



Universidad Politécnica de Madrid

Facultad de Informática

Trabajo fin de carrera

Desarrollo de videojuegos para plataforma Android con aceleración hardware

Autor: Adrián Ángeles Ramón

Tutor: Ángel Herranz Nieva

FEBRERO,2014



Resumen

El presente proyecto de fin de carrera, se desarrolla en el departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software, de la Facultad de Informática, de la Universidad Politécnica de Madrid.

El proyecto está estructurado en tres partes. En la primera se explica la lector cuáles han sido las motivaciones para su realización desde un punto de vista técnico, personal y de situación del mercado (*Time to market*).

En la segunda parte se explican los conocimientos técnicos que han sido necesarios para el desarrollo de un videojuego, centrándose en:

- El origen y evolución de la plataforma Android y su arquitectura para desarrollar aplicaciones en smartphones.
- El concepto de computación gráfica, que nos permite generar animaciones en la pantalla utilizando imágenes o renderizando mediante *OpenGL* modelos en tres dimensiones formados por luces, cámaras, mallas y modelos de iluminación.
- Implicaciones de la leyes físicas en las computadoras para poder describir las trayectorias y colisiones de los elementos de un videojuego.

La adquisición de todos los conceptos forma parte del primer objetivo del proyecto y son necesarios para la alcanzar los dos objetivos siguientes:

- Desarrollar una librería para la plataforma *Android*, llamada *TfcGameEngine*, cuya funcionalidad es facilitar el desarrollo de videojuegos, permitiendo leer y renderizar ficheros en formato *Wavefront* mediante el uso de *shaders* en *OpenGL*, aplicar comportamientos a los elementos de un videojuego utilizando leyes de la cinemática y detección de colisiones, creación de animaciones con cámaras, etc. Todo ello aplicando un diseño que permita extender fácilmente su funcionalidad por terceros desarrolladores.

- Implementación del videojuego *Pacmania* de Nanco, conocido en España por *comecocos*, como prueba de concepto de la librería desarrollada.

En la última parte del documento se describe el catálogo de requisitos, tanto de la librería como del videojuego, junto con la explicación de sus diseños ayudándose de diagramas de clases y secuencia en *UML*.



Abstract

The project is structured in three parts. The first part allows the reader to understand the motivations for its realization from a technical, personal and market situation viewpoint.

The second part comprises the technical knowledge that has been needed for the videogame development, focusing on:

- The start out and evolution of the Android platform and the architecture for mobile application development.
- The graphic computing concept, which allows to generate screen animations using images or rendering through a three-dimensional model based on OpenGL consisting of lights, cameras, meshes and illumination models.
- Implications of physical laws in computers in order to describe element trajectories and collisions in a videogame.

The acquirement of all concepts is part of the first aim of the project and are the basis for achieving the following goals:

- To develop a library for Android platform, called TfcGameEngine, whose functionality is to simplify videogame development, allowing read and render Wavefront format files by using OpenGL shaders, application of behaviors to elements of a videogame using laws of kinematics and collision detection, creation of animations with cameras, etc. All this implementing a design that provides an easy way to extend its functionality by third-party developers.
- Implementation of Namco's Pac-Man videogame, known in Spain as *Comecocos*, as *proof of concept* of the library developed.

Finally, the last part of the document outlines the requirements catalogue of the library as well as of the videogame itself, together with an explanation of their designs using UML Class and Sequence diagrams.

*Dedicado a mis padres por su apoyo incondicional.
A mi hermana por sus múltiples paseos hasta mi cuarto.
A Yolanda por formar parte del Pequeño Team.
A Margüenda por dar sentido a la palabra amistad.
A Emilio por la camiseta de SlashMobility que aún espero.
A Ángel Herranz por su paciencia y consejos.*

Índice general

Resumen	III
Summary	v
I Introducción	1
1 Definición del proyecto	3
1.1. Motivación del proyecto	4
1.2. Evolución de los videojuegos	7
1.3. Objetivos del proyecto	13
II Conocimientos previos	15
2 Android	17
2.1. El origen	18
2.2. Arquitectura	19
2.3. Entorno de desarrollo	21
2.4. Estructura de una aplicación	22
2.5. Evolución	24
3 Computación gráfica	33
3.1. ¿Qué es la computación gráfica?	34
3.2. Formatos bidimensionales (2D)	34
3.3. Formatos tridimensionales (3D)	37
3.4. Modelos de iluminación	43
3.5. Librerías de computación gráfica	51
4 Leyes físicas en computadoras	57

4.1. Aplicación de la física en un videojuego	58
4.2. Origen del movimiento	58
4.3. Detección de colisiones	63
III Desarrollo del proyecto	71
5 Requisitos del proyecto	73
5.1. Nomeclatura	74
5.2. Requisitos del videojuego	74
5.3. Requisitos de la librería	79
6 Diseño TfcGameEngine	83
6.1. Visión general	84
6.2. Módulo Scene	85
6.3. Renderer GameEngine	91
6.4. Módulo de colisiones	96
6.5. Módulo de inteligencia artificial	100
6.6. Módulo de renderizado	106
7 Diseño de Pacmania	127
7.1. Visión General	128
7.2. PacmanConfig	129
7.3. Actividad de arranque	142
7.4. Actividad principal	143
7.5. Preferencias	144
7.6. Marcadores	146
7.7. Actividad StarGame	148
7.8. Actividad StartLevel	151
8 Conclusiones y líneas futuras	157
Bibliografía	161
IV Apendices	163
Características del móvil HTC Dream	165
Características del móvil HTC Desire	166

Parte I

Introducción

CAPÍTULO

1

Definición del proyecto

En este primer capítulo se acerca al lector a las motivaciones que han dado forma al proyecto fin de carrera. Para ello se explicará el concepto de los móviles de tipo smartphone, describiendo su situación actual del mercado y las distintas plataformas existentes para el desarrollo de aplicaciones, en concreto para videojuegos.

Un segundo punto a tratar, será la evolución de los videojuegos durante el tránscurso de los años, desde un enfoque de hardware y software, sin olvidar las implicaciones de ser desarrollados en smartphones.

