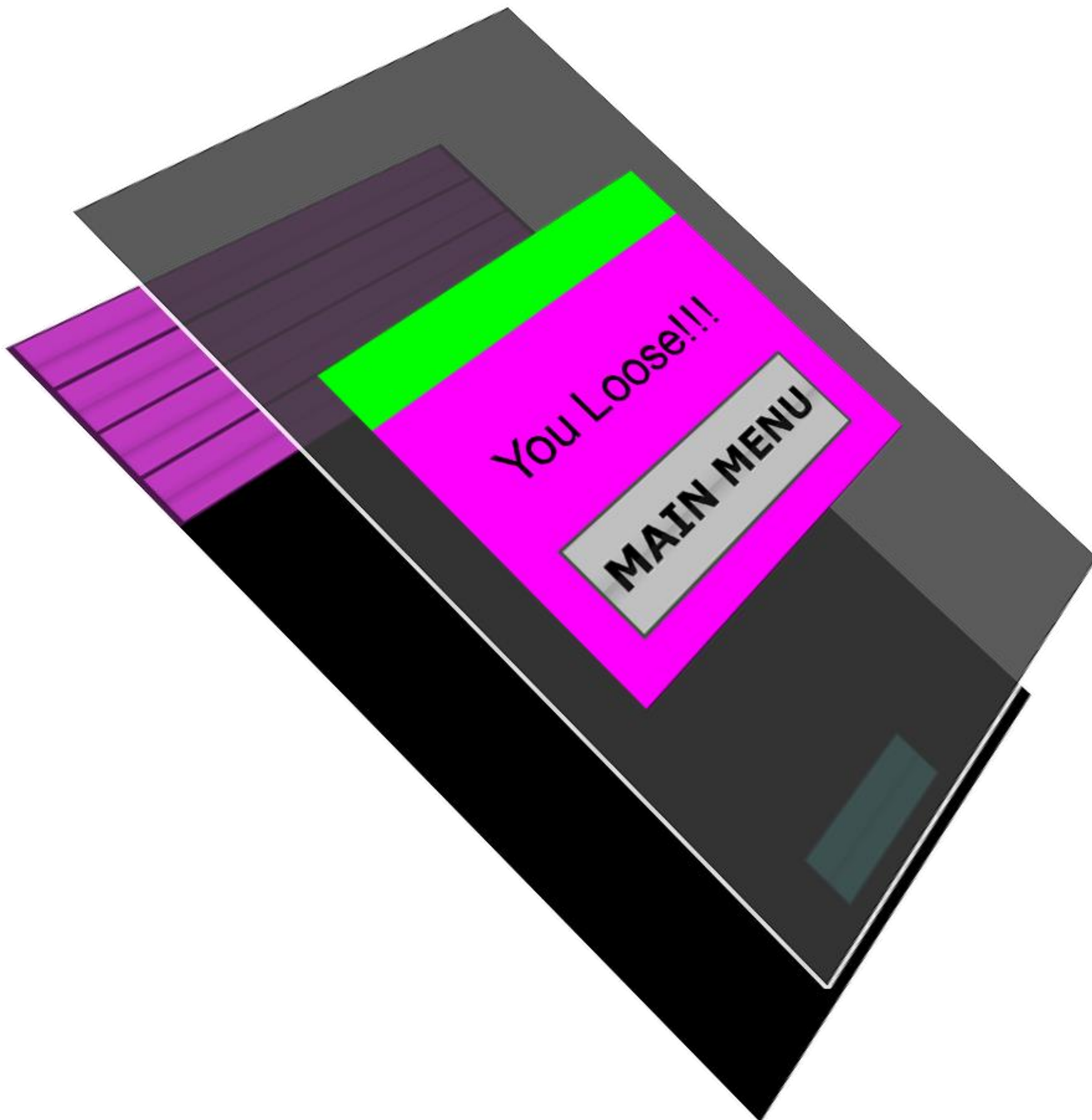

Figura 23. Ejemplo de funcionamiento de la pila de estados activos.

4.3.3.1.3. Pruebas

Para verificar que la gestión de estados del juego funciona correctamente se han realizado principalmente dos demos.

En la primera de ellas se puede comprobar cómo funciona la sustitución del estado que se encuentra en la cima de la pila por otro estado, que es la operación que el usuario utilizará con mayor frecuencia.

Y en la segunda demo, además, se muestra cómo se apilan dos estados, que consiste básicamente en lo que se ha descrito en el ejemplo del apartado anterior.

Además, en la demo en la que se incluyen las principales características del motor, que consiste en una versión muy simple del juego *Breakout*, también se incluye la gestión de estados.

4.3.4. Gráficos

En este bloque se explican las principales características relacionadas con el apartado gráfico de DroidEngine2D, desde definición de animaciones hasta representación de elementos en pantalla, gestión de texturas, etc.

4.3.4.1. Animaciones

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la definición y actualización de animaciones.

4.3.4.1.1. Análisis

En los videojuegos, es común tener elementos animados. Por tanto, una característica que puede llegar a ser bastante útil en un motor de juegos es la posibilidad de que el usuario defina animaciones.

A la hora de trabajar con animaciones en DroidEngine2D, el usuario debe poder:

- Reproducir animaciones.
- Pausar y reanudar animaciones.
- Reiniciar animaciones.
- Reproducir las animaciones una sola vez o en modo bucle.
- Definir callbacks que se llaman cuando ocurren determinados eventos en una animación. Los eventos a tener en cuenta son los siguientes:
    - La animación comienza a reproducirse.
    - La animación se pausa.
    - La animación se reanuda tras la pausa.
    - La animación cambia de frame.
    - La animación completa un ciclo.
    - La animación termina (esto solo ocurre si el modo bucle no está activado).

4.3.4.1.2. Diseño e implementación

En DroidEngine2D, una animación se puede definir haciendo uso de las clases `Animation` y `AnimationFrame`. Y para definir los callbacks mencionados anteriormente, se puede utilizar la clase `AnimationStateListener` o la clase `AnimationStateAdapter`.

Clase AnimationFrame

La clase `AnimationFrame`, como su propio nombre indica, representa un frame, o etapa, de una animación. Un frame de una animación consiste en una imagen y el tiempo, en milisegundos, que se muestra dicha imagen.

Clase Animation

La clase `Animation` permite definir animaciones. Una animación consiste en una lista de objetos tipo `AnimationFrame`. Al pasar el tiempo se va actualizando de un frame al siguiente. Por ejemplo, una animación que dura un segundo podría consistir en 5 frames que se muestren 200 milisegundos cada uno, o bien 2 frames que se muestren 350 milisegundos cada uno, y 3 frames que se muestren 100 milisegundos cada uno. Como se puede ver, al poder definir el tiempo de duración de cada frame en el propio frame, el usuario tiene mayor libertad a la hora de definir las animaciones que necesite.

Las animaciones en DroidEngine2D tienen una opción denominada modo bucle. Este modo, si está activo, permite que la animación comience a reproducirse desde el principio tras haber terminado un ciclo.

Para actualizar la animación, es necesario llamar al método `update(float)` en cada frame. Este método se encarga de comprobar si ha transcurrido el tiempo suficiente como para pasar al siguiente frame y, en caso de que la animación llegue a su fin, la reiniciará si el modo bucle está activado.

Además, las animaciones pueden reiniciarse, pausarse y reanudarse cuando el usuario desee. En el siguiente diagrama de estados se puede ver el funcionamiento de la clase `Animation` más detalladamente.

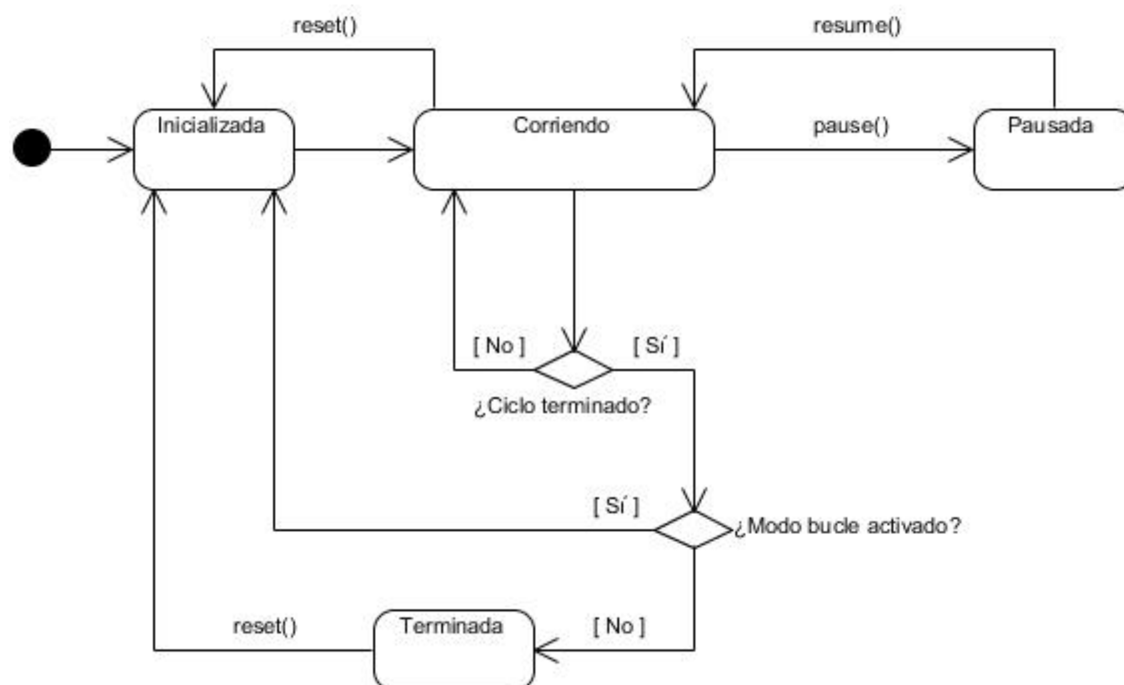

Figura 24. Ciclo de vida de una animación.

Interfaz AnimationStateListener

La interfaz `AnimationStateListener` define un conjunto de métodos que se llaman cuando ocurren determinados eventos en la clase `Animation`. Para escuchar estos eventos, es necesario registrar un objeto que implemente esta interfaz en el objeto tipo `Animation` cuyos eventos se desee escuchar.

Los métodos definidos en `AnimationStateListener` son los siguientes:

Método	Descripción
`onAnimationStarted(Animation)`	Este método se llama cuando la animación comienza a reproducirse.
`onAnimationPaused(Animation)`	Este método se llama cuando la animación se pausa.
`onAnimationResumed(Animation)`	Este método se llama cuando la animación se reanuda tras la pausa.
`onAnimationFrameChanged(Animation)`	Este método se llama cuando la animación cambia de frame.
`onAnimationLoopFinished(Animation)`	Este método se llama cuando un ciclo de la animación se completa.
`onAnimationFinished(Animation)`	Este método se llama cuando la animación termina de reproducirse. Una animación solo puede terminar de reproducirse si el modo bucle está desactivado.

Tabla 4. Métodos de la interfaz AnimationStateListener.

Clase AnimationStateAdapter

La clase `AnimationStateAdapter` es simplemente una clase que implementa `AnimationStateListener` y proporciona una implementación vacía de todos sus métodos.

El objetivo de `AnimationStateAdapter` es que el usuario pueda sobrescribir únicamente los métodos que necesite, reduciendo así el número de líneas de código que necesita escribir para implementar el juego.

4.3.4.1.3. Pruebas

Para verificar el funcionamiento de las animaciones se han implementado dos demos: una con el modo bucle activado y otra con el modo bucle desactivado.

La demo que tiene el modo bucle desactivado, además incluye un botón para reiniciar la animación para poder comprobar que, efectivamente, tras reiniciarla, solo se realiza un ciclo de animación.

Para probar la pausa, la reanudación y los callbacks se han utilizado mensajes de depuración impresos utilizando la clase `Log` proporcionada por el SDK de Android.

4.3.4.2. Representación de rectángulos utilizando materiales

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la representación de rectángulos utilizando materiales.

4.3.4.2.1. Análisis

Una característica prácticamente indispensable en un motor de juegos es que permita representar elementos en pantalla de distintas formas o con distintos efectos. DroidEngine2D, tal y como se ha comentado anteriormente, permite representar rectángulos utilizando diferentes materiales. A continuación se explica qué son los materiales y cómo se utilizan en DroidEngine2D.

En primer lugar, es necesario recordar que para poder representar un rectángulo en pantalla con OpenGL es necesario definir los vértices de dicho rectángulo y enviarlos a la GPU para que sean procesados por los shaders. Además, a la GPU hay que enviar otros datos como el color del rectángulo o, en caso de querer representar una imagen, la textura que se aplicará al rectángulo.

El concepto de "material" es básicamente lo que se describe en el párrafo anterior, un conjunto de datos que definen el aspecto (material) del objeto.

El principal objetivo de poder definir y utilizar materiales en DroidEngine2D es que el usuario no tenga que interactuar directamente con los shaders para representar un objeto con un aspecto determinado. En lugar de eso, el usuario puede definir un material con las propiedades que desee y aplicar dicho material a un rectángulo. Además, DroidEngine2D proporciona varios materiales por defecto, lo cual permite a los usuarios que no tengan conocimientos de OpenGL y GLSL utilizar DroidEngine2D igualmente.

Los materiales que proporciona DroidEngine2D por defecto son los siguientes:

- Color.
- Textura.
- Textura con control de opacidad.
- Textura con superposición de color RGBA.
- Textura a la que se le puede alterar el tono, la saturación y el brillo.

Todos los materiales permiten además modificar las opciones de fusión, es decir, permiten configurar la ecuación de blending y los parámetros de dicha ecuación. La configuración por defecto permite representar texturas con transparencia, dado que es la configuración que más se utiliza en juegos 2D y normalmente no será necesario modificarla. No obstante, si se desean crear efectos más elaborados, como superposición de colores, tan solo hay que especificar la ecuación de blending a utilizar y sus parámetros. Esto se ha explicado anteriormente de forma resumida en el apartado 3.5.1.9 y, tal y como se comenta en dicho apartado, se puede encontrar una explicación mucho más detallada de cada una de las ecuaciones y parámetros en las referencias [74], [75] y [76].

Como ya se ha comentado, los materiales son solo datos que especifican cómo se mostrará un determinado objeto (en el caso de DroidEngine2D, un rectángulo) en pantalla. Por tanto, es necesario, además de definir los materiales, crear un mecanismo que permita renderizar rectángulos utilizando un material concreto, es decir, que obtenga los datos de la geometría (vértices de los rectángulos) y los datos de los materiales y los envíe a la GPU para que puedan ser renderizados.

Por último mencionar que uno de los objetivos iniciales del proyecto era implementar un renderer eficiente que hiciese uso de batch rendering. Esto se ha tenido en cuenta y se ha implementado esta técnica para poder representar múltiples rectángulos una sola llamada a la función `glDrawElements` de OpenGL en los renderers de materiales.

4.3.4.2.2. Diseño e implementación

En esta sección se explicarán, en primer lugar, cómo se han definido las opciones de fusión, como crear un material personalizado y cómo se han implementado los materiales por defecto.

Y, posteriormente, se explicará cómo se han definido los mecanismos que permiten al usuario utilizar materiales sin tener que gestionar manualmente el envío de datos a la GPU con la API de OpenGL ES. A estos mecanismos que permiten representar elementos con un material especificado se hará referencia como "renderers de materiales".

Definición de opciones de fusión

La clase `BlendingOptions` representa las opciones de fusión que puede definir el usuario para que sean aplicadas a los objetos que pretenda representar en pantalla.

Permite definir básicamente tres cosas: la ecuación de blending, el factor fuente y el factor destino. Estos parámetros se han explicado anteriormente en el apartado 3.5.1.9.

Hay que destacar que estos parámetros se especifican haciendo uso de las constantes que proporciona OpenGL ES 2.0 directamente, en lugar que proporcionar constantes en DroidEngine2D para ello. Esta decisión se debe a que las constantes de OpenGL ES 2.0 son bien conocidas y existe bastante documentación explicando todas y cada una de ellas. Esta documentación se puede encontrar por ejemplo en las referencias [74], [75] y [76], como ya se ha mencionado anteriormente.

Los objetos `BlendingOptions` tienen por defecto la configuración necesaria para representar texturas con transparencia, por lo que para la mayoría de los casos el usuario no necesitará modificarlos. No obstante, si deseara representar otros efectos, como por ejemplo la superposición de los colores de una imagen sobre otra, sería necesario que conociera el funcionamiento del blending en OpenGL, lo cual está bien documentado en las referencias citadas en el párrafo anterior.

Creación de materiales personalizados

La creación de un material personalizado en DroidEngine2D es bastante simple, tan solo hay que crear una subclase de la clase `Material`.

La clase `Material` posee un objeto `BlendingOptions`, por lo que cada instancia de un material puede estar configurada con distintas opciones de fusión.

Más adelante se explicará cómo hacer que DroidEngine2D sea capaz de representar un objeto con un material concreto.

Materiales definidos por defecto

En la siguiente tabla se exponen los materiales que proporciona DroidEngine2D por defecto.