Para finalizar el capítulo se describirán los objetivos que se pretenden alcanzar en la realización del proyecto.



1.1. Motivación del proyecto

Lejos estamos de los años 90 cuando la segunda generación de móviles comenzaba a inundar los mercados. Hoy en día es difícil encontrar a alguien que no tenga uno, es más, se han convertido en un utensilio indispensable en la vida cotidiana de las personas. Los teléfonos móviles han ido evolucionando hasta convertirse en lo hoy conocemos como *smartphone* o *teléfonos inteligentes*.

Esta denominación se debe a una capacidad de computación avanzada, que unida a su conectividad, nos permite trasladar la mayoría de las tareas que realizamos en computadoras personales como son gestionar correos, ver vídeos, escuchar música, jugar a videojuegos... a nuestro ordenador de bolsillo, el smartphone.

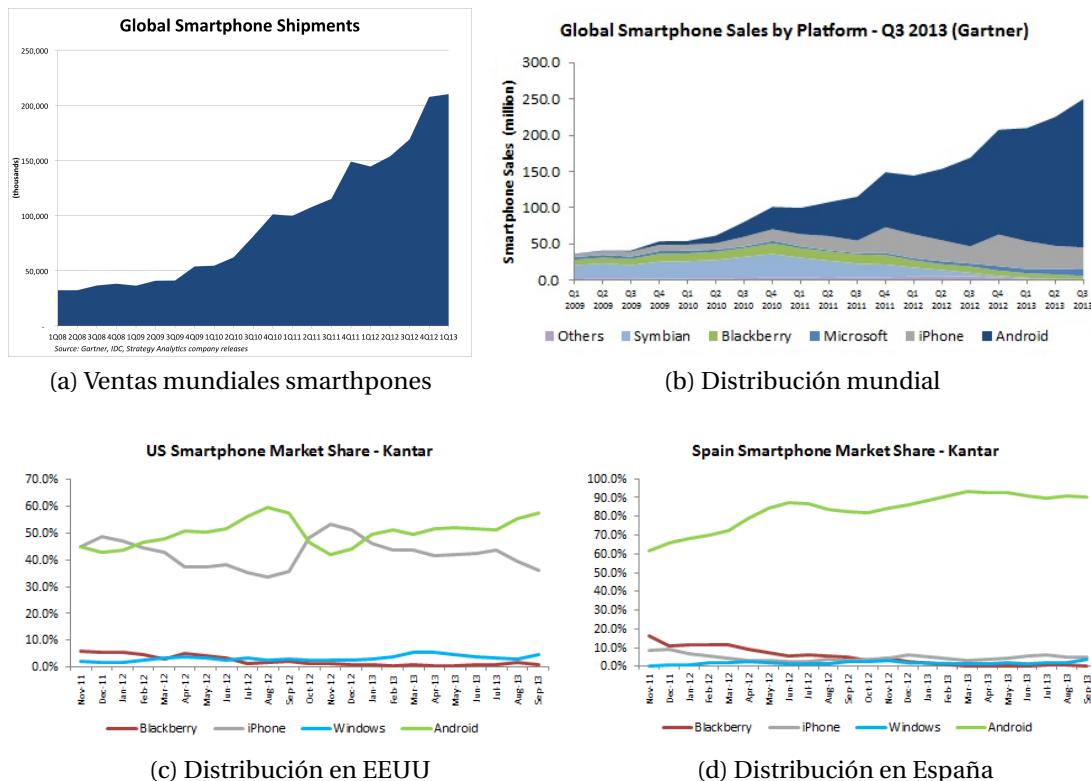


Figura 1.1: Situación actual de los smartphones

Al observar en la gráfica “a” de la figura 1.1, podemos ver que el número de ventas de móviles se incrementan año tras año. Debido a su alto coste, la estrategia de mercado se está centrándose en desarrollar este tipo de dispositivo low-cost, con un menor coste, para que estén al alcance de todos los bolsillos. El objetivo de esta estrategia

1.1. Motivación del proyecto

es doblar la venta de smartphones de bajo presupuesto de forma anual hasta 2016, pasando de 4 millones de unidades en el año 2010 a unas 311 millones 6 años después.

En el resto de los gráficos de la figura 1.1 se muestra la distribución de las principales plataformas ejecutadas en los smartphones a nivel mundial, en Estados Unidos y en España. Cabe destacar que la plataforma Symbian, de Nokia, ha desaparecido casi por completo. A nivel mundial es un mercado dominado por Android. Ocurre lo mismo en España pero en Estados Unidos comparte el mercado junto con iPhone.

Tras realizar un pequeño análisis de las dos plataformas dominantes, se ha llegado a las siguientes conclusiones:

- Tanto Android como iPhone ofrecen un marco de trabajo en una primera fase estable y suficientemente evolucionado para el desarrollo de un proyecto de fin de carrera. El desarrollo sobre iPhone implica unos gastos superiores respecto a Android debido a temas de licencias, coste del dispositivos y software necesario.
- El tiempo requerido en el desarrollo sobre un iPhone es menor que en Android. Esta diferencia radica en que iPhone esta soportado en un pequeño conjunto de dispositivos muy reducido, tan sólo los productos ofrecido por la compañía Apple, mientras que en Android existen una gran variedad de dispositivos con características dispares. Por este motivo, un programa desarrollado en Android ha de ser probado en un mayor número de dispositivos móviles, lo que conlleva un mayor tiempo de pruebas.
- El precio de un dispositivo móvil con un sistema Android y las características de aceleración gráfica 3D por hardware, están dentro del presupuesto personal del proyecto.

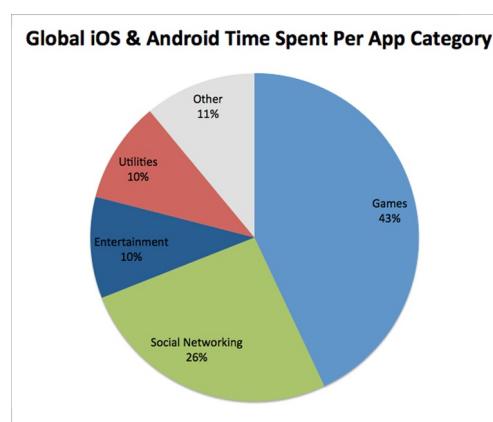


Figura 1.2: ¿Para qué usamos los smartphones?