Clase	Descripción
`ColorMaterial`	Material que solo permite especificar un color.
`TextureMaterial`	Material que solo permite especificar una textura.
`TransparentTextureMaterial`	Material que permite especificar una textura y la opacidad de dicha textura. A menor opacidad, más transparente será el objeto.
`TextureColorMaterial`	Material que permite especificar una textura y un color en formato RGBA. El color de cada téxel (píxel de textura) de la textura se multiplica por el color especificado.
`TextureHsvMaterial`	Material que permite especificar una textura y un color en formato HSVA. El color de cada téxel de la textura se pasa a HSV. A la componente H se le suma la H del color especificado, y las componentes S y V se multiplican por la S y la V del valor especificado, respectivamente. La componente alfa (A) se aplica igual que en los 2 materiales anteriores.

Tabla 5. Materiales definidos por defecto en DroidEngine2D.

Como se puede ver, en la tabla anterior se ha explicado cómo se utilizan los datos especificados en cada uno de los materiales. Sin embargo, es conveniente recordar que los materiales son solo datos, los encargados de utilizarlos son los renderers de materiales.

Uso de atlas de texturas y batch rendering en DroidEngine2D

Antes de explicar los renderers de materiales, es necesario explicar en qué consiste el batch rendering y cómo se ha implementado en DroidEngine2D, y a que los renderers de materiales de DroidEngine2D están pensados para funcionar utilizando esta optimización. Se comentarán también las principales limitaciones que se han encontrado durante la implementación de esta técnica. Y, en cuanto a los atlas de texturas, en este apartado se mencionará lo necesario para entender el funcionamiento del batch rendering. La definición y gestión atlas de texturas se explicará más adelante.

En primer lugar, mencionar que un atlas de texturas (texture atlas) es una imagen compuesta por un conjunto de imágenes que se pueden utilizar de forma individual. Un ejemplo de atlas de texturas es el siguiente:

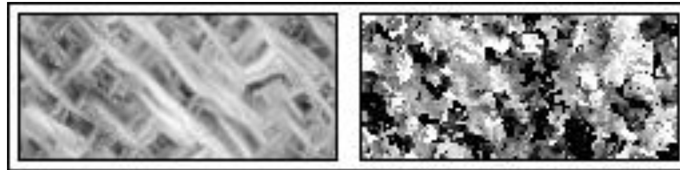

Figura 25. Ejemplo de atlas de texturas compuesto por dos texturas.

El atlas de texturas de la figura anterior es una imagen de 256x64 píxeles que contiene dos texturas de 120x56 píxeles cada una (los bordes están marcados en negro para facilitar la comprensión del ejemplo). La primera textura se encuentra en la posición (4, 4) y la segunda textura está en la posición (132, 4) del atlas de texturas.

El uso de atlas de texturas en DroidEngine2D supone una optimización frente a usar texturas independientes puesto que para poder representar un rectángulo con una determinada textura en OpenGL es necesario enlazar primero la textura al contexto de OpenGL para que sea posible acceder a ella desde los shaders. Para ello se utiliza la función `glBindTexture`, que debe ser llamada antes de `glDrawElements`.

Como se ha expuesto en el párrafo anterior, solo la textura enlazada antes de la llamada a glDrawElements estará disponible en los shaders, por lo que si el

El caso que se ha expuesto en el apartado de análisis es que es necesario que DroidEngine2D pueda representar múltiples rectángulos en una sola llamada a `glDrawElements`. Sin embargo, solo la última textura enlazada mediante `glBindTexture` antes de la llamada a `glDrawElements` será accesible desde los shaders. Esto supone inicialmente un problema a la hora de representar múltiples elementos en una llamada a `glDrawElements`, puesto que solo se podrían agrupar elementos que utilizasen la última textura enlazada.

Al usar atlas de texturas es posible enlazar al contexto de OpenGL una sola textura que está compuesta por varias texturas y acceder a las texturas que la componen desde los shaders, lo cual permite representar en una sola llamada a `glDrawElements` un conjunto de elementos (batch rendering) con texturas distintas, siempre que las texturas formen parte del mismo atlas de texturas. Además, esto permite reducir el número de llamadas a `glBindTexture`, ya que al utilizar un atlas de texturas de N texturas solo se realiza una llamada a esta función, mientras que al utilizar las N texturas por separado serían necesarias N llamadas. Por ejemplo, si se quisieran renderizar 10 rectángulos con 10 texturas diferentes, habría que realizar 10 llamadas a `glBindTexture` y 10 llamadas a `glDrawElements`, mientras que utilizando un atlas de texturas compuesto por las 10 texturas sería posible realizar una sola llamada a `glBindTexture` y una llamada a `glDrawElements` (utilizando batch rendering) para representar los mismos 10 rectángulos. En este ejemplo se reduciría un 90% el número de llamadas a estas funciones de OpenGL, que normalmente son bastante costosas.

Tras exponer las principales ventajas de utilizar atlas de texturas y batch rendering, a continuación se explica en mayor detalle la implementación de esta técnica y las

principales limitaciones que se han encontrado a la hora de llevar a cabo dicha implementación.

En lo que a atlas de texturas se refiere, hay que tener en cuenta que aunque, teóricamente, al poner todas las texturas del juego en un atlas de texturas solo sería necesario hacer una llamada a `glBindTexture`, en la práctica hay limitaciones con respecto a las dimensiones máximas que una textura puede tener, y estas dimensiones a veces varían de un dispositivo a otro. Por ejemplo, en los dispositivos Android más antiguos, las mayores dimensiones de textura que se pueden utilizar con seguridad son 1024x1024 píxeles, por lo que si un desarrollador está interesado en dar soporte a dispositivos antiguos deberá limitarse a utilizar atlas de texturas de dichas dimensiones o más pequeños.

Otra cuestión a tener en cuenta al implementar el batch rendering es que para poder representar N rectángulos situados en distintas posiciones, con distinta escala y/o rotación, es necesario especificar N matrices de transformación homogénea, una para cada rectángulo. Estas matrices se utilizan en el vertex shader para calcular las posiciones de los vértices, tal y como se ha explicado detalladamente en el capítulo 3. Esto puede no parecer un problema, sin embargo GLSL ES 1.00 [47] especifica el almacenamiento disponible en memoria para las variables tipo `varying` y `uniform` (explicadas en el apartado 3.5.2.1), en términos de vectores de 4 elementos tipo `float`. En la última página de dicha especificación se pueden ver las constantes que definen los valores mínimos que toda implementación de OpenGL ES 2.0 debe soportar para cada tipo de variable. En el caso de DroidEngine2D, al implementar el batch rendering, la variable que se utiliza para almacenar las matrices de transformación homogénea en el shader es un array de elementos tipo `mat4` (matrices de 4x4 elementos tipo `float`) [77], definido como variable tipo `uniform.` Por tanto, de las constantes definidas en la especificación de GLSL ES 1.00, la que supone una limitación es la siguiente:

```
gl_MaxVertexUniformVectors = 128
```

Esta constante garantiza que todo dispositivo que soporte OpenGL ES 2.0 permitirá almacenar como mínimo 128 vectores de 4 elementos tipo `float` definidos como `uniform`.

En la implementación de batch rendering de DroidEngine2D, como ya se ha comentado, se utiliza un array de matrices de 4x4 elementos, y GLSL almacena las matrices de 4x4 elementos como 4 vectores de 4 elementos, por lo que el máximo número de matrices que se pueden enviar al vertex shader en una llamada viene dado por la siguiente expresión:

$$\frac{gl_MaxVertexUniformVectors}{numero\ de\ vectores\ por\ matriz} = \frac{128}{4} = 32$$

Por este motivo, en DroidEngine2D se pueden representar, como máximo, 32 rectángulos en una sola llamada a la función `glDrawElements` de OpenGL.

Otra limitación que ha habido que tener en cuenta es que los shaders se compilan juntos (vertex shader y fragment shader) formando un shader program, y para poder utilizar este programa hay que llamar a la función `glUseProgram`. Y solo un shader program puede estar activo a la vez, lo cual hace imposible incluir en un mismo paquete rectángulos con textura y rectángulos con un color plano, ya que se renderizan

utilizando shaders distintos. Por tanto, se quisieran representar 2 rectángulos, uno con color plano y otro con textura, habría que hacer forzosamente 2 llamadas a `glDrawElements`, y entre ambas llamadas habría que cambiar el shader program que está en uso. En este caso concreto no se aprovecharían las ventajas del batch rendering.

Por último, mencionar que para aprovechar las ventajas de esta optimización es recomendable representar seguidos todos los elementos que utilicen el mismo shader (o el mismo material) y que, en caso de que los elementos utilicen texturas, éstas deberían estar definidas en el mismo texture atlas, en la medida de lo posible.

Shader programs

Un shader program en OpenGL ES 2.0 es el resultado de compilar el código de un vertex shader y el código de un fragment shader juntos. En DroidEngine2D los shader programs se representan mediante la clase `ShaderProgram`.

Desde que se definen los shaders hasta que se pueden utilizar hay que pasar por una serie de etapas. En la clase ShaderProgram de DroidEngine2D estas etapas se resumen a dos métodos, el método `setShaders` y el método `compileAndLink`. Una vez llamados estos 2 métodos se puede utilizar el shader program con el método `use`.

En la siguiente imagen se puede ver el proceso de inicialización y compilación de un shader program tal y como se ha implementado en la clase `ShaderProgram`.

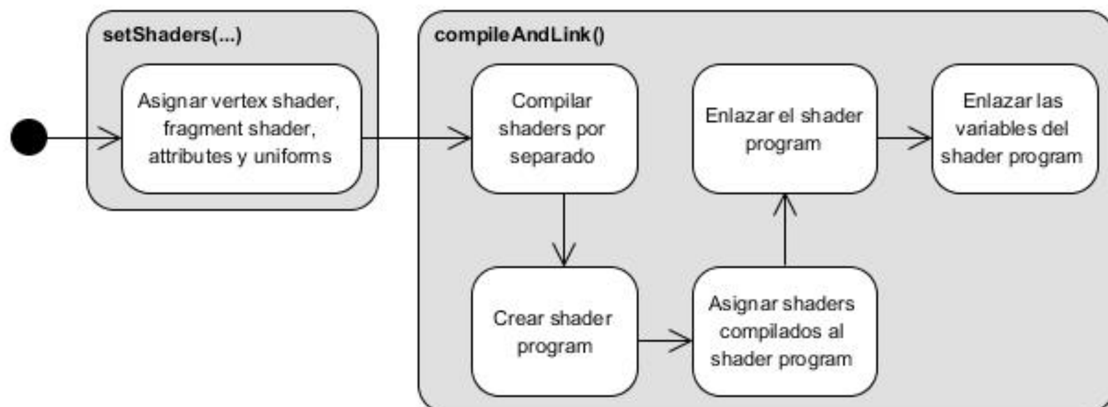

Figura 26. Diagrama que ilustra la inicialización de la clase ShaderProgram.

A continuación se explica cada una de las acciones que se muestran en el diagrama anterior, en orden de ejecución.

1. Se asigna el código del vertex shader y el fragment shader y se especifican los nombres de las variables tipo `attribute` y las variables tipo `uniform` definidas en los shaders.
2. Se compilan ambos shaders por separado utilizando las funciones `glShaderSource` y `glCompileShader`. Tras compilar el shader, OpenGL devuelve un número entero que identifica al shader compilado.
3. Se crea un shader program con la función `glCreateProgram`. Esta función devuelve un número entero que identifica al shader program.

4. Se asignan los shaders compilados al shader program utilizando la función `glAttachShader`, a la que se le pasan como argumentos el identificador del shader program y el identificador del shader compilado.
5. Se enlaza el shader program al contexto de OpenGL.
6. Se obtienen los identificadores de las variables definidas en el shader program en GLSL (números enteros) y se indexan por el nombre de la variable para que sea más sencillo acceder después.

Tras realizar este proceso se podrá llamar al método `use()` de `ShaderProgram`, que internamente llama a la función `glUseProgram` de OpenGL, para utilizar el shader program cada vez que sea necesario.

Una vez seleccionado el shader program para su uso, se podrán enviar datos al shader program, que se ejecuta en la GPU, desde el programa Java, que se ejecuta en la CPU, utilizando los métodos `setAttribute`, `setUniform{1|2|3|4}{f|v}` y `setUniformMatrix{2|3|4}v`.

Renderers de materiales

Hasta ahora se ha explicado qué son los materiales, en qué consiste la optimización mediante el uso de atlas de texturas y batch rendering y qué son y cómo se utilizan los shader programs. Sin embargo, para poder representar un conjunto de rectángulos en una sola llamada, aparte de todo lo anterior, es necesario definir la geometría de los rectángulos a representar y enviarla a los shaders junto con los datos del material a utilizar. Esta geometría consiste en una malla de rectángulos separados entre sí, definidos por sus vértices.

En OpenGL, como ya se ha explicado en el capítulo 3, solo se pueden representar puntos, líneas y triángulos. Por tanto, para representar un rectángulo la mejor opción es utilizar dos triángulos.

En la siguiente figura se puede ver cómo se definiría un rectángulo utilizando dos triángulos: el primer triángulo estaría definido por los vértices V1, V2 y V3; y el segundo triángulo estaría definido por los vértices V3, V4 y V1.

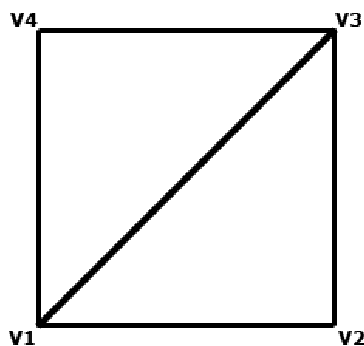

Figura 27. Rectángulo definido mediante 2 triángulos.

La clase `Geometry` se utiliza para definir esta geometría. Y, en concreto, la clase `RectangleBatchGeometry` es la que se utiliza para definir la geometría de los

conjuntos de 32 rectángulos que se pueden llegar a representar en una sola llamada a `glDrawElements`.

Una vez definida la geometría de la malla de rectángulos, es posible representar una malla de 32 rectángulos utilizando dicha geometría, un shader program y el material correspondiente. Sin embargo, habría que gestionar manualmente el envío de datos a la GPU a través del objeto `ShaderProgram` y habría que gestionar un buffer de 32 matrices de transformación homogénea, así como los datos de los materiales de los 32 rectángulos.

Esto es un proceso bastante laborioso y requiere ciertos conocimientos sobre el funcionamiento de OpenGL y el envío de datos a los shaders. Por ello, en DroidEngine2D se han definido unos mecanismos que permiten al usuario utilizar materiales sin tener que gestionar manualmente el envío de datos a la GPU con la API de OpenGL ES. A estos mecanismos se hace referencia como renderers de materiales.

Los renderers de materiales no deben confundirse con el renderer que se ha mencionado cuando se ha mencionado la sincronización entre el hilo del juego y el hilo del renderer. Ese renderer hace referencia al sistema completo de rendering del motor, mientras que los renderers de materiales son simplemente componentes que gestionan la geometría de un conjunto de rectángulos y permiten representar dicho conjunto de rectángulos con un material concreto, haciendo uso del shader program correspondiente a dicho material. Los renderers de materiales permiten que el usuario pueda representar elementos haciendo uso de materiales y batch rendering sin necesidad de conocer el funcionamiento interno de OpenGL ni de la implementación del batch rendering.

En el siguiente diagrama se muestra cómo puede el usuario hacer uso de un renderer de materiales. Como se puede ver, siempre es necesario llamar al método `begin()` al principio para seleccionar el shader program que se va a utilizar y realizar algunas inicializaciones. Tras llamar al método `begin()` se podría llamar a `end()`, que envía a la GPU todos los datos que haya en el buffer pendientes de enviar, aunque no tiene sentido, puesto que no se habrían almacenado datos en el buffer aún. El modo normal de uso es llamar al método `draw` tras llamar a `begin()`. Este método tiene como parámetros la posición, escala, origen y rotación del rectángulo a representar, y lo que hace es generar la matriz de transformación homogénea correspondiente a esos valores y almacenarla en el buffer. Si el buffer está lleno, envía el buffer completo a la GPU (32 rectángulos), de lo contrario tan solo se almacena sin enviar. Finalmente, cuando se han realizado todas las llamadas al método `draw` necesarias (una por cada elemento a representar), es necesario llamar al método `end()` para enviar a la GPU los elementos restantes en el buffer. De lo contrario dichos elementos podrían quedarse sin representar.

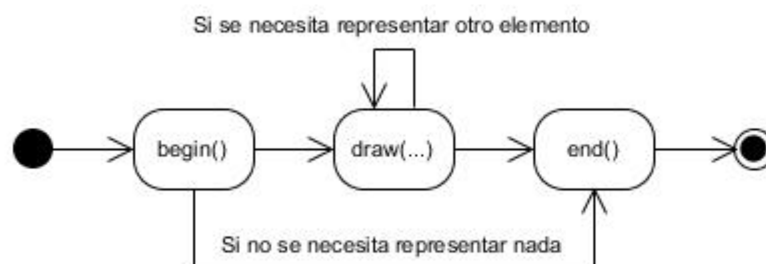

Figura 28. Funcionamiento básico de un renderer de materiales.

En la figura anterior se representa la forma en la que debe utilizarse la clase `GraphicsBatchRenderer`, que es la clase base de la que heredan todos los renderers de materiales de DroidEngine2D.

Por otro lado, en DroidEngine2D se ha definido la clase `RectangleBatchRenderer`, que es una subclase de `GraphicsBatchRenderer` que incorpora la gestión de la geometría de una malla de 32 rectángulos. Todos los renderers de materiales de los materiales por defecto de DroidEngine2D son subclases de `RectangleBatchRenderer`.