Capítulo 1. Definición del proyecto

La situación descrita suscita un interés especial en adquirir un mayor conocimiento en el desarrollo de aplicaciones para smartphones Android. En la figura 1.2 se muestran un diagrama con las facetas en las que más tiempo empleamos cuando manejamos un smartphone, lo que nos da una idea de las aplicaciones típicas en un dispositivo de estas características.

Hemos de destacar el gran porcentaje de tiempo que dedicamos jugando con el smartphone, lo que justifica que el desarrollo de videojuegos se está convirtiendo en un nuevo nicho de mercado para las empresas. Actualmente existen varios framework de desarrollo de videojuegos con aspecto gráfico tridimensional para Android, el más famoso es Unity 3D. Sin embargo, esta situación era muy distinta cuando comenzó el proyecto, ya que no existía ningún framework o los que existían era muy precarios.

Debido a la situación del mercado de los smartphone respecto a la creación de videojuegos con gráficos en 3D, surge en forma de proyecto de fin de carrera, la idea de dar respuesta a la pregunta que hizo que estudiara ingeniería informática años atrás, esa pregunta es:

¿Cómo se hace un videojuego?

El juego elegido es Pacmania, una versión que simula las tres dimensiones y que la empresa Nanco desarrolló en 1988 basándose en una versión anterior en 2D llamada Pacman, conocida popularmente como come cocos. Se ha decidido implementar este videojuego ya que no existe ninguna versión parecida en Android, tan sólo una versión en 2D. En los siguientes gráficos podemos ver la versión actual para Android del juego Pacman junto con la versión de PC del juego Pacmania.



(a) Pacmania (PC)

(b) Pacman (Android)

1.2. Evolución de los videojuegos

La evolución de los videojuegos puede ser vista desde dos prismas bien diferenciados. Podemos hablar de cómo se han ido adaptando los formatos con los cuales se presentan al mercado, por ejemplo en máquinas recreativas, consolas... Otra forma de ver esta evolución es centrándonos en las características de los videojuegos como por ejemplo el aspecto visual, conectividad...

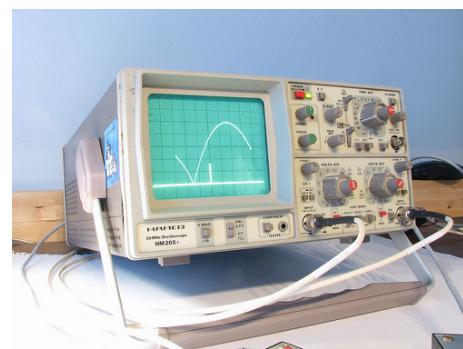
A continuación vamos a detallar la evolución desde ambos puntos de vista, no sin antes mencionar un aspecto a tener en cuenta. Un cambio de formato puede tener implicaciones directas sobre las características del videojuego, por ejemplo, la diferencia de capacidad de procesamiento entre consolas y dispositivos móviles conlleva una perdida de calidad gráfica.

Evolución de los formatos

Actualmente el concepto de videojuego ha sido asimilado por la sociedad, cualquiera sabe qué es un videojuego, incluso han dejado de ser un producto exclusivo para adolescentes. Si nos remontamos a la década de los 40, la palabra videojuego aún no existía. En esta década se crean las primeras computadoras de la historia como Zuse, Colossus y ENIAC... pero fue en 1949 cuando el laboratorio de Matemáticas de la Universidad de Cambridge presenta EDSAC, una computadora que no sólo se utilizó para realizar cálculos matemáticos. En 1952 Alexander S. Douglas construye sobre ella el videojuego Nought and Crosses, también llamado OXO. En España dicho juego es conocido como las tres en raya y permitía enfrentar a un jugador humano contra la máquina, visualizando el tablero sobre una pantalla de tubo de rayos catódicos.



(a) OXO



(b) Tennis for two

Figura 1.3: Los primeros videojuegos caseros que se crearon



Capítulo 1. Definición del proyecto

En 1958, Will Higinbotham en el laboratorio nacional de Brookhaven, simuló en un osciloscopio un partido de tenis. No estaba dotada de ningún tipo de inteligencia por lo que no se podía jugar contra la máquina y se necesitaban dos jugadores. Cada uno de ellos manejaba una especie de mando que tenía una pequeña rueda para indicar el ángulo de golpeo y un botón para golpear la pelota con una raqueta invisible.

En 1961 unos estudiantes del Instituto de Tecnológico de Massachusetts (MIT), Steve Russell entre ellos, construye SpaceWar, un videojuego donde dos naves espaciales y armadas tenían que destruirse mientras evitaban la fuerza gravitatoria de una estrella. Tras pasarse horas jugando a SpaceWar, en 1971 Russell finalmente decidió comercializarlo con el nombre “Computer Space”, lo que dio origen a la primera máquina recreativa.

Sin embargo, la difusión en el mercado de Computer Space fue mucho menor que PONG de Atari, la cual simulaba el juego del ping-pong. Esta segunda máquina recreativa salió al mercado un año más tarde, pero debido a que su difusión fue mucho mayor es considerada como la primera máquina recreativa.