Representación de rectángulos con la clase Graphics

Como se ha visto anteriormente, los renderers de materiales permiten representar rectángulos utilizando batch rendering. Sin embargo, un renderer de materiales solo sirve para representar un tipo de material concreto, y el usuario debe saber que debe llamar a los métodos `begin`, `draw` y `end` tal y como se describe en el diagrama de la *Figura 28*. Además el usuario deberá tener en cuenta que debe llamar al método `end` de un renderer de materiales antes de llamar al método `begin` de otro renderer para representar elementos con un material distinto, de lo contrario podría dejar elementos sin representar. Esto es un proceso que, aunque no es demasiado laborioso, puede dar lugar a error y, ciertamente, hace que el usuario tenga que gestionar detalles de más bajo nivel del que se pretendía originalmente.

Para evitar lo anterior, DroidEngine2D cuenta con la clase `Graphics`, que contiene un método llamado `drawRect` que permite representar rectángulos utilizando los renderers de materiales predefinidos. El usuario no tiene que saber cómo funcionan los renderers de materiales. En lugar de ello, tan solo tendrá que llamar al método `drawRect` especificando un objeto `Transform` (define la posición, escala, rotación y origen de un rectángulo) y un material.

La clase `Graphics`, por defecto solo permite representar rectángulos con los materiales incluidos por defecto en DroidEngine2D, aunque es posible crear una subclase de `Graphics` que agregue renderers de materiales personalizados para soportar materiales implementados por el usuario. Para ello, en dicha subclase tan solo habría que sobrescribir el método `loadMaterialRenderers()`, y cargar los renderers de materiales del usuario en el mapa de renderers tal y como se describe en la documentación de la API [62]. Por último habría que crear una implementación de `EngineRenderer` que utilizase la subclase de `Graphics` implementada, en lugar de utilizar el objeto `Graphics` que utiliza `DefaultRenderer`, y especificar que se use dicho `EngineRenderer` al inicializar el motor.

4.3.4.2.3. Pruebas

Las pruebas se han realizado en forma de demos que muestran el funcionamiento de las características del motor descritas anteriormente. Estas demos se exponen a continuación.

Demos de representación básica de rectángulos

Las demos de representación básica de rectángulos que se han desarrollado son las siguientes:

- Demo de representación de rectángulos sin rotación.
- Demo de representación de rectángulos rotados con respecto a su centro.
- Demo de representación de rectángulos rotados con respecto a un punto cualquiera del rectángulo.

Demos de uso de materiales

Las demos de uso de materiales que se han desarrollado son las siguientes:

- Demo de uso de las opciones de fusión (`BlendingOptions`).
- Demo de uso de `ColorMaterial`.
- Demo de uso de `TextureMaterial`.
- Demo de uso de `TextureColorMaterial`.
- Demo de uso de `TransparentTextureMaterial`.
- Demo de uso de `TextureHsvMaterial`.

En todas estas demos se permite al usuario modificar los parámetros de color y transparencia, de forma que pueda ver el comportamiento de cada tipo de material.

Demos de batch rendering

Estas demos permiten mostrar al usuario la reducción en el número de llamadas a la función `glDrawElements` de OpenGL que se obtiene al utilizar batch rendering.

Se han realizado cuatro demos, que muestran el comportamiento del renderer en cuatro escenarios distintos. Estas demos se explican a continuación.

- **Un solo material:** En esta demo se representan 512 rectángulos utilizando un solo tipo de material, `TextureMaterial`. Se puede observar que los 512 rectángulos se representan en 16 llamadas a `glDrawElements` (512 / 32 = 16).
- **Varios materiales (caso 1):** En esta demo se representan 256 rectángulos con `ColorMaterial`, seguidos de 256 rectángulos con `TextureMaterial`. Se puede observar que, al igual que en la demo anterior, se realizan 16 llamadas a `glDrawElements`. Los primeros 256 se representan en 8 llamadas (256 / 32 = 8) y los siguientes 256 en otras 8 llamadas.
- **Varios materiales (caso 2):** En esta demo se representan 257 rectángulos con `ColorMaterial`, seguidos de 255 rectángulos con `TextureMaterial`. Se puede observar que se realizan 17 llamadas a `glDrawElements` en lugar de 16 como en el caso anterior. Esto se debe a que 257 entre 32 es 8,03, lo que implica que para representar 257 rectángulos sean necesarios 9 paquetes (batches) y, por tanto, 9 llamadas a `glDrawElements`. Por otro lado, los 255 rectángulos que re representan con `TextureMaterial` siguen pudiendo ser representados con 8

llamadas, aunque en la última llamada se representan 31 elementos en lugar de un paquete completo de 32.

- **Varios materiales (caso 3):** En esta demo se representan 256 rectángulos con `ColorMaterial` y 256 rectángulos con `TextureMaterial`, de forma que nunca se representen 2 rectángulos seguidos con el mismo material. Se puede observar que se realizan 512 llamadas a `glDrawElements`, puesto que no es posible enviar a la GPU, a la vez, elementos que hacen uso de distintos shaders. Este es el peor caso posible desde el punto de vista de la optimización del rendering, puesto que no se aprovecha el batch rendering. Para evitarlo, lo mejor es representar consecutivamente todos los rectángulos que hagan uso del mismo material, siempre que sea posible.

4.3.4.3. Representación de texto

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la representación de texto.

4.3.4.3.1. Análisis

La representación de texto es una característica bastante útil en un motor de juegos, puesto que en muchas ocasiones es conveniente mostrar mensajes al usuario, descripciones de elementos o acciones del juego, etc.

En el caso de DroidEngine2D, el objetivo era poder representar texto en pantalla utilizando OpenGL ES. Además, el usuario debe poder representar el texto utilizando una fuente de su elección especificada en el formato definido por el programa BMFont [78], o bien definiendo otro cargador de fuentes que le permita cargar una fuente definida en un formato distinto.

4.3.4.3.2. Diseño e implementación

OpenGL es una librería gráfica pensada para representar escenas 3D y no cuenta con soporte para representar texto directamente en pantalla, por lo que la única opción restante es representar las fuentes utilizando imágenes y representar cada letra como una imagen.

Conversión de fuentes en formato TrueType a formato bitmap

Para poder representar texto utilizando una fuente concreta, en primer lugar hay que convertirla del formato TrueType (estándar) al formato bitmap.

Una fuente en formato bitmap se define por un conjunto de atlas de texturas que contienen las representaciones de todos los símbolos de dicha fuente y un archivo XML que define las coordenadas en el atlas de texturas en las que se encuentra cada uno de los símbolos de la fuente, así como las dimensiones de cada símbolo, el espaciado, y otros atributos necesarios para representar texto con dicha fuente.

DroidEngine2D proporciona la clase `BitmapFont`, que implementa la interfaz `Font` y permite cargar y utilizar una fuente en formato bitmap generada con la aplicación externa BMFont [78], que es un generador de fuentes en formato bitmap a partir de

fuentes en formato TrueType. Esta aplicación genera tanto los atlas de texturas como el archivo XML que contiene las descripciones de cada uno de los caracteres de la fuente.

Una vez generada la fuente en formato bitmap con BMFont, tan solo hay que colocar los archivos generados en la carpeta *assets* del proyecto Android correspondiente al videojuego que se está desarrollando. De esta forma DroidEngine2D puede utilizar la clase `BitmapFont` para leer dichos archivos.

En el apéndice A se describe detalladamente cómo utilizar BMFont para generar una fuente compatible con la clase `BitmapFont` de DroidEngine2D.

Representación de texto utilizando las clases Graphics y BitmapFont

Tras convertir la fuente de formato TrueType a formato bitmap y cargar dicha fuente con la clase `BitmapFont` de DroidEngine2D, se puede utilizar la clase `Graphics` para representar en pantalla un texto con dicha fuente. Para ello, la clase `Graphics` incluye un método llamado `drawText`, que internamente gestiona un renderer de materiales que permite representar rectángulos con `TextureColorMaterial`.

El método `drawText` lo que hace es obtener la imagen correspondiente a cada carácter del texto a mostrar con la fuente especificada y representa cada carácter como una imagen independiente haciendo uso de un renderer de materiales, por lo que los caracteres se envían a la GPU en paquetes de 32 elementos como máximo (tal y como se ha explicado anteriormente al describir el funcionamiento del batch rendering en DroidEngine2D).

Al representar texto con el método `drawText` se puede especificar la fuente a utilizar, la posición en la que aparecerá el texto, el tamaño y color del texto y, si se desea, la rotación y el punto alrededor del cual se rotará el texto.

Uso de otros formatos distintos del que utiliza BMFont

Como ya se ha comentado varias veces anteriormente, DroidEngine2D pretende ser un motor extensible, que se pueda adaptar a las necesidades concretas de los usuarios con relativa facilidad.

Por defecto, solo soporta fuentes en el formato de salida de BMFont (este formato se explica en detalle en el apéndice A), sin embargo, también es posible utilizar otros formatos de representación de fuentes en caso de que el usuario lo necesite. Para ello, habría que crear una clase que implemente la interfaz `Font` y que sea capaz de leer el formato deseado. Una vez hecho esto, habría que extender la clase `Graphics` agregando un método que permita representar texto con dicha fuente.

4.3.4.3.3. Pruebas

Las pruebas se han realizado en forma de demos que muestran el funcionamiento de las características del motor descritas anteriormente. En este caso concreto, solo se ha desarrollado una demo, que consiste en representar varios mensajes con varias fuentes distintas. También se ha comprobado que se pueda representar texto rotado alrededor de un punto especificado.

4.3.4.4. Gestión de texturas

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la gestión de texturas.

4.3.4.4.1. Análisis

Como ya se ha mencionado anteriormente, cuando una activity que contiene un `GLSurfaceView` pasa a segundo plano, el contexto de OpenGL se destruye; y cuando la activity vuelve a primer plano, se crea un contexto de OpenGL nuevo.

Al destruirse el contexto de OpenGL, todos los recursos vinculados a dicho contexto se liberan automáticamente, por lo que, en Android, cuando la activity vuelve a primer plano es necesario volver a cargar todas las texturas. Esto es una tarea que no debería tener que realizar el usuario, por lo que se tomó la decisión de que DroidEngine2D contase con un mecanismo que permita que la recarga de texturas se realice de forma automática.

El gestor de texturas de DroidEngine2D permite almacenar todas las texturas y atlas de texturas de forma que cualquier objeto con acceso al `Game` pueda acceder a las texturas del juego.

Además, el renderer tiene acceso al gestor de texturas, lo que permite que éstas sean recargadas cuando se reinicializa el contexto de OpenGL.

DroidEngine2D también permite utilizar atlas de texturas. Para ello hay dos opciones: utilizar el formato soportado por defecto (atlas de texturas definidos con TexturePacker [79] en formato XML), o crear un cargador de atlas de texturas que permita cargar un formato distinto.

4.3.4.4.2. Diseño e implementación

A continuación se describen los detalles más relevantes del diseño y la implementación de la gestión de texturas.

Gestor de texturas

El gestor de texturas de DroidEngine2D está definido por la clase `TextureManager`, que simplemente almacena todas las texturas que se utilizan en el juego.

La interfaz del juego, `Game`, permite acceder al `TextureManager` del juego, por lo que todas las texturas son accesibles desde cualquier objeto que tenga acceso al objeto que implementa la interfaz `Game`. De esta forma, el renderer, al tener acceso al `Game`, puede acceder al `TextureManager` y recargar las texturas cuando sea necesario.

La idea detrás del gestor de texturas es que todas las texturas del juego deben ser registradas en él. Para ello proporciona algunos métodos como `addTexture`, `addTextureAtlas`, `addTextureRegion` y `addFontTextures`.

También es posible eliminar texturas del gestor de texturas en caso de que ya no sean necesarias o porque haga falta liberar memoria para cargar otros recursos del juego.

Atlas de texturas

Los atlas de texturas ya se han explicado anteriormente, no obstante, en este apartado se explica más detalladamente cómo se han implementado en DroidEngine2D.

A la hora de explicar esto, son de especial interés `Texture`, `TextureRegion`, `TextureAtlas` y `TexturePackerAtlas`,.

La clase `Texture` define una textura y encapsula las llamadas a las funciones de OpenGL necesarias para enlazar dicha textura al contexto de OpenGL. En la mayoría de los casos, el usuario no necesitará tratar con objetos `Texture` directamente.

La clase `TextureRegion` define simplemente una región dentro de una textura. Almacena una referencia a dicha textura, las coordenadas en las que se encuentra la región dentro de la textura y las dimensiones de la región. Además es posible voltear la región horizontalmente y verticalmente. Esta característica puede ser de utilidad cuando se desea representar una imagen, por ejemplo una cara de perfil, mirando hacia un lado y hacia el otro. Si no se pudiese voltear la región, habría que definir dos regiones con la misma imagen, una volteada y otra sin voltear, en el mismo atlas de texturas, lo cual supondría que la aplicación ocupase más espacio en disco al tener que duplicar imágenes.

La interfaz `TextureAtlas` representa un atlas de texturas y, en caso de que el usuario necesite utilizar un atlas de texturas definido en un formato distinto del formato que soporta DroidEngine2D por defecto, tan solo tendría que crear una clase que implemente esta interfaz e implementar la carga del atlas de texturas manualmente en el método `loadFromFile`.

Por último, la clase `TexturePackerAtlas` es la implementación de la interfaz `TextureAtlas` proporcionada por DroidEngine2D por defecto. Esta clase permite cargar un atlas de texturas previamente creado con el programa TexturePacker [79] y representado en formato XML.

En el apéndice B se explica detalladamente qué es TexturePacker, cómo obtenerlo y cómo utilizarlo para definir un atlas de texturas compatible con la clase `TexturePackerAtlas` de DroidEngine2D.

Por último, mencionar que, al igual que las fuentes generadas con BMFont, los atlas de texturas generados con TexturePacker también deben incluirse en la carpeta *assets* del proyecto Android.

4.3.4.4.3. Pruebas

Las pruebas se han realizado en forma de demos que muestran el funcionamiento de las características del motor descritas anteriormente.

Prácticamente todas las demos desarrolladas hacen uso de texturas, por lo que solo se ha implementado una demo adicional, que permite verificar el funcionamiento del volteo horizontal y vertical de objetos `TextureRegion`.

4.3.5. Entrada de usuario

En este bloque se explican las principales características relacionadas con la gestión de la entrada de usuario en DroidEngine2D.

4.3.5.1. Gestión de eventos de entrada táctil

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la gestión de eventos de entrada táctil.

4.3.5.1.1. Análisis

Una de las características más necesarias en un motor de juegos para Android es la capacidad de recibir eventos de entrada táctil y permitir reaccionar ante ellos.

Los eventos de entrada de usuario en Android se lanzan en el hilo principal de la aplicación y, por tanto, los callbacks que registre el usuario para poder reaccionar a dichos eventos, también se ejecutan en el hilo principal. Esto, en el caso de DroidEngine2D, suponía un problema originalmente, puesto que para poder actualizar el estado del juego en función de los eventos de entrada de usuario era necesario tener en cuenta que el juego y los eventos se procesaban en hilos distintos.

Como los eventos se producen de forma asíncrona, para el usuario del motor era necesario sincronizar la actualización del juego con el procesamiento de eventos, lo cual es una tarea compleja que debería realizar el propio motor. Por este motivo, se decidió que DroidEngine2D debía poder encolar los eventos de entrada táctil cuando se reciben para poder procesarlos posteriormente en el hilo del juego de forma secuencial. De esta forma, la entrada de usuario siempre se procesa en orden conocido, justo antes de comenzar a actualizar cada frame, evitando posibles problemas de sincronización.

Además, los procesadores de eventos deben ser específicos de cada estado del juego. Es común que en cada estado del juego se desee procesar los eventos de entrada de usuario de forma distinta.

4.3.5.1.2. Diseño e implementación

Para implementar lo descrito en el apartado anterior, DroidEngine2D cuenta con varas clases.

En primer lugar, la clase `GameInputManager`, gestiona toda la entrada de usuario a nivel de juego. Esta clase contiene un `GameStateInputManager`, que es el que gestiona la entrada de usuario a nivel de estado del juego.

El `GameInputManager` deriva toda la entrada de usuario recibida (táctil y mediante teclas físicas del dispositivo) al `GameStateInputManager` actual, que será siempre el correspondiente al estado del juego que se encuentre en la cima de la pila de estados activos.

Todos los eventos recibidos en el hilo principal de la aplicación se clonan y se almacenan en una cola de eventos pendientes de ser procesados. Posteriormente, en el hilo del juego se procesan todos los eventos previamente almacenados en dicha cola, quedando ésta vacía.

Recepción de eventos en el hilo principal

En el caso de la entrada táctil, todos los eventos recibidos se clonan y se almacenan en una cola de eventos pendientes de ser procesados.

El motivo por el que hay que almacenar una copia del objeto `MotionEvent` recibido es que el sistema recicla automáticamente los objetos `MotionEvent` tras salir del método `onTouch`, por lo que si se encola y ocurre otro evento, es posible que el sistema modifique la instancia de `MotionEvent` que se había encolado. Almacenando una copia del objeto se evita este posible problema.

Para clonar los eventos se ha utilizado el método `obtain(MotionEvent)` de la clase `MotionEvent`. Este método devuelve un objeto `MotionEvent` que es una copia del objeto que se le pasa por parámetro. Es importante que al utilizar el método `obtain`, el objeto obtenido se recicle más adelante cuando deje de ser necesario. Esto se explica de nuevo más adelante, cuando se explique el procesamiento de eventos.

En el siguiente diagrama se puede ver cómo gestiona `GameInputManager` un evento de entrada de usuario táctil cuando lo recibe.

Es conveniente aclarar que en caso de que el GameStateInputManager actual sea nulo, el método `onTouch` no hace nada. Sin embargo, esto no se ha incluido en el diagrama para facilitar la comprensión del mismo, ya que lo que se pretende ilustrar es cómo se encolan en el `TouchProcessor` del `GameStateInputManager` actual los eventos recibidos.

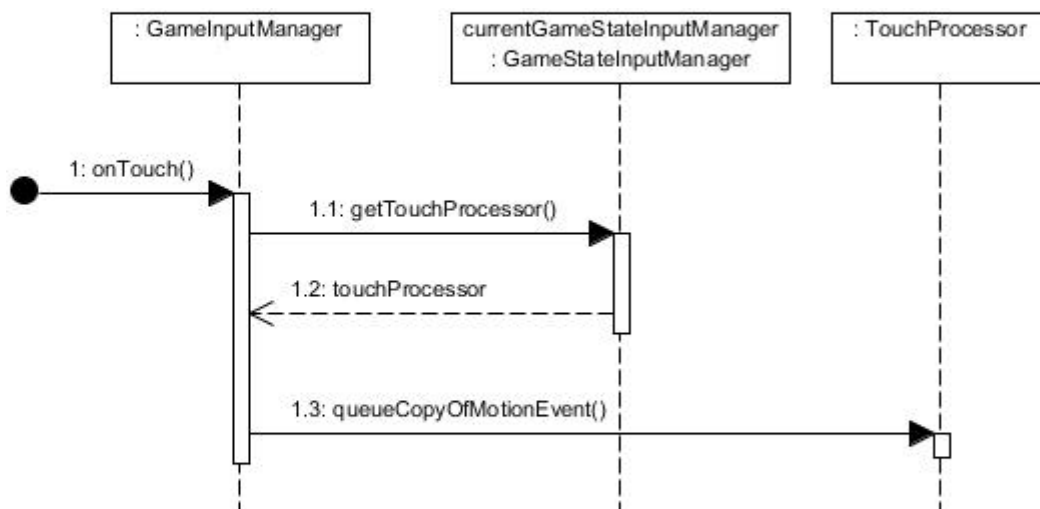

Figura 29. Diagrama de secuencia de la recepción de eventos de entrada táctil en GameInputManager.

Y en el siguiente diagrama se puede ver más detalladamente el funcionamiento interno del método `queueCopyOfMotionEvent` de la clase `TouchProcessor`.

Los parámetros de los métodos no se muestran por simplificar el diagrama. Sin embargo, es conveniente aclarar que al método `obtain` se le pasa por parámetro el evento recibido y devuelve una copia exacta de dicho evento; y al método `add` de la cola se le pasa por parámetro la copia del evento devuelta por el método `obtain`. También cabe destacar que el acceso a la cola de eventos se realiza en exclusión mutua, puesto que dicha cola se modifica desde dos hilos distintos. El encolado de eventos se

realiza en el hilo principal, mientras que el procesamiento de dichos eventos se realiza en el hilo del juego.

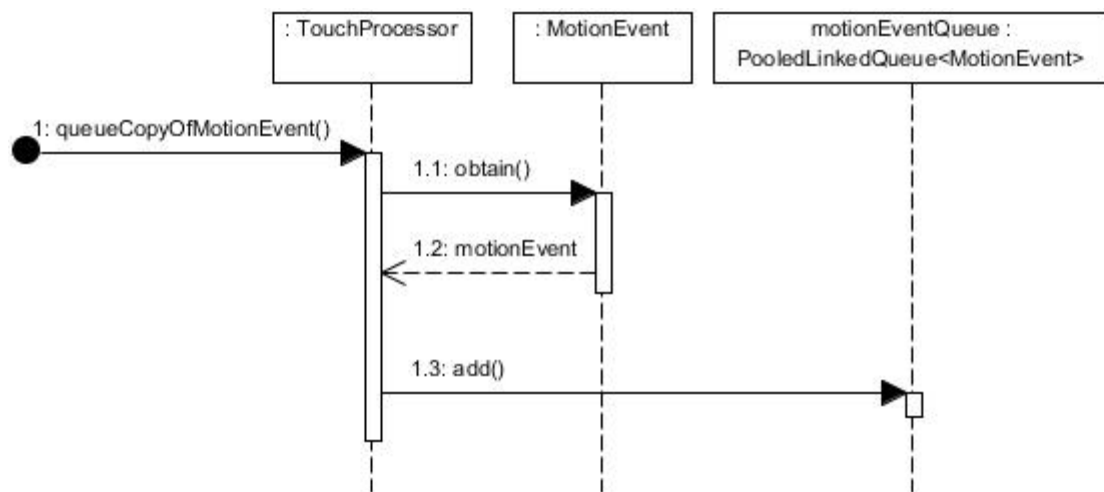

Figura 30. Diagrama de secuencia del método queueCopyOfMotionEvent de TouchProcessor.

Procesamiento de eventos en el hilo del juego

Una vez encolados los eventos que se reciben en el hilo principal, es necesario procesarlos en el hilo del juego. Para ello, en cada frame, antes de actualizar el `GameState` que se encuentra en la cima de la pila de estados activos, se procesan todos los eventos encolados, vaciando la cola de eventos pendientes.

En el siguiente diagrama se puede observar cómo se procesan los eventos antes de actualizar cada frame. Esto incluye eventos de entrada táctil y eventos de entrada mediante teclas físicas del dispositivo.

Si el `GameState` activo es nulo, las llamadas desde la 1.2 en adelante no se ejecutan. No se ha incluido esta comprobación no en el diagrama para facilitar la comprensión de lo que se pretende ilustrar, que es cómo se procesa la entrada de usuario en el `GameState` que se encuentra en la cima de la pila de estados activos.

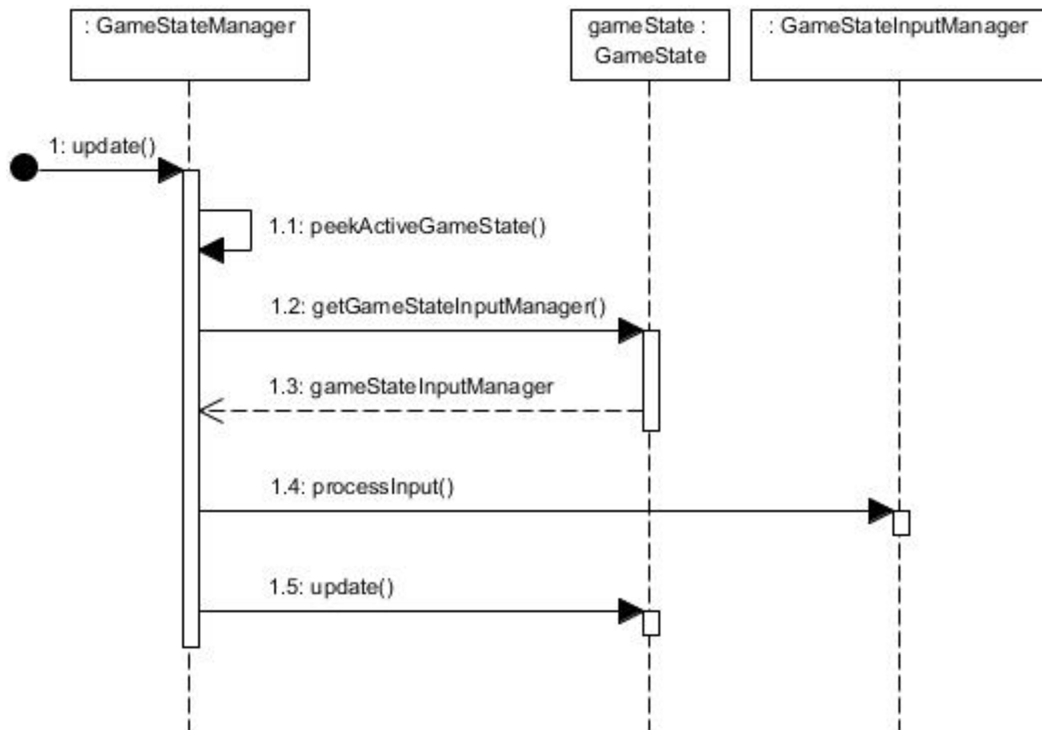

Figura 31. Diagrama de secuencia del método update de GameStateManager.

En el siguiente diagrama se puede ver el funcionamiento del método `processInput()` de `GameStateInputManager`.

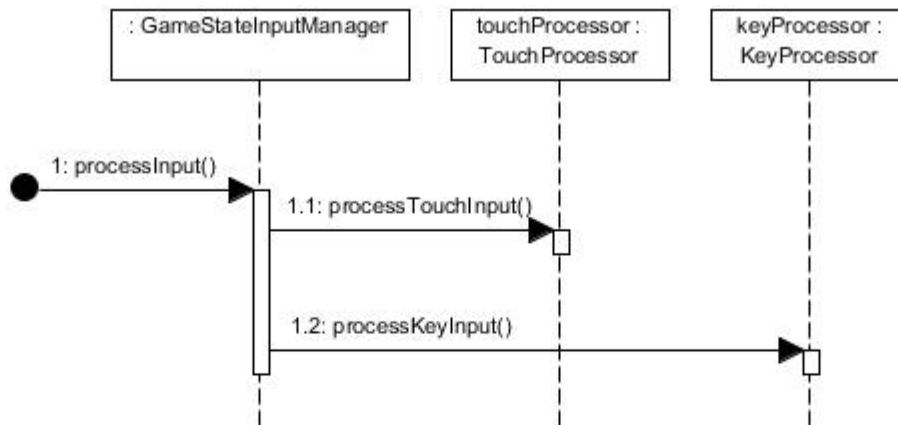

Figura 32. Diagrama de secuencia del método processInput de GameStateInputManager.

A continuación se expone el funcionamiento del método `processTouchInput()` de `TouchProcessor`. En este diagrama se asume que el usuario ha registrado un `MotionEventProcessor` en el `TouchProcessor` para poder procesar los eventos encolados.

Como se puede observar, tal y como se ha comentado anteriormente, tras procesar el evento, se llama al método `recycle()` para que el sistema pueda utilizar el mismo objeto más adelante y no tenga que instanciar objetos nuevos si es posible.

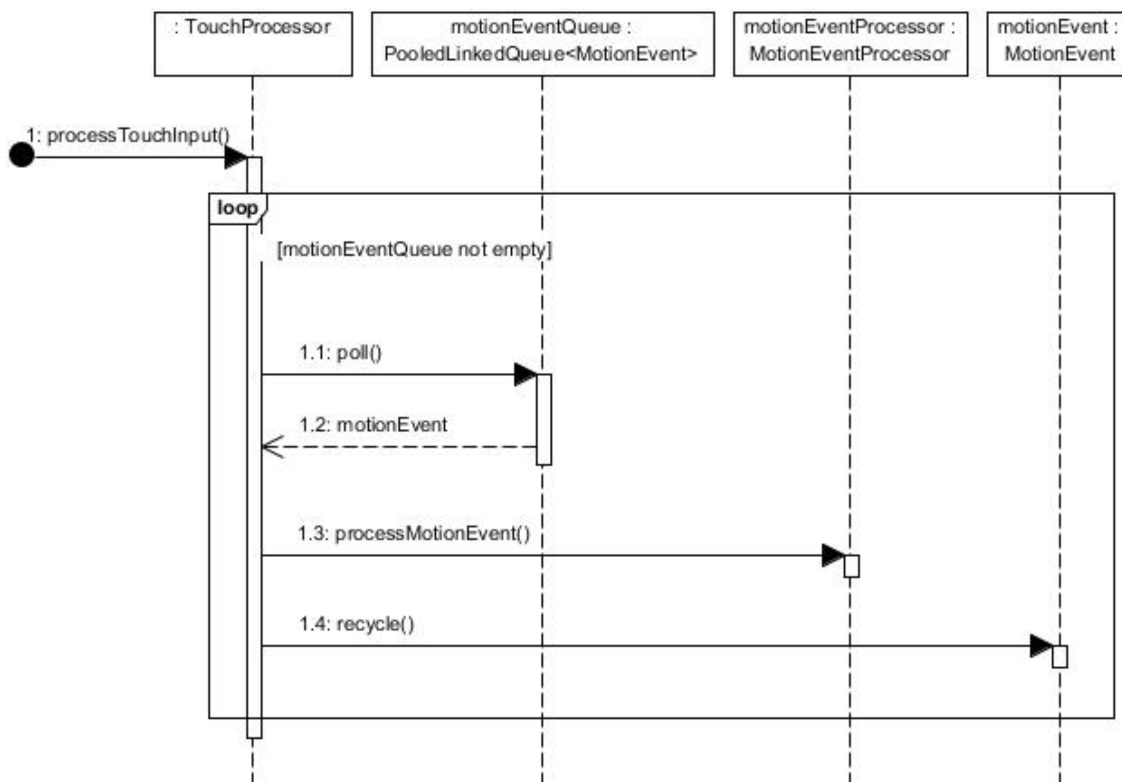

Figura 33. Diagrama de secuencia del método processTouchInput de TouchProcessor.

Por último, mencionar que para procesar eventos de entrada táctil, el usuario tan solo debe definir un objeto que implemente `MotionEventProcessor` y registrarlo en el `TouchProcessor` del `GameState` cuyos eventos desee procesar. Y en el método `processMotionEvent` del `MotionEventProcessor` deberá implementar el procesamiento del evento.

Pruebas

Las pruebas se han realizado en forma de demos que muestran el funcionamiento de las características del motor descritas anteriormente. En este caso, únicamente se ha implementado una demo, que permite desplazar un círculo por la pantalla del dispositivo utilizando la entrada táctil. En esta demo, al tocar sobre cualquier punto de la pantalla, el círculo se colocará en dicho punto. También es posible arrastrar el círculo desplazando el dedo por la pantalla del dispositivo sin levantarlo. Y el círculo cambia de color mientras está pulsado, por lo que con esta demo es posible comprobar bastante bien el funcionamiento de la entrada táctil.

Además, aunque no se han desarrollado específicamente para verificar el funcionamiento de la entrada táctil, las demos del funcionamiento de los estados del juego permiten comprobar que, efectivamente, cada estado del juego gestiona su propio procesador de eventos de entrada táctil.

4.3.5.2. Gestión de eventos de entrada mediante teclas físicas

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la gestión de eventos de entrada mediante teclas físicas del dispositivo.

4.3.5.2.1. Análisis

En este caso, lo que se pretendía era poder gestionar los eventos de entrada mediante teclas físicas del dispositivo de la misma forma que se gestionan los eventos de entrada táctil (ver apartado 4.3.5.1.1).

La única diferencia en cuanto a funcionalidad es que, en este caso, además se debe proporcionar un procesador de eventos de teclas físicas por defecto, de forma que el usuario no tenga que implementar uno si no desea procesar las teclas físicas.

4.3.5.2.2. Diseño e implementación

El diseño y la implementación de esta característica son muy similares a los de la gestión de eventos de entrada táctil. De hecho, todos los diagramas y las explicaciones del apartado 4.3.5.1.2 sirven para entender la forma en la que está diseñada la gestión de eventos de teclas físicas, aunque hay unas pequeñas diferencias de implementación por limitaciones de la API de Android.

La principal limitación es que la clase `KeyEvent`, a diferencia de la clase `MotionEvent`, no permite acceder al método `recycle()`, por lo que ha sido necesario crear la clase `KeyEventInfo` y encolar objetos de este tipo en lugar de `KeyEvent`. La clase `KeyEventInfo` contiene la información más relevante de la clase `KeyEvent`, y posee un constructor que recibe por parámetro un `KeyEvent` y crea un `KeyEventInfo` que contiene prácticamente la misma información que el `KeyEvent` proporcionado.

Además, si se elimina el `KeyEventProcessor` del `KeyProcessor` de un `GameState`, la tecla “Back” se procesa de la misma forma que lo hace Android por defecto, terminando la activity actual.

Finalmente, se ha incluido la clase `DefaultKeyEventProcessor`, que es la implementación por defecto de `KeyEventProcessor`. Esta clase procesa las teclas “Back”, “Volume up” y “Volume down” de forma que tengan el comportamiento que tienen en Android por defecto. El `KeyProcessor`, por defecto, gestiona una instancia de esta clase, de forma que las teclas básicas se procesan tal y como espera el usuario.

4.3.5.2.3. Pruebas

Las pruebas se han realizado en forma de demos que muestran el funcionamiento de las características del motor descritas anteriormente. En este caso, únicamente se ha implementado una demo específica, que permite comprobar que al tocar cualquier tecla,

el evento se procesa. En este caso, esto se puede verificar porque se cambia el color de la pantalla al pulsar una tecla.

En cuanto a las pruebas de que `DefaultKeyEventProcessor` funciona tal y como se espera, todas las demás demos utilizan dicho `KeyEventProcessor`, por lo que no ha sido necesario agregar ninguna demo adicional.

4.3.5.3. Captura de datos del acelerómetro

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la captura de datos del acelerómetro.