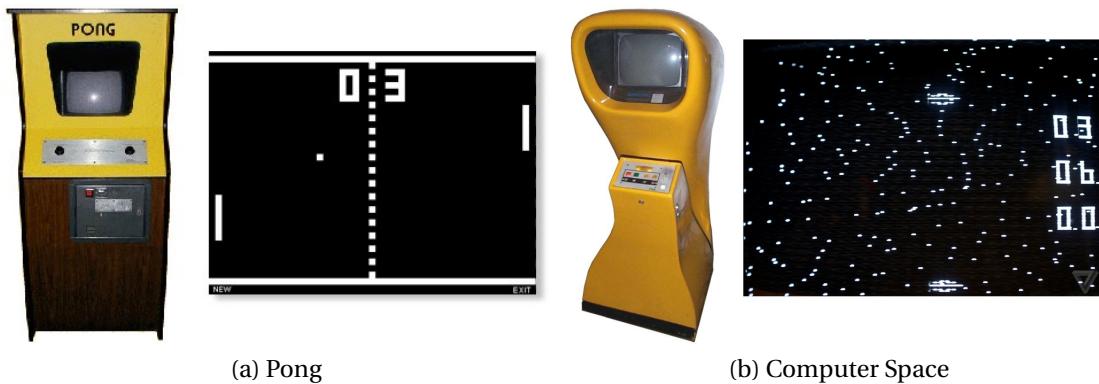


Figura 1.4: Los primeras máquinas recreativas

Pronto surgieron nuevas máquinas recreativas, algunas de ellas son King Kong, Pacman, Tetris, Super Mario Bros... Los adolescentes acudían a un nuevo tipo de negocio, los salones recreativos, donde insertando una moneda podían jugar una partida al videojuego deseado. Este fue el comienzo de la industria de los videojuegos.

Con el trascurso de los años, fueron evolucionando en las temáticas, el aspecto visual y se hicieron más interactivos con la inclusión de nuevos periféricos como volantes, guitarras, cámaras que detectan el movimiento, etc.

1.2. Evolución de los videojuegos

En 1972 la industria de los videojuegos daba sus primeros pasos en los hogares con la creación de la primera generación de consolas. Un gran salto que trasladaba la diversión de los salones recreativos al salón de casa.

A continuación se muestra una tabla con las principales consolas que han salido al mercado y a qué generación pertenecen.



Durante las distintas generaciones de consolas han existido intentos, con mayor o menor acierto, de hacerlas portátiles, ejemplo de ello son la GameBoy y la Nintendo DS, pero han sido los dispositivos móviles los que han logrado llevar los videojuegos a cualquier lugar en el que nos encontremos.

Google, con su plataforma Android, y Apple, con iPhone, son las firmas que lideran esta revolución tecnológica hacia un mundo móvil. Ambas por políticas empresariales bien distintas y exitosas. Google está más interesado en desarrollar sistemas abiertos y Apple genera productos cerrados.

Evolución de las características

¿Qué es lo primero que nos llama la atención de un videojuego? El primer contacto que tenemos con un videojuego es su **aspecto visual** o gráficos. Apreciamos el nivel de detalle con el que son representados los paisajes, personajes y objetos. Inicialmente los gráficos eran minimalistas debido a la falta de procesamiento de los soportes, lo que conllevaba representar todo mediante cuadrados de escasos colores. Por este motivo podemos decir que existe una relación directamente proporcional entre capacidad de procesamiento y el aspecto visual. Al aumentar el número de cálculos matemáticos que se pueden realizar por segundo, podemos mejorar el nivel de detalle de lo representado en pantalla.

El cambio más significativo ha sido la capacidad de poder reproducir modelos tridimensionales, dando la posibilidad de crear efectos visuales que simulan el movimiento de una cámara en un escenario.



Figura 1.5: Evolución gráfica en los videojuegos de fútbol

Ahora que el videojuego ha captado nuestra atención gráficamente, prestamos atención a la trama que se va entrelazando según avanzamos en él. En los primeros videojuegos la **historia** desarrollada era nimia pero, poco a poco, éste factor ha ganado importancia, hasta el punto en el cual podemos decir que los costes de desarrollo de un videojuego son equiparables al de una película.

Existe una característica que pasa desapercibida por el jugador al interiorizarla como algo natural. Estamos hablando de las **propiedades físicas** del escenario en el cual nos movemos. El jugador no se para a pensar porqué el personaje que está manejando puede saltar, puede coger otros objetos pero no puede atravesar las paredes. Sin embargo, cuando la física del videojuego no funciona a la perfección, como por ejemplo cuando se atraviesan ligeramente ciertos objetos, no ve el agua de un río fluir o las llamas del fuego no se mueven, son captadas por el usuario como fallos de importancia. Al igual que el aspecto gráfico de un videojuego, la física esta relacionada con la capacidad de procesamiento. Hoy en día podemos ver juegos que aplican algoritmos para calcular cómo han de moverse los elementos sólidos y fluidos, dando un gran realismo a los juegos.

1.2. Evolución de los videojuegos

En el transcurso del videojuego podemos encontrarnos frente a otros personajes, que se relacionan en el escenario, como si fueran manejados por terceras personas. Es la **inteligencia artificial** la responsable de este comportamiento, que maneja personajes dotándoles de un comportamiento natural, como por ejemplo que un perro persiga y ladre a un extraño o que la gente de tu alrededor te mire si rompes algo.

Es sorprendente la evolución de la inteligencia artificial, al principio los movimientos de los personajes podrían catalogarse de burdos y hoy en día nos hace dudar si tras un personaje se encuentra una persona o no.

Una de las preguntas habituales que todos nos hacemos de forma natural es cómo compartir las experiencias. Existen videojuegos en los cuales no es posible, en otros puede haber varios jugadores simultáneos. Al principio para que se pudiera dar esta situación, había que estar en la misma sala, sin embargo esta limitación desapareció debido a la **conectividad** de los soportes a través de Internet. Esta característica ha llevado a los videojuegos a un nuevo nivel convirtiéndolos en un acto social.

Videojuegos en dispositivos móviles

Una vez descrita la evolución de los videojuegos, tanto a niveles de formatos como características, se van a describir las implicaciones en el ámbito de los smartphone.