4.3.5.3.1. Análisis

El uso del acelerómetro puede ser una opción interesante en algunos videojuegos, así que se decidió que DroidEngine2D incorporase un mecanismo para facilitar dicha tarea.

Este mecanismo permite al usuario decidir cuándo se comienzan a capturar valores del acelerómetro, además de poder parar de capturar valores en el momento que desee.

DroidEngine2D proporciona un listener por defecto que se puede utilizar para capturar las coordenadas del vector de aceleración. No obstante, también permite al usuario utilizar un listener personalizado, en caso de que lo necesitase.

4.3.5.3.2. Diseño e implementación

La clase que gestiona el uso del acelerómetro en DroidEngine2D es la clase `Accelerometer`. Esta clase, permite comenzar a escuchar eventos del acelerómetro con el método `startListening()`; y también permite dejar de escuchar eventos con el método `stopListening()`.

La clase `Accelerometer` por defecto cuenta con un listener de valores de aceleración, un objeto tipo `AccelerometerValuesListener`. Este objeto, siempre que se estén escuchando eventos del acelerómetro, almacena el vector de aceleración devuelto por el último evento recibido, de forma que en cualquier momento se puede dicho vector.

La clase `AccelerometerValuesListener` devuelve las coordenadas del vector de aceleración en el sistema de coordenadas por defecto del dispositivo o el sistema de coordenadas de la pantalla.

En el sistema de coordenadas por defecto del dispositivo, cuando el dispositivo se encuentra en su orientación natural, el eje X apunta hacia la derecha de la pantalla del dispositivo, el eje Y apunta hacia arriba, y el eje Z es perpendicular a la pantalla y apunta hacia fuera (los valores negativos irían hacia atrás de la pantalla). Para utilizar este sistema hay que llamar al método `useDefaultCoordinateSystemOfDevice()` tras crear el objeto `Accelerometer`.

El problema de este sistema de coordenadas es que si por ejemplo la orientación por defecto del dispositivo es portrait (vertical), cuando la activity está en dicha orientación el sistema de coordenadas del acelerómetro coincide con el sistema de coordenadas de la activity, mientras que si la activity pasa a orientación landscape (apaisada), los sistemas de coordenadas no coinciden.

En la siguiente figura se puede observar de forma gráfica este comportamiento. El sistema de coordenadas que se muestra dentro de la pantalla de los dispositivos es el sistema de coordenadas de la activity, y el que se representa con flechas grises (ax, ay) es el del acelerómetro (el eje Z coincide siempre en ambos sistemas).

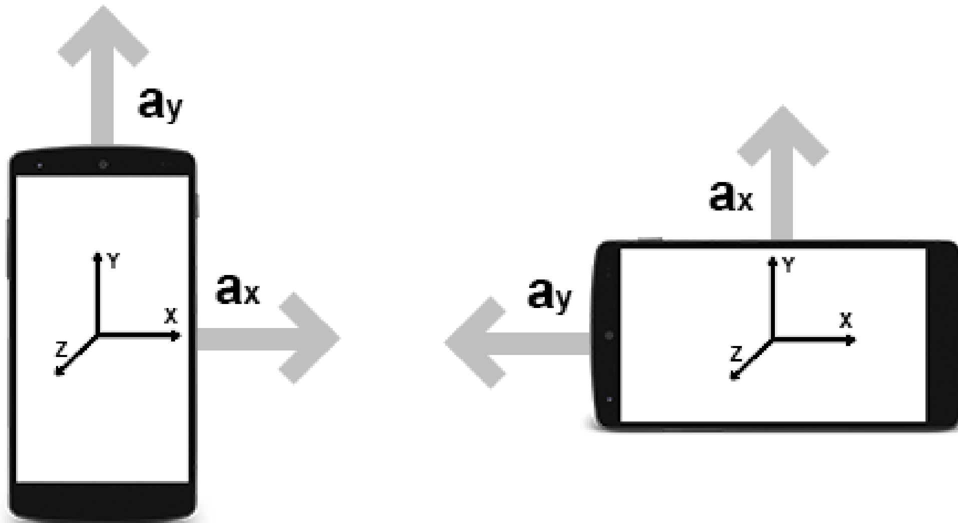

Figura 34. Sistema de coordenadas con useDefaultCoordinateSystemOfDevice() en un dispositivo portrait.

Para evitar este problema, DroidEngine2D introduce una nueva opción en la clase `Accelerometer`, que es utilizar el sistema de coordenadas de la pantalla. Este sistema de coordenadas, en lugar de estar definido con respecto a la orientación natural del dispositivo, está definido con respecto a la orientación de la activity, y siempre coincide con el sistema de coordenadas de la misma. Para utilizar este sistema, tan solo hay que llamar al método `useCoordinateSystemOfDisplay()` tras crear el objeto `Accelerometer`.

En la siguiente imagen se puede ver el mismo ejemplo anterior, pero utilizando este sistema de coordenadas.

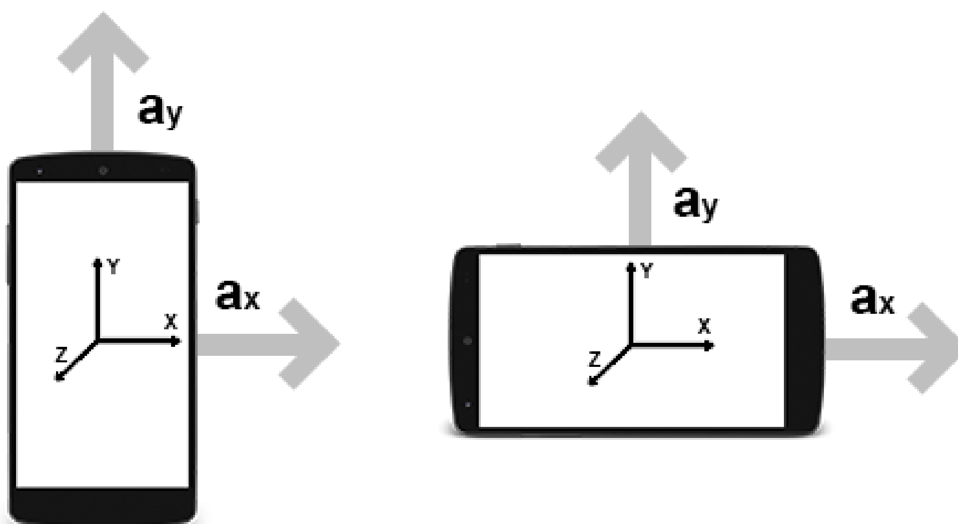

Figura 35. Sistema de coordenadas con useCoordinateSystemOfDisplay() en un dispositivo portrait.

La clase `Accelerometer`, además, permite asignar un listener personalizado (de forma opcional). Esto facilita el trabajo a los usuarios que necesiten realizar otras operaciones con el acelerómetro distintas de obtener el vector de aceleración. Estos listeners deben implementar la interfaz `SensorEventListener` proporcionada por el SDK de Android.

4.3.5.3.3. Pruebas

Las pruebas del funcionamiento del acelerómetro se han realizado mediante una demo que permite controlar un objeto inclinando el dispositivo. Y en una demo de un minijuego que integra la mayoría de características del motor también se ha hecho uso de la clase `Accelerometer`.

4.3.6. Carga de recursos

En este bloque se explican las principales características relacionadas con la carga de recursos en DroidEngine2D.

4.3.6.1. Carga de recursos desde archivos

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de la carga de recursos desde archivos.

4.3.6.1.1. Análisis

En todo motor de juegos es necesario poder cargar recursos desde archivos. Hay multitud de tipos de recursos que puede ser necesario cargar en el juego: imágenes, audio, niveles del juego, etc.

En el caso de DroidEngine2D, se decidió que era indispensable poder cargar imágenes y archivos de audio. Aunque también se proporcionan utilidades para leer archivos de texto, lo cual permite leer archivos XML por ejemplo.

4.3.6.1.2. Diseño e implementación

Para facilitar la carga de recursos desde archivos, DroidEngine2D proporciona la clase `AssetsLoader`. Esta clase proporciona métodos de utilidad para cargar prácticamente cualquier tipo de recurso de la carpeta *assets* del proyecto. Además es posible especificar rutas externas a la carpeta assets, aunque esto normalmente no es necesario.

4.3.6.1.3. Pruebas

Las pruebas de esta característica se han realizado en forma de demos. Sin embargo no hay demos específicas, puesto que la carga de recursos se utiliza prácticamente en todas las demos implementadas.

4.3.7. Utilidades

En este bloque se explican las principales utilidades que proporciona DroidEngine2D. Estas utilidades son, en su mayoría, simplemente componentes o funciones que se han

implementado durante el desarrollo del motor y que se han utilizado para facilitar la implementación de algunas características del mismo. Aunque algunas de ellas se han implementado con intención de que el usuario las utilice en sus juegos si esto le facilita el trabajo.

4.3.7.1. Utilidades para trabajar con tiempos

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de las utilidades para trabajar con tiempos que proporciona DroidEngine2D.

4.3.7.1.1. Análisis

En muchos videojuegos es necesario trabajar con tiempos, y uno de los principales usos de esta característica sería un simple contador de tiempo transcurrido. Por tanto, DroidEngine2D proporciona al usuario un contador de tiempo con el objetivo de facilitarle esta tarea.

4.3.7.1.2. Diseño e implementación

El contador de tiempo se ha implementado en la clase `TimeCounter`, y su funcionamiento es muy sencillo. Tan solo hay que inicializarlo y llamar al método `update()` cuando se desee. También es posible reiniciar el contador con el método `reset()`.

El contador almacena el tiempo transcurrido desde que se inició hasta la última llamada a `update()` realizada.

La clase `TimeCounter`, obviamente, también posee métodos para acceder al tiempo transcurrido. Estos métodos son `getMilliseconds()`, `getSeconds()`, `getMinutes()`, `getHours()` y `getDays()`.

Para todas estas conversiones de tiempo se han utilizado las constantes definidas en la clase `TimeConstants`.

4.3.7.1.3. Pruebas

Las pruebas del contador de tiempo se han llevado a cabo, en su mayoría, haciendo uso de la clase `Log` proporcionada por el SDK de Android, que permite imprimir mensajes de depuración. Y en una de las demos del motor también se hace uso de `TimeCounter` para hacer que una pelota sea inmune a colisiones durante cierto tiempo tras haber colisionado con un objeto.

4.3.7.2. Utilidades para trabajar con vectores y matrices

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de las utilidades para trabajar con vectores y matrices que proporciona DroidEngine2D.

4.3.7.2.1. Análisis

En la parte de gráficos del motor se hace bastante uso de vectores de dos y tres coordenadas, por lo que era necesario crear utilidades que permiten trabajar con dichos vectores.

Y, en cuanto a las matrices, tan solo es necesario trabajar con matrices de 4x4 elementos para representar las matrices con las que trabaja OpenGL, por lo que el desarrollo se ha limitado exclusivamente a este aspecto.

Por último, era necesario implementar una alternativa a la función `rotateM` de la clase `Matrix` proporcionada por el SDK de Android, puesto que en la implementación de dicha función en las versiones anteriores a la API 10 (Android 2.3.3), se creaba una matriz auxiliar en tiempo de ejecución cada vez que se llamaba. Y, DroidEngine2D soporta desde la API 9 en adelante (aunque inicialmente soportaba desde la API 8, que corresponde a Android 2.2).

4.3.7.2.2. Diseño e implementación

Las utilidades que permiten trabajar con vectores de dos y tres dimensiones se han implementado en las clases `Vector2` y `Vector3`. Estas clases proporcionan métodos que permiten realizar algunas de las operaciones más comunes con vectores de dos y tres coordenadas.

En cuanto a las matrices de 4x4, se ha implementado la clase `Matrix4`, que proporciona métodos para trabajar con matrices de 4x4 elementos tipo `float`. Esta clase posee los principales métodos necesarios para trabajar con matrices en OpenGL (translación, rotación, escala, perspectiva, etc.). La implementación de todas estas operaciones se ha realizado haciendo uso de la clase `Matrix` del SDK de Android.

DroidEngine2D también proporciona la clase `TransformUtilities`, que permite aplicar transformaciones (translación, rotación y escala) sobre matrices de 4x4.

Y por último, la implementación alternativa del método `rotateM` de la clase `Matrix` se ha realizado en la clase `MatrixFix`. El método `rotateM` implementado en `MatrixFix` tiene el mismo comportamiento que el método `rotateM` de `Matrix`, la única diferencia es que el proporcionado por DroidEngine2D no crea objetos en tiempo de ejecución, mientras que el de la clase `Matrix`, en versiones antiguas del SDK de Android, sí.

4.3.7.2.3. Pruebas

El uso de vectores y matrices se realiza de forma intensiva en todo el motor, por lo que inicialmente se realizaron múltiples pruebas haciendo uso de la clase `Log` proporcionada por el SDK de Android y, posteriormente, en las demos se puede observar que todo funciona tal y como se espera.

4.3.7.3. Utilidades específicas de Android

En esta sección se explica el análisis, el diseño, la implementación y las pruebas de las utilidades específicas de Android que proporciona DroidEngine2D.

4.3.7.3.1. Análisis

Las utilidades específicas de Android proporcionadas por DroidEngine2D son dos, poder detectar si un dispositivo soporta OpenGL ES 2.0 y poder detectar la orientación por defecto del dispositivo.

Actualmente casi el 100% de dispositivos soportan OpenGL ES 2.0. No obstante, el emulador de Android no lo soporta en todas las versiones, y algunos dispositivos antiguos tampoco, por lo que poder comprobar si el dispositivo soporta OpenGL ES 2.0 es necesario.

En cuanto a detectar la orientación por defecto del dispositivo, es necesario por los motivos expuestos en el apartado 4.3.5.3.2, donde se explica el sistema de coordenadas por defecto de los sensores.

4.3.7.3.2. Diseño e implementación

Ambas utilidades se han implementado en la clase `ActivityUtilities` de DroidEngine2D y se pueden utilizar llamando a los métodos `detectOpenGLES20` y `getDefaultOrientationOfDevice`, respectivamente.

4.3.7.3.3. Pruebas

Las pruebas del método que permite detectar si un dispositivo soporta OpenGL ES 2.0 o no, se han realizado ejecutando el motor en un dispositivo antiguo sin soporte para OpenGL ES 2.0 y en un dispositivo más moderno que sí soporta OpenGL ES 2.0. Se ha podido comprobar que el funcionamiento es el esperado en ambos casos.

Y las pruebas de la obtención de la orientación por defecto del dispositivo se han realizado mediante mensajes de depuración, utilizando la clase `Log` proporcionada por el SDK de Android. Y el funcionamiento correcto se puede comprobar en las demos que hacen uso del acelerómetro.

4.4. Ejemplos de uso

Durante el desarrollo del proyecto, se ha desarrollado un conjunto de demos que se han utilizado para probar el funcionamiento del motor, tal y como se ha ido explicando en los apartados de pruebas de las características del motor descritas en el apartado 4.3.

Estas demos tienen principalmente dos propósitos. El primero de ellos, tal y como se ha descrito anteriormente, es verificar que cada una de las características del motor funciona tal y como se espera que funcione. Y el segundo propósito de estas demos es servir de ejemplos de uso de las características del motor por separado, y del motor en conjunto. Las demos del motor pretenden facilitar la iniciación al desarrollo de videojuegos con DroidEngine2D, sirviendo como complemento a la documentación del motor y la documentación de la API (javadoc).

A continuación se listarán todas las demos de características individuales implementadas y, posteriormente, se explicará más detalladamente el minijuego que se ha desarrollado para mostrar la estructura de un juego realizado con DroidEngine2D.

Es conveniente aclarar que la apariencia visual de las demos no es la que tendría un videojuego terminado. Las demos se han realizado principalmente para comprobar el funcionamiento del motor, y sus características, por lo que la apariencia visual no se ha considerado importante en este caso. Aunque sí lo sería en caso de haber pretendido desarrollar juegos completos.

4.4.1. Demos de características individuales

Las demos realizadas para las características individuales del motor son las que se exponen a continuación:

- Demos básicas de representación de rectángulos.
  - Representar rectángulos de distintas dimensiones.
  - Representar rectángulos rotados con respecto a su centro.
  - Representar rectángulos rotados con respecto a un punto especificado.
- Demos de TextureRegions.
  - Representar una imagen volteada horizontalmente y verticalmente.
- Demos de animaciones.
  - Uso de animaciones con el modo bucle activado.
  - Uso de animaciones con el modo bucle desactivado.
- Demos de representación de texto.
  - Representación de texto utilizando varias fuentes, con y sin rotación.
- Demos del uso de materiales.
  - Uso de opciones de fusión.
  - Representación de rectángulos con `ColorMaterial`.
  - Representación de rectángulos con `TextureMaterial`.
  - Representación de rectángulos con `TextureColorMaterial`.
  - Representación de rectángulos con `TextureHsvMaterial`.
  - Representación de rectángulos con `TransparentTextureMaterial`.
- Demos de batch rendering.
  - Representación de 512 rectángulos con el mismo solo material.
  - Representación de 256 rectángulos con un material, seguidos de 256 rectángulos con otro material distinto.
  - Representación de 257 rectángulos con un material, seguidos de 255 rectángulos con otro material distinto.
  - Representación de 256 rectángulos con un material y 256 rectángulos con otro material distinto. En esta demo, a diferencia de las 2 anteriores, nunca se representan 2 rectángulos seguidos con el mismo material.

- Demos de reproducción de audio.
    - Reproducción de una pista de audio desde un archivo. En esta demo se muestra la reproducción, la pausa y el control de volumen.
    - Reproducción de efectos de sonido cargados en memoria.
- Demos de entrada de usuario.
    - Procesamiento de entrada táctil.
    - Procesamiento de entrada mediante teclas físicas del dispositivo.
    - Captura de datos del acelerómetro.
- Demos de la gestión de estados del juego.
    - Intercambio de un estado activo del juego por otro.
    - Funcionamiento de la pila de estados del juego activos.

Estas demos se han ido explicando detalladamente a lo largo del apartado 4.3, en el apartado de pruebas de cada una de las características del motor. Por tanto, en este apartado no se entrará en detalle acerca del funcionamiento de cada una de ellas, ni se repetirán las conclusiones obtenidas al realizar estos experimentos.

4.4.2. Demo de un minijuego desarrollado con DroidEngine2D

Finalmente, tras desarrollar todas las demos de las características individuales del motor, se ha desarrollado un minijuego que sirve de ejemplo de uso del motor en conjunto. Este minijuego es una versión muy simplificada del juego *Breakout*.

Un nivel del juego está formado por un conjunto de bloques apilados en la parte de arriba de la pantalla, un bloque que se desplaza horizontalmente en la parte de debajo de la pantalla y está controlado por el jugador, y una pelota que rebota al colisionar con las paredes de la pantalla o los bloques.

Si la pelota colisiona con el bloque controlado por el jugador o en las paredes, rebota. Si colisiona con uno de los bloques situados en la parte superior de la pantalla, rebota y destruye dicho bloque. Y, si la pelota se sale por la parte inferior de la pantalla, el jugador pierde la partida. El jugador completa el nivel cuando consigue hacer que la pelota destruya todos los bloques situados en la parte superior de la pantalla.

La demo cuenta con un único nivel, puesto que el objetivo era desarrollar un minijuego simple que hiciese uso de las principales características de DroidEngine2D. No obstante, extenderla agregándole más niveles no sería una tarea complicada.

Al ejecutar el minijuego aparece el menú principal, que está formado simplemente por un logo animado y un botón que permite comenzar a jugar. Al pulsar dicho botón, se sustituye el estado que representa al menú principal por el estado que representa un nivel del juego. Cuando esto ocurre, la pila de estados activos solo tiene el nivel del juego, y el jugador puede comenzar a jugar dicho nivel. Llegado este punto, solo hay dos escenarios posibles (aparte de cerrar el juego, lo cual se puede hacer en cualquier momento): que el jugador consiga hacer que la pelota destruya todos los bloques y gane

la partida, o que el jugador no consiga evitar que la pelota salga por debajo de la pantalla y pierda la partida.

Si el jugador gana la partida, se apila un nuevo estado del juego que muestra una ventana flotante sobre el estado del juego anterior, indicando al jugador que ha ganado y permitiéndole volver al menú principal. Y si pierde, ocurre lo mismo, pero en este caso, se le indica al jugador que ha perdido.

A continuación se puede observar el diagrama de estados que representa el funcionamiento descrito anteriormente.

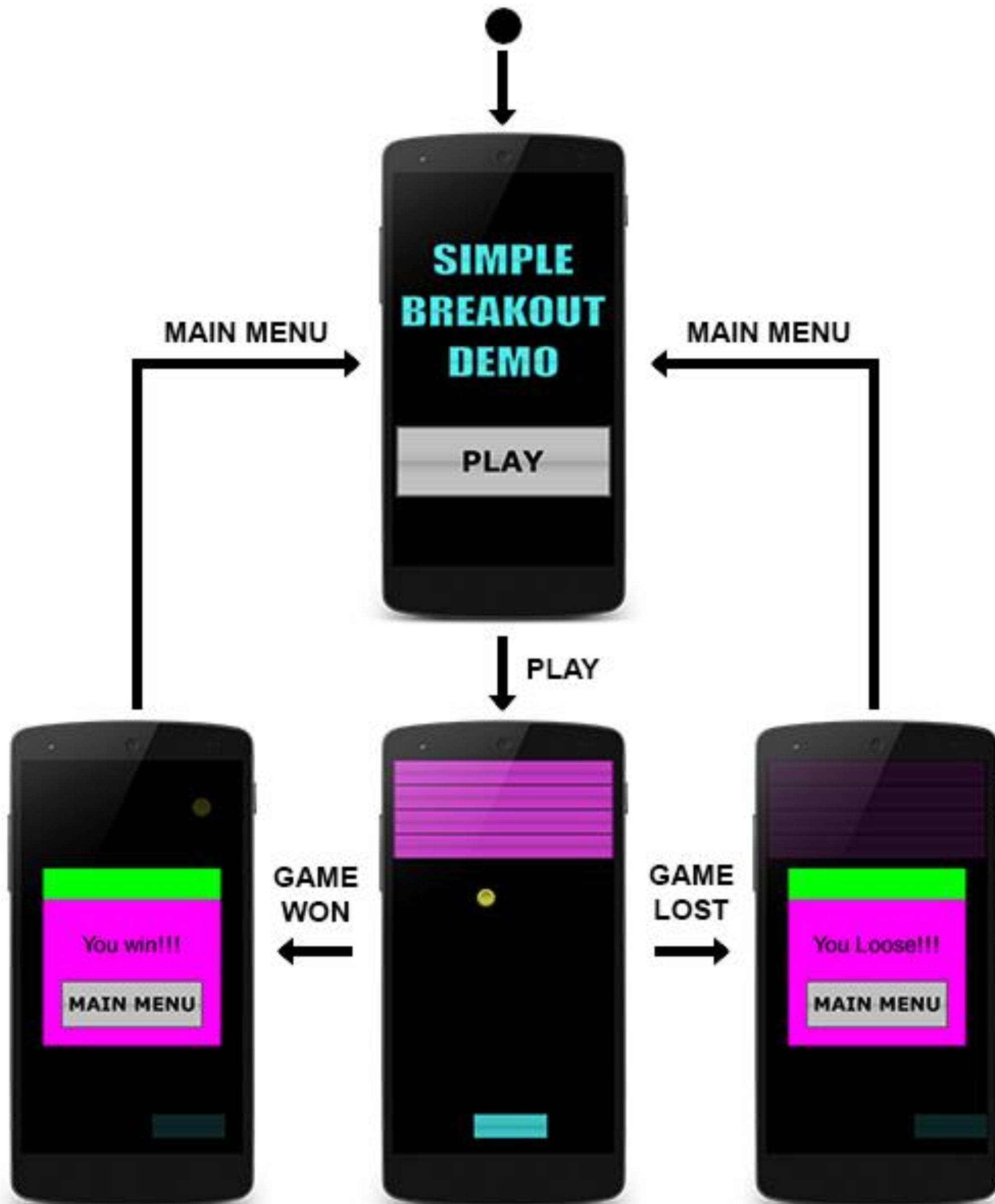

Figura 36. Diagrama de estados del minijuego de demostración.

Además, desde que comienza una partida hasta que acaba, se reproduce una pista de audio como música de fondo en modo bucle, y cada vez que la pelota colisiona con un objeto, se reproduce un efecto de sonido.

Otro detalle a mencionar es que las ventanas flotantes no son imágenes, sino que se han definido directamente en el código de la demo y se representan utilizando rectángulos de colores. Se han implementado de esta forma para mostrar que DroidEngine2D puede hacer uso de distintos materiales simultáneamente, aunque esto ya se ha podido ver en otras demos.

Por último, mencionar que el jugador puede controlar el bloque inferior haciendo uso del acelerómetro del dispositivo.

En resumen, el minijuego de demostración hace uso de las principales características del motor y puede servir, tanto de ejemplo de funcionamiento, como de ejemplo de uso de las mismas.

5

Conclusiones

5.1. Conclusiones generales

El motor de juegos desarrollado es una herramienta bastante completa que permite crear videojuegos en dos dimensiones para dispositivos Android, siempre que el usuario tenga conocimientos del lenguaje de programación Java y unas nociones mínimas del funcionamiento de las aplicaciones en dicho sistema.

Además de ser útil para desarrollar videojuegos, este proyecto ha permitido al autor adquirir conocimientos sobre bastantes temas relacionados con el mismo y puede servir a otras personas como fuente de información sobre dichos temas, entre los que se pueden destacar los siguientes:

- El desarrollo de videojuegos en general.
- La arquitectura y el funcionamiento de un motor de juegos.
- Los principales motores de juegos disponibles en la actualidad (Julio, 2014).
- El funcionamiento de las aplicaciones en Android.
- El funcionamiento de OpenGL ES en Android.
- El diseño y la implementación de las principales características de un motor de juegos 2D para Android.
- La implementación de técnicas de batch rendering utilizando OpenGL ES.
- La implementación de optimizaciones como reducir al mínimo el número de objetos creados durante la ejecución del juego. Para esto en DroidEngine2D se

utilizan object pools (piscinas de objetos), y otras estructuras de datos que evitan, en la medida de lo posible, instanciar objetos una vez que han sido creadas e inicializadas por primera vez.

- Algunos de los aspectos de más bajo nivel con los que ha habido que tratar a lo largo del desarrollo. Por ejemplo, la gestión de eventos de entrada de usuario, o el sistema de coordenadas del acelerómetro.

Toda esta información se puede obtener en este mismo documento, en la documentación de la API de DroidEngine2D (javadoc) y/o en el código fuente del proyecto y las demos.

Aparte de los conocimientos expuestos anteriormente, el autor ha puesto en práctica los conocimientos adquiridos durante los 5 años de carrera. Entre estos conocimientos se encuentra el uso del paradigma de programación orientado a objetos, el desarrollo de aplicaciones utilizando el lenguaje de programación Java, el uso de patrones de diseño cuando la situación lo requería, algunas prácticas de ingeniería del software, el uso de un sistema de control de versiones, etc.

Como ya se ha mencionado anteriormente en este documento, el proyecto se ha desarrollado siguiendo una metodología iterativa e incremental. La planificación del desarrollo se ha llevado a cabo utilizando un tablero de tareas que ha permitido en todo momento visualizar el estado del proyecto, es decir, lo que se está desarrollando, lo que ya se ha completado, y lo que falta por desarrollar. Esto puede no parecer necesario por ser un proyecto desarrollado por una sola persona, sin embargo, al compaginar el desarrollo con los últimos cursos de la carrera, ha resultado bastante útil tener un punto de referencia donde poder recordar qué es lo último en lo que se estaba trabajando y lo que falta por hacer.

En resumen, el desarrollo de este proyecto, además de tener como resultado una herramienta de código abierto bastante completa para desarrollar videojuegos 2D para Android, ha servido al autor para aplicar algunos de los conocimientos adquiridos durante la carrera y para adquirir otros conocimientos nuevos, y puede servir a otras personas como fuente de información sobre los temas expuestos anteriormente en este apartado, entre otros.

5.2. Posibles mejoras y trabajos futuros

Aunque actualmente (Julio, 2014), DroidEngine2D es una herramienta bastante completa que permite desarrollar videojuegos 2D para Android, hay algunos aspectos que se pueden mejorar y hay algunas características que sería deseable agregar al motor en futuras versiones. A continuación se expone un listado con algunas de estas mejoras:

- **Sistema de detección de colisiones:** Se podría integrar alguna librería de físicas de código abierto, como Box2D, o implementar un sistema de detección de colisiones en dos dimensiones en DroidEngine2D, ya que actualmente (Julio, 2014), el usuario tiene que integrar manualmente con el motor una librería externa, o implementar su propio sistema de detección de colisiones.

- **Posibilidad de representar otros polígonos con batch rendering:** Con DroidEngine2D solo se pueden representar rectángulos utilizando batch rendering. Si el usuario necesitara representar triángulos tendría que implementar dicha funcionalidad él mismo, o bien utilizar rectángulos con texturas que simulen triángulos. La mejora consistiría en dar soporte a la representación de triángulos para que el usuario no tuviese que recurrir a estas soluciones.
- **Implementación de tests unitarios:** Actualmente (Julio, 2014), con el motor se proporciona un conjunto de demos que permiten comprobar el funcionamiento de sus principales características. No obstante, sería muy conveniente implementar tests unitarios que permitan verificar el funcionamiento del motor de forma más precisa.
- **Soporte por defecto para un mayor número de formatos de fuente:** Actualmente (Julio, 2014), DroidEngine2D, por defecto solo permite cargar fuentes en formato bitmap definidas con la aplicación externa BMFont en el formato explicado en el apéndice A. Y, aunque es posible que el usuario defina otros formatos, sería conveniente proporcionar soporte por defecto para otros formatos comunes, ya que BMFont solo está disponible para Windows.
- **Mejora del proceso de integración de cargadores de fuentes implementados por el usuario en el motor:** Actualmente (Julio, 2014), el usuario puede definir cargadores de fuentes en otros formatos distintos al de BMFont si lo necesita, e integrarlos en el motor. Pero dicha integración no es un proceso trivial, por lo que se podría mejorar el diseño de esta característica de forma que el motor fuese más fácilmente extensible en este aspecto.
- **Soporte por defecto para un mayor número de formatos de atlas de texturas:** Actualmente (Julio, 2014), DroidEngine2D, por defecto solo permite cargar atlas de texturas definidos con la aplicación externa TexturePacker en el formato explicado en el apéndice B. Y, aunque es posible que el usuario defina otros formatos, sería conveniente proporcionar soporte por defecto para otros formatos comunes, como JSON, de forma que el usuario no tenga que implementar el cargador de atlas de texturas en dicho formato manualmente.
- **Mejora del aspecto visual de las demos del motor:** Las demos del motor se han desarrollado con los objetivos de verificar el comportamiento de las características del motor y servir de ejemplos de uso de las mismas. Una posible mejora a desarrollar en el futuro podría consistir en renovar el aspecto de las demos, de forma que fuesen más atractivas visualmente.

6

Bibliografía

6.1. Referencias

[1] Jason Gregory. *Game Engine Architecture*. A. K. Peters / CRC Press, 2009.

[2] Colaboradores de Wikipedia. *Doom (video game)*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 4 de Abril de 2014]. Disponible en: http://en.wikipedia.org/wiki/Doom_(video_game).

[3] Colaboradores de Wikipedia. *Videojuego de disparos en primera persona*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 13 de Abril de 2014]. Disponible en: http://es.wikipedia.org/wiki/Videojuego_de_disparos_en_primera_persona.

[4] Colaboradores de Wikipedia. *Platform game*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 13 de Abril de 2014]. Disponible en: http://en.wikipedia.org/wiki/Platform_game.

[5] Colaboradores de Wikipedia. *Fighting game*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 13 de Abril de 2014]. Disponible en: http://en.wikipedia.org/wiki/Fighting_game.

[6] Colaboradores de Wikipedia. *Racing video game*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 17 de Abril de 2014]. Disponible en: http://en.wikipedia.org/wiki/Racing_video_game.

[7] Colaboradores de Wikipedia. *Real-time strategy*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 17 de Abril de 2014]. Disponible en: http://en.wikipedia.org/wiki/Real-time_strategy.

[8] Colaboradores de Wikipedia. *Heightmap*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 17 de Abril de 2014]. Disponible en: http://en.wikipedia.org/wiki/Heightmap.

[9] Colaboradores de Wikipedia. *Massively multiplayer online game*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 17 de Abril de 2014]. Disponible en: http://en.wikipedia.org/wiki/MMOG.

[10] Antonio L. Corral Liria. *Transparencias de Diseño de Sistemas Operativos*. Teoría de la asignatura de Diseño de Sistemas Operativos. Universidad de Almería, 2013-2014.

[11] Colaboradores de Wikipedia. *Viewport*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 6 de Mayo de 2014]. Disponible en: http://en.wikipedia.org/wiki/Viewport.

[12] Colaboradores de Wikipedia. *Shader*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 6 de Mayo de 2014]. Disponible en: http://en.wikipedia.org/wiki/Shader.

[13] Colaboradores de Wikipedia. *Space partitioning*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 6 de Mayo de 2014]. Disponible en: http://en.wikipedia.org/wiki/Space_partitioning.

[14] David H. Eberly. 3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic. Morgan Kaufmann Publishers, 2005.

[15] Colaboradores de Wikipedia. *Full screen effect*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 6 de Mayo de 2014]. Disponible en: http://en.wikipedia.org/wiki/Full_screen_effect.

[16] Colaboradores de Wikipedia. *Mecánica del sólido rígido*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 6 de Mayo de 2014]. Disponible en: http://es.wikipedia.org/wiki/Mec%C3%A1nica_del_s%C3%B3lido_r%C3%ADgido.

[17] Eric Holec. *The Game Loop*. Entropy Interactive, 2011 [última fecha de consulta: 13 de Junio de 2014]. Disponible en: http://entropyinteractive.com/2011/02/game-engine-design-the-game-loop/

[18] Glenn Fiedler. *Fix Your Timestep!*. Glenn Fiedler's Game Development Articles and Tutorials, 2006 [última fecha de consulta: 13 de Junio de 2014]. Disponible en: http://gafferongames.com/game-physics/fix-your-timestep/.

[19] Android Team at Google. *Application Fundamentals*. Android Developers, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: http://developer.android.com/guide/components/fundamentals.html.

[20] Android Team at Google. *Activities*. Android Developers, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: http://developer.android.com/guide/components/activities.html.

[21] Android Team at Google. *Services*. Android Developers, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: http://developer.android.com/guide/components/services.html.