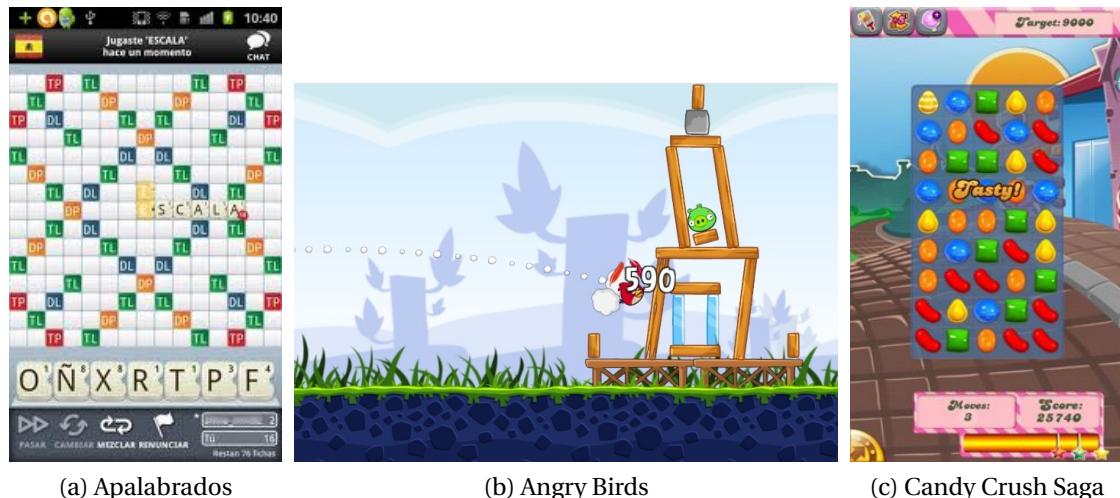


Figura 1.6: Top éxitos en Android

A pesar de la gran evolución de los móviles, su capacidad de procesamiento no es, ni de lejos, comparable con las computadoras de sobremesa ni consolas. A esta situación se le suma un mercado emergente y que no dispone de frameworks especializados



Capítulo 1. Definición del proyecto

para el desarrollo de videojuegos, lo que ha provocado una involución en casi todas sus características. Este hecho puede constatarse al consultar los videojuegos más descargados para Android. Predominan los juegos sin una historia desarrollada y con un aspecto visual simplista. Algunos de ellos son Apalabradados, Candy Crush Saga y Angry Bird, de los cuales se muestran capturas de pantalla en la figura 1.6.

Un ejemplo de un videojuego que aprovecha el potencial de los smartphones actuales es el Grand Theft Auto, a pesar de ello, existe una gran diferencia en su aspecto gráfico en comparación con la versión de consola, tal y como muestran las siguientes capturas de pantallas.



(a) Android/iPhone



(b) Play Station 3

Figura 1.7: Grand Theft Auto

Una dificultad añadida consiste en la desaparición del teclado y mandos con los que controlar el videojuego, siendo sustituida en la mayoría de los casos por la pantalla táctil o los sensores de movimiento del dispositivo.

Hay que destacar que un smartphone ofrece una serie de características adicionales propias de ellos, las más destacables son el GPS y la cámara. De estas cualidades han surgido dos conceptos nuevos:

- La **movilidad** permite a las aplicaciones mejorar su funcionamiento al poder incluir la posición en la que nos encontramos. Por ejemplo, nos permite saber cuál es la estación de metro más cercana o hacer campañas publicitarias por proximidad.
- La **realidad aumentada** permite mezclar el mundo real y el digital a través del uso de la cámara. Por ejemplo, podríamos ver como nos queda un mueble en un salón a través de la cámara.

Estas particularidades están estimulando la creación de otros tipos de videojuegos, como por ejemplo Ingress, de Google. Este videojuego consiste en salir a la calle con el

1.3. Objetivos del proyecto



Figura 1.8: Ejemplo de realidad aumentada

móvil, abrir la aplicación y buscar *fuentes de esta misteriosa energía* ocultas en nuestra ciudad, transformando el mundo real en el escenario del videojuego mediante el uso de la cámara, las redes de telefonía y el GPS.

1.3. Objetivos del proyecto

Una vez explicadas las motivaciones que han dado lugar al proyecto de fin de carrera, la evolución de los videojuegos y la implicaciones que existentes en smartphone, se van a definir los objetivos del proyecto.

1. Adquirir los conocimientos específicos necesarios para poder desarrollar videojuegos con modelos en 3D.
2. Aprender a realizar diseños gráficos con Blender para generar el aspecto gráfico de los elementos de un videojuego.
3. Aprender el lenguaje de programación específico para tarjetas gráficas llamado GLSL (OpenGL Shading Language).
4. Conseguir los conocimientos del desarrollo de aplicaciones en Android.
5. Comprar un smartphone adecuado para el desarrollo del proyecto que cuente con el hardware que permita aceleración gráfica mediante OpenGL ES 2.0 y la versión Gingerbread de Android. Estos conceptos son desarrollos en la primera parte del proyecto. Tras realizar un estudio de mercado de los smartphone que cubrían esta características, el móvil elegido fue el HTC Desire.
6. Desarrollar una librería genérica que facilite el desarrollo de videojuegos para Android. La llamaremos TfcGameEngine y contendrá diversas funcionalidades:
 - Bucle principal del videojuego en el cual se van secuenciando imágenes como si de una película se tratase.



Capítulo 1. Definición del proyecto

- Soporte para OpenGL ES 2.0 (Este concepto que se explicará en el capítulo 3 del proyecto).
- Utilidades para el cálculo de movimiento en función del tiempo, velocidad y aceleración, es decir, cinemática.
- Detección de colisiones entre elementos del juego.
- Algoritmos que permitan desplazar los personajes de videojuego sobre la escena de forma manual, aleatoria o entre dos puntos.
- Procesamiento de los diseños gráficos realizados con Blender.
- Motor de música que permita reproducir melodías y efectos de sonido.

Un principio básico que ha primado en el diseño de la librería es que tiene que ser fácilmente extensible por terceros, permitiendo de esta forma, añadir nuevas funcionalidades.

7. Realizar una prueba de concepto de dicha librería programando un videojuego de un comecocos en 3D, llamado Pacmania. Este juego consiste en controlar al Pacman, un personaje que ha de recoger las pastillas de un escenario custodiado por fantasmas. Para ello será necesario:

- Realizar los diseños gráficos de los escenarios y personajes del videojuego, que son dos escenarios, un fantasma y un pacman.
- Realizar un proceso de carga de los ficheros contenidos en un directorio de configuración. Este directorio contiene los escenarios, personajes, efectos especiales, melodías, vídeos, etc.
- Ampliación de la librería TfcGameEngine para incluir la lógica específica del videojuego como por ejemplo, controlar la puntuación y número de vidas de la partida.
- Creación de animaciones con la cámara cuándo comienza un nivel o el Pacman es capturado por un fantasma.
- Reproducción de efectos especiales y melodías.
- Pantalla de configuración que permita indicar si el sonido esta activo y donde se encuentra el directorio de configuración.
- Registro de las mejores puntuaciones en una base de datos SQLite.

8. Testar el videojuego en otros smartphone como Galaxy S2 y Nexus 5.

Para mas información sobre la librería TfcGameEngine y el juego Pacmania, consultar los catálogo de requisitos del capítulo 5.

Parte II

Conocimientos previos

CAPÍTULO

2

Android

Una vez enmarcado el proyecto sobre la plataforma Android, en este capítulo se pretende dar una vista más específica de las funcionalidades que nos ofrece en cada una de sus versiones. Se explicará cómo surgió esta plataforma y los conceptos básicos de cómo están compuestas sus aplicaciones y el entorno de desarrollo necesario para crearlas y distribuirlas.



2.1. El origen

El germen de Android nace en el proyecto Paradigm, de la empresa General Magic, en el año 1990. Esta empresa surgió de Apple y su objetivo fue crear un sistema operativo para teléfonos móviles al que llamarían Magic Cup. El proyecto fracasó pero uno de sus arquitectos continuó su carrera profesional en Artemis Research, que posteriormente comprada por Microsoft. Más tarde fundó Danger Inc, también comprada por Microsoft y que abandonó para formar una nueva empresa llamada Android Inc, que fue vendida a Google en el año 2005. El arquitecto del que estamos hablando es Andy Rubin, actual vicepresidente de Google.

En 5 de Noviembre del 2007, se funda la Open Handset Alliance, dirigida por Google junto con unos 34 miembros más, entre los que se incluían fabricantes de dispositivos móviles, desarrolladores de aplicaciones, algunos operadores de comunicaciones, fabricantes de chips... anunciando la creación de una plataforma de código libre para teléfonos móviles.

El 23 de septiembre de 2008, nace la primera versión de lo que hoy se conoce como Android y un mes más tarde sale al mercado HTC Dream (T-Mobile G1) el primer móvil con Android.



Figura 2.1: HTC Dream

2.2. Arquitectura

Android es una plataforma para smartphones, pero ha evolucionado hacia otro tipo de dispositivos móviles como las tabletas. En el siguiente gráfico se muestra la estructura general de esta plataforma:



Figura 2.2: Arquitectura Android

Un sistema operativo Linux, responsable de gestionar los recursos ofrecidos por el hardware como memoria, batería, pantalla... Cada fabricante ofrece distintos drivers para comunicarse con su hardware y es el sistema operativo quien nos independiza de tener que conocer cada una de ellas, ofreciéndonos una interfaz común. También es el responsable de gestionar los distintos recursos que ofrece el dispositivo como son:

- Pantalla
- Teclado
- Cámara de fotos
- Wifi
- Memorias flash
- Audio
- Batería
- etc



Librerías en C y C++ que permiten acceder al hardware del dispositivo móviles u otros componentes que estén integrados. Las librerías principales se listan a continuación:

- Surface Manager: gestiona los gráficos mostrados en pantalla.
- Media Framework: permite reproducir imágenes, audio y vídeo.
- SQLite: gestiona el acceso a bases de datos relacionales.
- WebKit: navegador web optimizado.
- SGL: permite realizar gráficos en 2D.
- OpenGL ES: permite realizar gráficos en 3D.
- FreeType: procesamiento de imágenes vectoriales.

Entorno de ejecución, con la máquina virtual basada en Java llamada **Dalvik**, que ha sido modificada para optimizar entornos con restricciones de procesamiento, memoria y espacio, como son los dispositivos móviles.

Dalvik permite independizar las aplicaciones del hardware mediante ficheros con un formato con extensión **dex**, que significa Dalvik executable.

El programador utilizará la sintaxis Java y dispondrá de un amplio conjunto de JME¹, optimizado para Dalvik. Al compilar el código se generarán bytecode de java, que posteriormente es optimizado mediante la herramienta dx, para que optimice el código y genere el fichero dex.

La comunicación del entorno de ejecución con las librerías en C y C++ se realiza mediante JNI².

Framework de aplicaciones, ofrece la funcionalidad básica de Android, siendo las más destacables:

- Activity Manager: gestiona las actividades que se están ejecutando en el smartphone. Toda aplicación en Android está compuesta por actividades y se define como un componente gráfico con su propio ciclo de vida. En la siguiente sección veremos en detalle la composición las aplicaciones en Android.
- Windows Manager: gestiona qué es lo que se visualiza por pantalla. Cuando una actividad necesita ser visualizada por el usuario, ha de invocar a este componente.
- Contents Provider: gestiona grupos de información que son utilizados por varias aplicaciones. Los principales contenidos de Android son la agenda de contactos y el calendario.

¹Java Micro Edition: Librería Java para el desarrollo de aplicaciones móviles

²Java native interface: Interfaz nativo de java

- Telephony Manager: permite acceder a las funcionalidades de telefonía del dispositivo, protocolos de comunicación permitidos, los datos de la SIM, estado de la conexión...
- Resource Manager: permite acceder al contenido estático de la aplicación, como ficheros de audio, XML de configuración...
- Location Manager: permite acceder a la información de localización del dispositivo por GSP, WIFI y las torretas de telefonía.
- Notification Manager: permite acceder al componente de notificaciones de Android, un panel donde se registran avisos de distinta índole, por ejemplo, nuevos mensajes, llamadas perdidas, actualizaciones...