[22] Android Team at Google. *Content Providers*. Android Developers, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: http://developer.android.com/guide/topics/providers/content-providers.html.

[23] Unity Technologies. *What is Unity?* Official site of Unity, 2014 [última fecha de consulta: 10 de Mayo de 2014]. Disponible en: http://unity3d.com/pages/what-is-unity.

[24] Epic Games. *Unreal Engine. Frequently Asked Questions (FAQ)*. Official site of Unreal Engine [última fecha de consulta: 10 de Mayo de 2014]. Disponible en: https://www.unrealengine.com/faq.

[25] Epic Games. *UDK*. Official site of Unreal Engine, 2014 [última fecha de consulta: 14 de Mayo de 2014]. Disponible en: https://www.unrealengine.com/products/udk.

[26] Epic Games. *UDK Licensing*. Official site of Unreal Engine, 2014 [última fecha de consulta: 14 de Mayo de 2014]. Disponible en: https://www.unrealengine.com/products/udk/udk-licensing-resources#faq.

[27] Colaboradores de Wikipedia. *CryEngine*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 21 de Mayo de 2014]. Disponible en: http://en.wikipedia.org/wiki/CryEngine.

[28] Crytek. *CryEngine 3*. Official site of Crytek, 2014 [última fecha de consulta: 21 de Mayo de 2014]. Disponible en: http://www.crytek.com/cryengine/cryengine3/overview.

[29] Torus Knot Software Ltd and the community of OGRE3D. *Ogre wiki*. Official site of OGRE3D, 2014 [última fecha de consulta: 22 de Mayo de 2014]. Disponible en: http://www.ogre3d.org/tikiwiki/tiki-index.php.

[30] Developers of OGE. *The Open Game Engine*. OGE wiki on Sourceforge, 2014 [última fecha de consulta: 22 de Mayo de 2014]. Disponible en: http://sourceforge.net/apps/mediawiki/oge/index.php?title=Main_Page.

[31] Mario Zechner. *LibGDX*. Official site of LibGDX, 2014 [última fecha de consulta: 26 de Mayo de 2014]. Disponible en: http://libgdx.badlogicgames.com/.

[32] Mario Zechner. *LibGDX. Goals and Features*. Official site of LibGDX, 2014 [última fecha de consulta: 26 de Mayo de 2014]. Disponible en: http://libgdx.badlogicgames.com/features.html.

[33] Mario Zechner. *Beginning Android Games*. Apress, 2011.

[34] Andreas Oehlke. *Learning Libgdx Game Development*. Packt Publishing, 2013.

[35] Nicolas Gramlich. *AndEngine*. Official site of AndEngine, 2014 [última fecha de consulta: 26 de Mayo de 2014]. Disponible en: http://www.andengine.org/.

[36] Nicolas Gramlich. *AndEngine*. GitHub, 2014 [última fecha de consulta: 27 de Mayo de 2014]. Disponible en: https://github.com/nicolasgramlich/AndEngine.

[37] Khronos Group. *OpenGL ES. The Standard for Embedded Accelerated 3D Graphics*. Official site of the Khronos Group, 2014 [última fecha de consulta: 28 de Mayo de 2014]. Disponible en: http://www.khronos.org/opengles/.

[38] David Blythe. *OpenGL ES Common/Common-Lite Profile Specification. Version 1.0.02*. The Khronos Group Inc, 2002-2004 [última fecha de consulta: 29 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/1.0/opengles_spec_1_0.pdf.

[39] David Blythe, Aaftab Munshi and Jon Leech. *OpenGL ES Common/Common-Lite Profile Specification. Version 1.1.12*. The Khronos Group Inc, 2002-2007 [última fecha de consulta: 29 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/1.1/es_full_spec_1.1.12.pdf.

[40] Khronos Group. *OpenGL ES. The Standard for Embedded Accelerated 3D Graphics. OpenGL ES 1.X*. Official site of the Khronos Group, 2014 [última fecha de consulta: 29 de Mayo de 2014]. Disponible en: http://www.khronos.org/opengles/1_X/.

[41] Khronos Group. *OpenGL ES. The Standard for Embedded Accelerated 3D Graphics. OpenGL ES 2.X*. Official site of the Khronos Group, 2014 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://www.khronos.org/opengles/2_X/.

[42] Aaftab Munshi and Jon Leech. *OpenGL ES Common/Common-Lite Profile Specification. Version 2.0.25*. The Khronos Group Inc, 2002-2010 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

[43] John Kessenich and Robert J. Simpson. *The OpenGL ES Shading Language. Version 1.0.17*. The Khronos Group Inc, 2006-2009 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf.

[44] Benj Lipchak. *OpenGL ES Version 3.0.3*. The Khronos Group Inc, 2006-2013 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf.

[45] Colaboradores de Wikipedia. *Geometry Instancing*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://en.wikipedia.org/wiki/Geometry_instancing.

[46] Colaboradores de Wikipedia. *Render Target*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://en.wikipedia.org/wiki/Render_Target.

[47] John Kessenich, Dave Baldwin, Randi Rost, Robert J. Simpson. *The OpenGL ES Shading Language Version 3.00.4*. The Khronos Group Inc, 2008-2013 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/3.0/GLSL_ES_Specification_3.00.4.pdf.

[48] John Leech. *OpenGL ES Version 3.1*. The Khronos Group Inc, 2006-2014 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/3.1/es_spec_3.1.pdf.

[49] John Kessenich, Dave Baldwin, Randi Rost, Robert J. Simpson. *The OpenGL ES Shading Language Version 3.10.2*. The Khronos Group Inc, 2008-2014 [última fecha de consulta: 30 de Mayo de 2014]. Disponible en: http://www.khronos.org/registry/gles/specs/3.1/GLSL_ES_Specification_3.10.pdf.

[50] Mason Woo, Jackie Neider and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1. Second Edition*. Addison-Wesley, 1997.

[51] Dave Shreiner, Graham Sellers, John M. Kessenich, Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3. 8th Edition*. Addison-Wesley, 2013.

[52] Aaftab Munshi, Dan Ginsburg and Dave Shreiner. *OpenGL ES 2.0 Programming Guide*. Addison-Wesley, 2009.

[53] Jeremiah van Oosten. *Introduction to OpenGL and GLSL*. 3D Game Engine Programming, 2014 [última fecha de consulta: 1 de Junio de 2014]. Disponible en: http://3dgep.com/?p=5303.

[54] Khronos Group. *Primitive*. OpenGL Wiki, 2013 [última fecha de consulta: 1 de Junio de 2014]. Disponible en: http://www.opengl.org/wiki/Primitive.

[55] Egon Rath. *Understanding GL_TRIANGLE_STRIP*. Egon Rath's Notes, 2014 [última fecha de consulta: 1 de Junio de 2014]. Disponible en: http://www.matrix44.net/cms/notes/opengl-3d-graphics/understanding-gl_triangle_strip.

[56] Alexander Overvoorde. *Extra buffers*. open.gl, 2014 [última fecha de consulta: 7 de Junio de 2014]. Disponible en: http://open.gl/depthstencils.

[57] Android Team at Google. *Android 4.3, Jelly Bean*. Official site of Android, 2013 [última fecha de consulta: 11 de Junio de 2014]. Disponible en: http://www.android.com/about/jelly-bean/.

[58] Android Team at Google. *Dashboards*. Android Developers, 2014 [última fecha de consulta: 11 de Junio de 2014]. Disponible en: https://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net.

[59] Android Team at Google. *GLSurfaceView*. Android Developers, 2014 [última fecha de consulta: 11 de Junio de 2014]. Disponible en: http://developer.android.com/reference/android/opengl/GLSurfaceView.html.

[60] Khronos Group. *EGL. Native Platform Interface*. Official site of the Khronos Group, 2014 [última fecha de consulta: 11 de Junio de 2014]. Disponible en: https://www.khronos.org/egl.

[61] Miguel Vicente Linares. *DroidEngine2D*. GitHub, 2014 [última fecha de consulta: 16 de Junio de 2014]. Disponible en: https://github.com/miviclin/droidengine2d.

[62] Miguel Vicente Linares. *DroidEngine2D API Documentation*. miviclin.github.io, 2014 [última fecha de consulta: 16 de Junio de 2014]. Disponible en: http://miviclin.github.io/droidengine2d-docs/.

[63] Miguel Vicente Linares. *DroidEngine2D Demos*. GitHub, 2014 [última fecha de consulta: 16 de Junio de 2014]. Disponible en: https://github.com/miviclin/droidengine2d-demos.

[64] Miguel Vicente Linares. *Collections*. GitHub, 2014 [última fecha de consulta: 16 de Junio de 2014]. Disponible en: https://github.com/miviclin/collections.

[65] Android Team at Google. *Get the Android SDK*. Android Developers, 2014 [última fecha de consulta: 16 de Junio de 2014]. Disponible en: http://developer.android.com/sdk/index.html.

[66] The Eclipse Foundation. *Eclipse Desktop and Web IDEs*. The Eclipse Foundation open source community website, 2014. [última fecha de consulta: 16 de Junio de 2014]. Disponible en: http://www.eclipse.org/ide/.

[67] Android Team at Google. *ADT Plugin*. Android Developers, 2014 [última fecha de consulta: 16 de Junio de 2014]. Disponible en: http://developer.android.com/tools/sdk/eclipse-adt.html.

[68] Git Contributors. *Git*. Git, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: http://git-scm.com/.

[69] Vincent Driessen. *A successful Git branching model*. Personal blog of Vincent Driessen: nvie.com, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: http://nvie.com/posts/a-successful-git-branching-model/.

[70] GitHub Inc. GitHub. GitHub, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: https://github.com/.

[71] Colaboradores de Wikipedia. *Kanban board*. Wikipedia, La Enciclopedia Libre, 2014 [última fecha de consulta: 22 de Junio de 2014]. Disponible en: http://en.wikipedia.org/wiki/Kanban_board.

[72] Android Team at Google. *MediaPlayer*. Android Developers, 2014 [última fecha de consulta: 25 de Junio de 2014]. Disponible en: http://developer.android.com/reference/android/media/MediaPlayer.html.

[73] Android Team at Google. *SoundPool*. Android Developers, 2014 [última fecha de consulta: 25 de Junio de 2014]. Disponible en: http://developer.android.com/reference/android/media/SoundPool.html.

[74] Khronos Group. *Blending*. OpenGL Wiki, 2013 [última fecha de consulta: 6 de Julio de 2014]. Disponible en: http://www.opengl.org/wiki/Blending.

[75] Silicon Graphics. *glBlendEquation*. Official website of the Kronos Group, 1991-2006 [última fecha de consulta: 6 de Julio de 2014]. Disponible en: https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBlendEquation.xml.

[76] Silicon Graphics. *glBlendFunc*. Official website of the Kronos Group, 1991-2006 [última fecha de consulta: 6 de Julio de 2014]. Disponible en: https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBlendFunc.xml.

[77] Anton Holmquist. *Sprite Batching in OpenGL ES 2.0*. Developer blog of Anton Holmquist, 2012 [última fecha de consulta: 7 de Julio de 2014]. Disponible en: http://antonholmquist.com/blog/sprite-batching-opengl-es-2-0/.

[78] Andreas Jönsson. *Bitmap Font Generator*. AngelCode.com, 2014 [última fecha de consulta: 8 de Julio de 2014]. Disponible en: http://www.angelcode.com/products/bmfont/.

[79] CodeAndWeb GmbH. *TexturePacker*. CodeAndWeb, 2014 [última fecha de consulta: 9 de Julio de 2014]. Disponible en: https://www.codeandweb.com/texturepacker.

[80] Miguel Vicente Linares. *Diagrama de clases de DroidEngine2D*. Dropbox, 2014 [última fecha de consulta: 14 de Julio de 2014]. Disponible en; https://www.dropbox.com/s/ozgy8gqj3kikw5b/Diagrama%20de%20clases%20de%20DroidEngine2D%20-%20A0.pdf.

BMFont

A.1. Introducción

En este apéndice se expone brevemente qué es BMFont, cómo se puede obtener, y, finalmente, se explica paso a paso cómo utilizar BMFont para generar una fuente en formato bitmap compatible con la clase `BitmapFont` de DroidEngine2D.

A.2. ¿Qué es BMFont?

BMFont es un generador de fuentes en formato bitmap a partir de fuentes en formato TrueType.

Esta aplicación genera tanto los atlas de texturas como el archivo XML que contiene las descripciones de cada uno de los caracteres de la fuente.

A.3. Obtener BMFont

BMFont es un programa gratuito de código abierto y, tanto el código fuente como el instalador, se pueden obtener online a través de la referencia [78].

A.4. Generar fuentes compatibles con DroidEngine2D

Para generar fuentes compatibles con la clase `BitmapFont` de DroidEngine2D tan solo hay que seguir los pasos que se exponen a continuación.

Nota: Las explicaciones expuestas a continuación se basan en la versión 1.13 de BMFont.

En primer lugar, es necesario seleccionar la configuración de la fuente. Para ello, hay que abrir la ventana de configuración en el menú *Options* > *Font settings*. Y, en dicha ventana, hay que especificar la configuración de la fuente tal y como se muestra en la siguiente figura. Teniendo en cuenta que algunos parámetros se pueden variar, como la fuente seleccionada, por ejemplo.

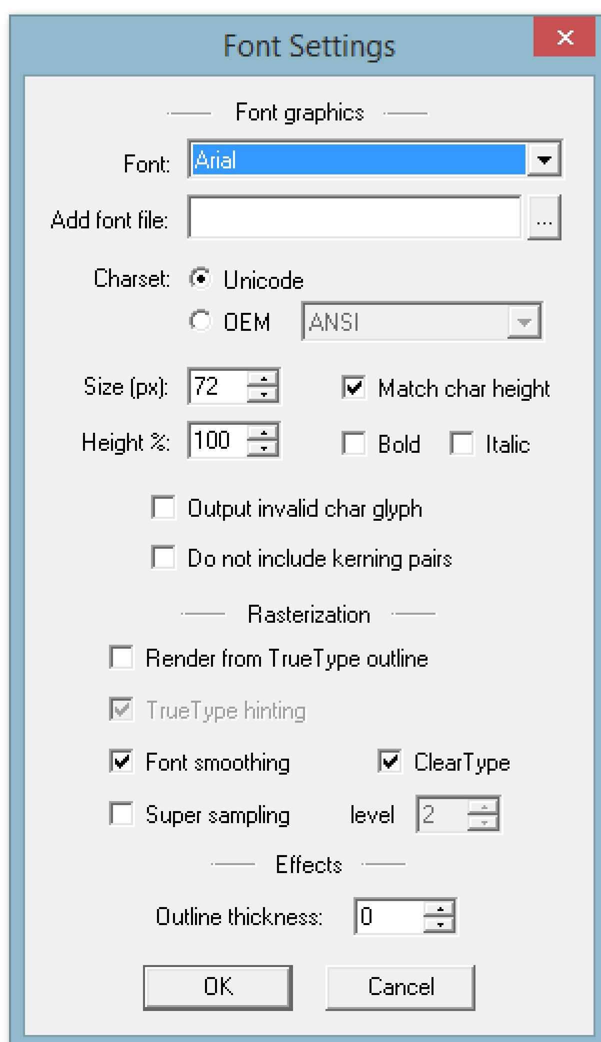

Figura 37. Configuración de la fuente en BMFont.

Como se puede ver, la fuente se puede seleccionar de entre las fuentes instaladas en el sistema, o bien especificando manualmente la ruta de un archivo de fuente en formato TrueType (.ttf).

Es importante también marcar la opción “Match char height” y seleccionar el tamaño, en píxeles. En la figura anterior se ha fijado el tamaño en 72 píxeles, sin embargo, según la resolución en la que se vaya a mostrar la aplicación y el tamaño al que se vaya a mostrar la fuente, es posible que sea necesario exportar la fuente a un tamaño mayor.

Tras configurar la fuente, hay que configurar las opciones de exportación. Esta ventana se abre desde el menú *Options* > *Export options* y debe configurarse tal y como se muestra en la siguiente figura.

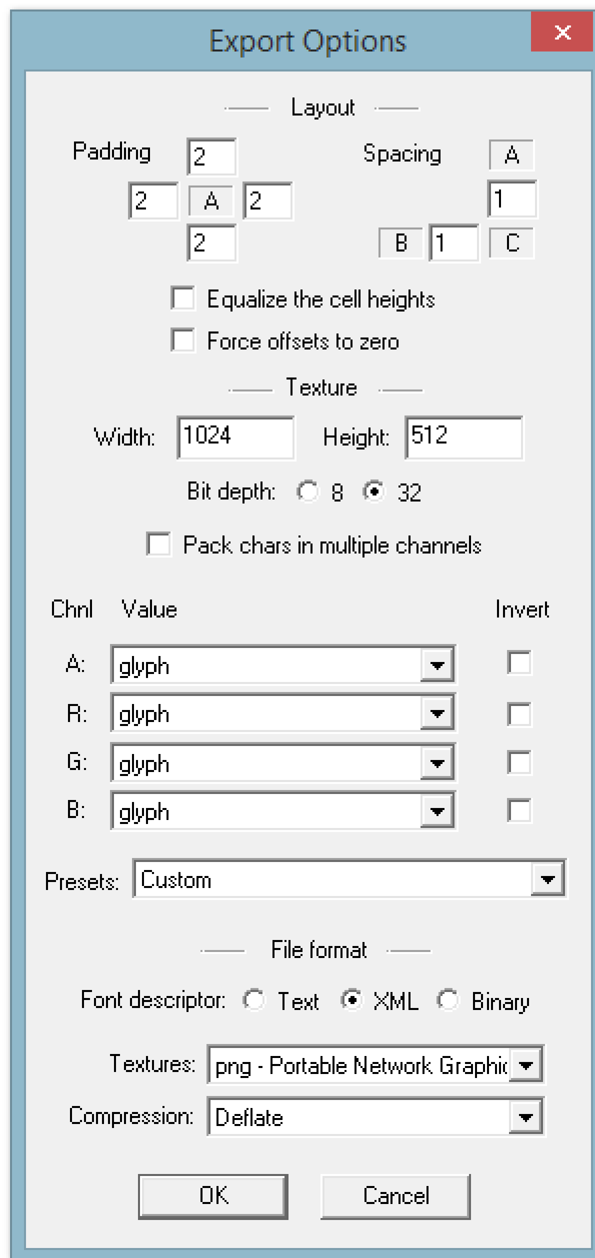

Figura 38. Opciones de exportación en BMFont.