Otras aplicaciones son instaladas en el dispositivo por las compañías u los usuarios de los dispositivos móviles. La forma más común en la que estas aplicaciones pueden ser instaladas es desde la web de Google. Inicialmente surgió con el nombre de Google Market y posteriormente fue renombrado como Google Play.

<https://play.google.com>

2.3. Entorno de desarrollo

El siguiente paso, tras haber tomado conciencia sobre qué es Android, es explicar las herramientas principales que nos ofrece la plataforma para facilitar el desarrollo de aplicaciones:

Eclipse, un IDE³ desarrollado por la comunidad OpenSource, que se caracteriza por permitir ser configurado por terceros.

Android Developer Tools (ADT), plugin que configura Eclipse para poder desarrollar aplicaciones en Android.

Android Debug Bridge (ADB) es una herramienta que permite conectarse vía USB con dispositivos Android y realizar un conjunto de acciones sobre él. Las más utilizadas son instalar aplicaciones, mover ficheros o crear una sesión mediante el protocolo de telnet, accediendo directamente con el sistema operativo Unix del dispositivo móvil.

Un emulador que cubre gran parte de las características soportadas en dispositivos móviles con Android. Permite ejecutar aplicaciones sin necesidad de tener un dispositivo físico.

³Integrated development Environment: aplicación compuesta por un conjunto de herramientas para el desarrollo de aplicaciones



Android Virutal Device (ADV), ficheros que definen la configuración hardware de un dispositivo móvil y son cargados en el emulador. Son generados mediante una herramienta llamada Android, integrada dentro de su entorno de desarrollo.

Mksdcard, herramienta que permite simular una tarjeta de memoria virtual sobre un fichero, la cual podrá cargarse dentro del emulador.

Logcat, herramienta que nos permite ver las trazas de información que van dejando todas las aplicaciones que se ejecutan en el dispositivo móvil.

Otras herramientas que permiten diversas funcionalidades para generar copias de seguridad, analizadores y ofuscadores de código...

2.4. Estructura de una aplicación

Una vez que tenemos montado nuestro entorno de desarrollo con las herramientas mencionadas en el apartado anterior, se explicará la estructura de una aplicación en Android. Al crear un proyecto, se generará la siguiente estructura de ficheros:

- **AndroidManifest.xml**, fichero principal de la aplicación donde se indican las características principales de la aplicación. Es obligatorio definir cuales son las actividades que componen la aplicación, cuál de ellas es la principal, la versión de la aplicación y versión requerida del SDK. Además se han de definir cuáles son los permisos necesarios para el correcto funcionamiento de la aplicación, por ejemplo envío automático de mensajes, acceso al GPS, contactos, cámara, etc. Al instalar la aplicación, se indicará al usuario los permisos que requiere y ha de aceptarlos si desea instalarla.
- Directorio **src** donde se ubican las fuentes java. El proyecto creado con el plugin ADT, genera una clase llamada como el proyecto que extiende de Activity y será el punto inicial que se ejecutará en la aplicación.
- Directorio **res** donde se ubican distintos recursos en una jerarquía de directorios en función de las características de los mismos:
 - drawable-hdpi: contiene las imágenes de alta densidad.
 - drawable-mdpi: contiene las imágenes de media densidad
 - drawable-ldpi: contiene las imágenes de baja densidad.
 - layout: contiene ficheros xml con la definición de las interfaces gráficas que se cargarán en la actividad.
 - values: contiene ficheros xml donde se definen propiedades utilizadas por la aplicación como colores, cadenas de texto...

2.4. Estructura de una aplicación

Una aplicación Android puede estar compuesta por varias actividades, consideradas como unidades atómicas, buscando en ellas una alta cohesión y bajo acoplamiento.

Debido a las limitaciones del dispositivo móvil, las actividades son gestionadas por Android atendiendo a su estado y gestionarlas de la forma más eficiente posible.

Para comprender mejor el concepto de actividad, recurriremos al ejemplo principal de la página de desarrolladores de Android⁴. El ejemplo *Hello World* está compuesto por dos actividades, en la primera de ellas se solicita una cadena de texto y en la segunda se muestra la cadena con una letra mayor. De forma coloquial, podemos entender las actividades como las distintas pantallas que componen una aplicación.

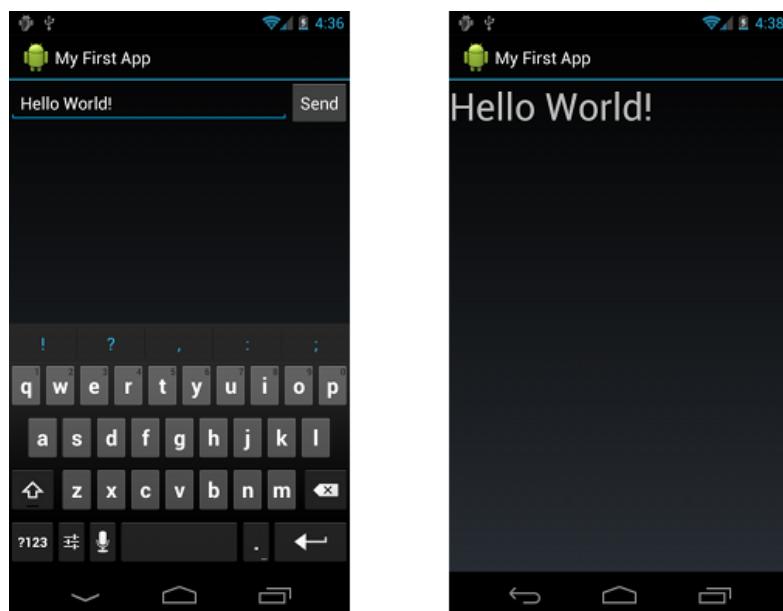


Figura 2.3: Actividades del ejemplo Hello World

Toda actividad sigue un ciclo de vida, se van produciendo eventos que provocan una transición a un nuevo estado. Los eventos principales de este ciclo de vida son los siguientes:

- **onCreate:** Ocurre cuando la aplicación se crea. En este punto se ha de crear la interfaz gráfica e inicializar las estructuras de datos pertinentes.
- **onStart:** Ocurre cuando la actividad se va a mostrar en pantalla por primera vez.
- **onRestart:** Ocurre cuando la actividad parada va a mostrarse de nuevo.