En esta ventana es importante que el padding y el spacing sean mayores que cero para evitar posibles errores al obtener las regiones del atlas de texturas correspondientes a cada carácter.

Además hay que seleccionar un bit depth de 32 para soportar transparencias y todos los canales deben tener el valor “glyph”.

Por último, es importante que en esta ventana se marque el formato XML para la descripción de la fuente y el formato PNG para los atlas de texturas. BMFont generará un archivo con extensión .fnt, pero internamente la fuente estará definida en formato XML.

Una vez seleccionadas las opciones de la fuente y las opciones de exportación, hay que seleccionar los caracteres de la fuente que se desean exportar en formato bitmap. Para ello, hay que marcar dichos conjuntos de caracteres en la lista que aparece a la derecha en ventana principal de la aplicación, tal y como se muestra en la siguiente figura.

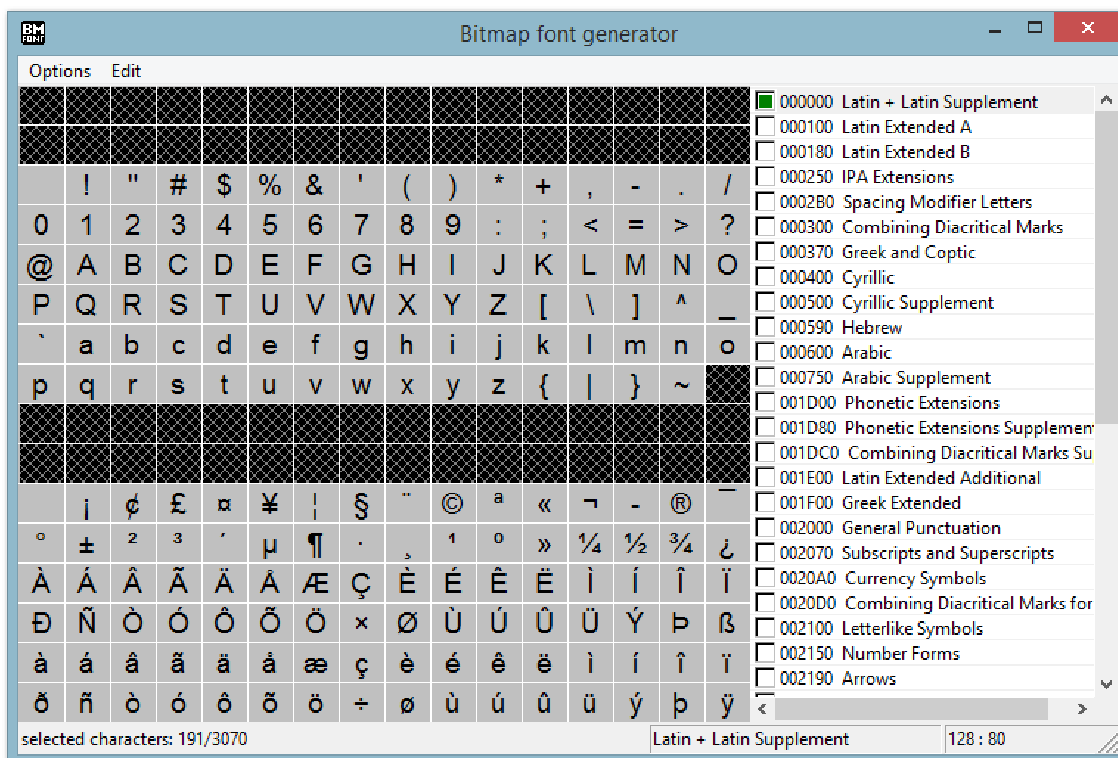

Figura 39. Ventana principal de BMFont.

Finalmente, tras haber realizado todos los pasos anteriores, se puede exportar la fuente en formato bitmap desde el menú *Options > Save bitmap font as…*

B

TexturePacker

B.1. Introducción

En este apéndice se expone brevemente qué es TexturePacker [79], cómo se puede obtener, y, finalmente, se explica paso a paso cómo utilizar TexturePacker para generar un atlas de texturas compatible con la clase `TexturePackerAtlas` de DroidEngine2D.

B.2. ¿Qué es TexturePacker?

TexturePacker es una herramienta multiplataforma que permite generar atlas de texturas a partir de imágenes de forma muy sencilla. Está disponible para Windows, Mac OS y Linux.

Esta herramienta permite generar atlas de texturas en múltiples formatos; desde formatos genéricos como XML o JSON, hasta formatos específicos de motores de juegos concretos.

B.3. Obtener TexturePacker

TexturePacker se puede obtener online a través de la referencia [78]. La descarga es gratuita, aunque para utilizar todas las características del producto es necesario adquirir una licencia. No obstante, el formato que utiliza DroidEngine2D por defecto no requiere ninguna característica que no se incluya en la versión gratuita.

B.4. Generar atlas compatibles con DroidEngine2D

A continuación se explican los pasos a seguir para generar un atlas de texturas compatible con la clase `TexturePackerAtlas` de DroidEngine2D utilizando la versión gratuita (lite) de TexturePacker.

Nota: Las explicaciones expuestas a continuación se basan en la versión 3.2.1 de TexturePacker.

Tras descargar e instalar TexturePacker, al abrirlo aparecerá una ventana en la que se debe seleccionar la versión a utilizar. Si no se dispone de licencia, se puede seleccionar la opción gratuita (lite).

Una vez dentro de la aplicación, en el menú que aparece a la izquierda de la ventana, se debe seleccionar la configuración que se describe a continuación.

En primer lugar, en la sección "Output", se debe seleccionar el formato de datos "Generic XML" y se debe especificar la ruta donde se guardará el archivo de datos y el archivo de textura generados, tal y como se muestra en la siguiente figura.

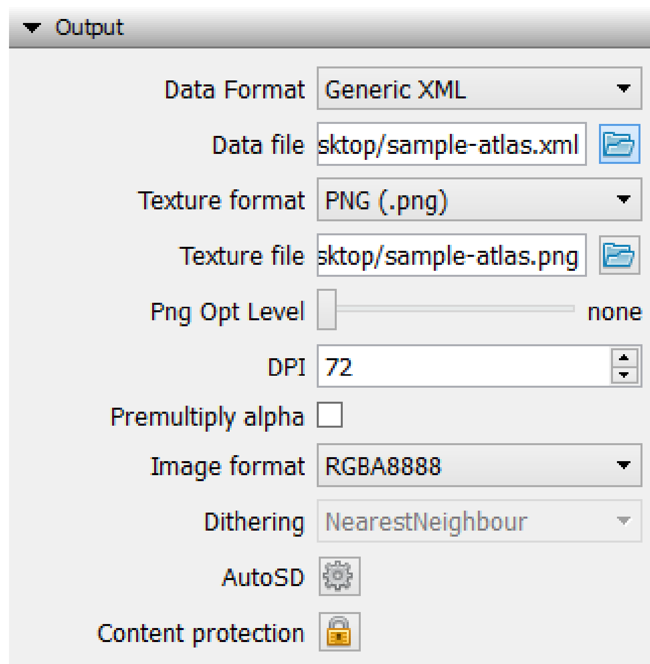

Figura 40. Sección Output de la configuración del atlas de texturas en TexturePacker.

En la sección "Geometry" no es necesario modificar nada. Es conveniente que el ancho y el alto de la textura sean potencias de dos, por restricciones de OpenGL ES. Y las dimensiones no es recomendable que superen 2048x2048 píxeles, puesto que podrían surgir incompatibilidades con algunos dispositivos. Incluso, si se pretende dar soporte a dispositivos antiguos, es probable que deban reducirse a 1024x1024 píxeles como máximo.

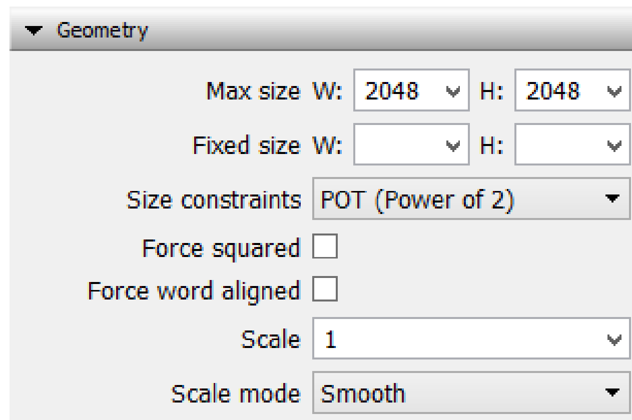

Figura 41. Sección Geometry de la configuración del atlas de texturas en TexturePacker.

En la sección "Layout" se debe seleccionar el algoritmo "Basic" y desactivar la rotación, el recorte (trim) y la opción de auto alias. También es recomendable que el padding sea mayor que cero, tal y como se muestra a continuación.

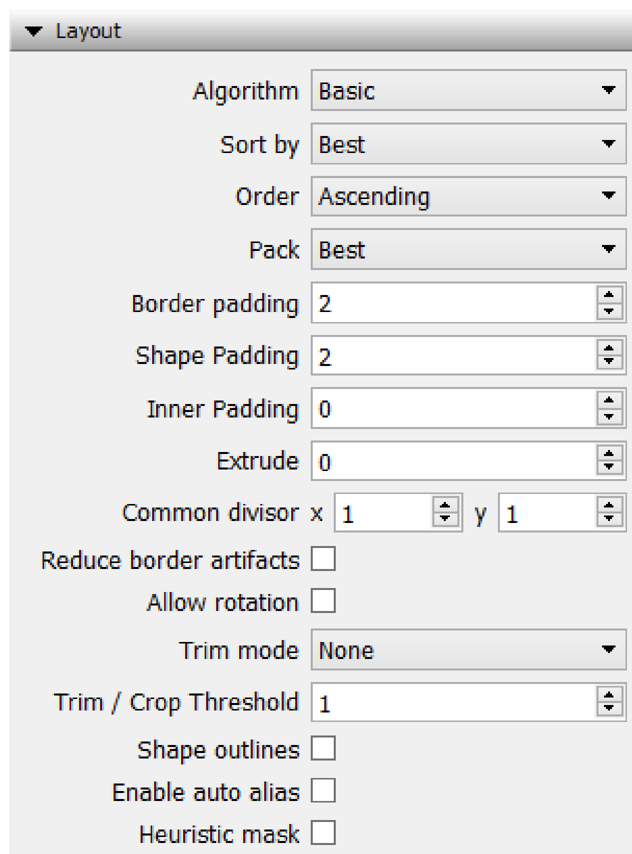

Figura 42. Sección Layout de la configuración del atlas de texturas en TexturePacker.

Por último, en la sección "Advanced" se pueden dejar los valores que fija TexturePacker por defecto.

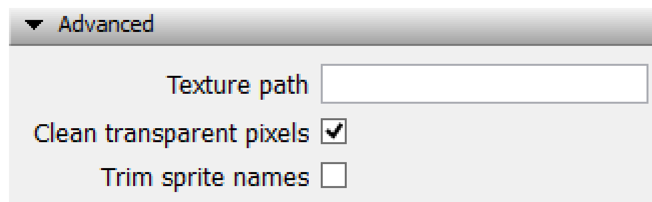

Figura 43. Sección Advanced de la configuración del atlas de texturas en TexturePacker.

Una vez realizada la configuración del atlas de texturas a generar, se deben arrastrar las imágenes que se desee incluir en el atlas sobre la región blanca que aparece a la derecha de la ventana de TexturePacker. Tras realizar este paso, se debe de poder ver en el centro de la ventana una vista previa del atlas de texturas que se generará, y en la ventana de sprites se pueden ver las imágenes incluidas en el atlas.

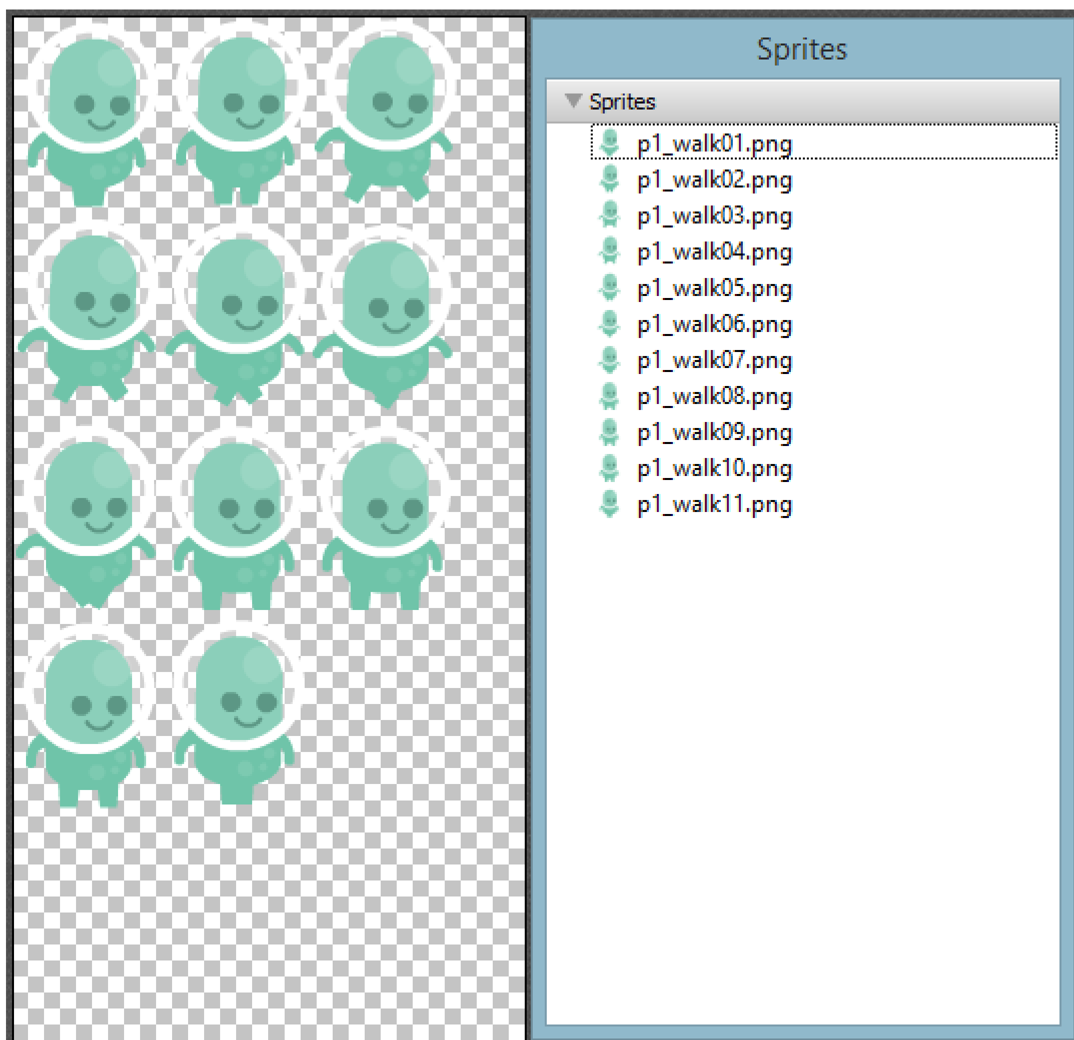

Figura 44. Pre-visualización del atlas de texturas a generar con TexturePacker.

Por último, para generar el atlas de texturas hay que pulsar el botón “Publish”, que se encuentra en la barra de herramientas de TexturePacker.

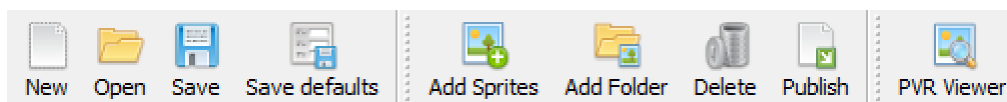

Figura 45. Barra de herramientas de TexturePacker.

Diagrama de clases de DroidEngine2D

C.1. Descripción

En este apéndice se adjunta el diagrama de clases en el que se pueden observar las relaciones entre todos los elementos que componen DroidEngine2D. Este diagrama, junto con la documentación del motor, la documentación de la API, y el propio código fuente del motor, permite al usuario tener una visión completa de los detalles del diseño de DroidEngine2D.

El diagrama se puede obtener a través de la referencia [80].