.....
⁴<http://developer.android.com>



- **onResume:** La actividad comienza a visualizarse en pantalla, interactuando con el usuario.
- **onPause:** La aplicación se ha pausado. Y deja de interaccionar con el usuario. En ese momento, todos los datos serán almacenados en la memoria secundaria, ya que el sistema puede terminar esta actividad pausada si se ve baja de recursos. Si se reactivara, pasaría al estado onResume.
- **onStop:** Cuando una segunda actividad, nueva o reactivada, ha de mostrarse en pantalla. La actividad mostrada en esos momentos recibe este evento, pasando a un segundo plano, un estado candidato a ser destruido en caso de falta de memoria. Si la actividad pasase de nuevo al primer plano, cambiaría a estado onRestart.
- **onDestroy:** La aplicación acaba y se destruye, bien porque acaba ella misma o el sistema la termina.

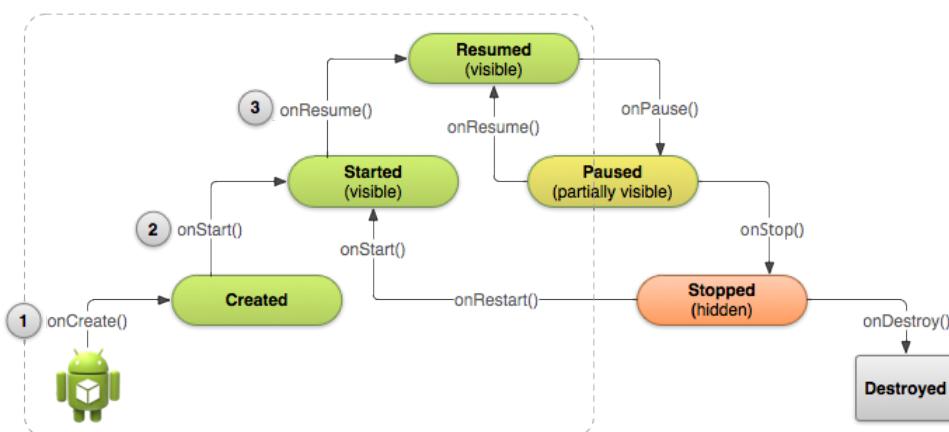


Figura 2.4: Ciclo de vida de una actividad

2.5. Evolución

Actualmente existen varias versiones de Android que van incorporando nuevas funcionalidades. La primera versión, llamada **Apple Pie** (Pastel de manzana), sólo se incorporó al modelo HTC-Dream⁵.

Su característica más importante consistía en poder sincronizar los datos del móvil y la cuenta de Google. Otras características a destacar son:

.....
⁵Para más información sobre las especificaciones hardware consulte los apéndices

- HSDPA/850/900/1800/1900.
- GPRS.
- SMS y MMS.
- Cámara de fotos.
- Wifi (Cifrado WEP).
- Bluetooth
- USB 1.0
- Navegador HTML
- Aplicación YouTube
- Aplicación música
- Aplicación Android Market⁶
- Aplicación de bloqueo táctil
- Aplicación GMail
- Aplicación Google Calendar
- Aplicación Google Contacts
- Aplicación Google Talk⁷
- Aplicación Google Maps



Figura 2.5: Logotipos de las versiones de Android

Cupcake

El 30 de abril del 2009 se presenta la versión 1.5 de Android, conocida como Cupcake. Esta versión se caracteriza por:

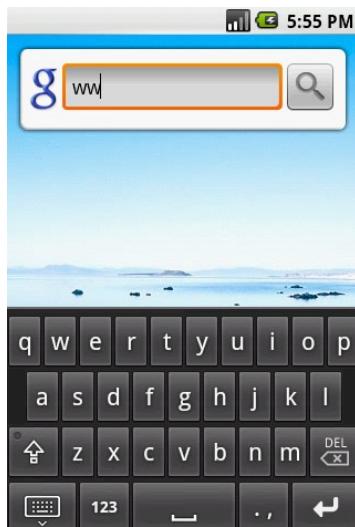
- Estar basada en el kernel Linux 2.6.27.

.....
⁶Centro de descarga de aplicaciones para el móvil

⁷Aplicación de intercambio de mensajes de texto



- Tener un teclado virtual, innecesario en la versión anterior ya que el HTC Dream tenía un teclado físico.
- Se mejora la velocidad de la cámara de fotos y se permite grabar vídeos.
- Permite subir video y fotos a YouTube y Picassa.
- Se mejorar el soporte bluetooth estéreo.
- Se permite insertar nuevos widget desarrollados por terceros.
- Transiciones de pantallas animadas.
- Mejoras de tiempo de respuesta en el acceso GPS al utilizar A-GPS.
- Mejorar el rendimiento en WebKit con el interprete de javascript *SquirrelFish*.
- Posibilidad de cortar y pegar texto seleccionado.
- Gestor de aplicaciones instaladas.



(a) Teclado virtual

Manage applications		
	Alarm Clock	64.00KB
	Android System	0.00B
	android.core	360KB
	Browser	1.56MB
	Calculator	84.00KB
	Calendar	252KB
	Running	Third-party
	Sort by size	200KB

(b) Gestor de aplicaciones

Figura 2.6: Novedades de la versión Cupcake

Donut

El 15 de Septiembre de 2009 se presenta la versión 1.6, conocida como Donut. Esta versión se caracteriza por:

- Soporte para telefonía de datos mediante CDMA.
- Creación de redes virtuales mediante VPN.
- Cifrado WAP en conexiones WiFi.
- Motor de texto por voz que permite traducir un texto y escucharlo.
- Nuevo applet de búsqueda que consulta tanto en Internet cómo en los datos del dispositivo.
- Gestures que permite realizar acciones en el móvil al realizar una trazada con el dedo sobre la pantalla.
- Soporte para pantallas de distintos tamaños y densidades.
- Aplicación de gestión de batería por aplicación.

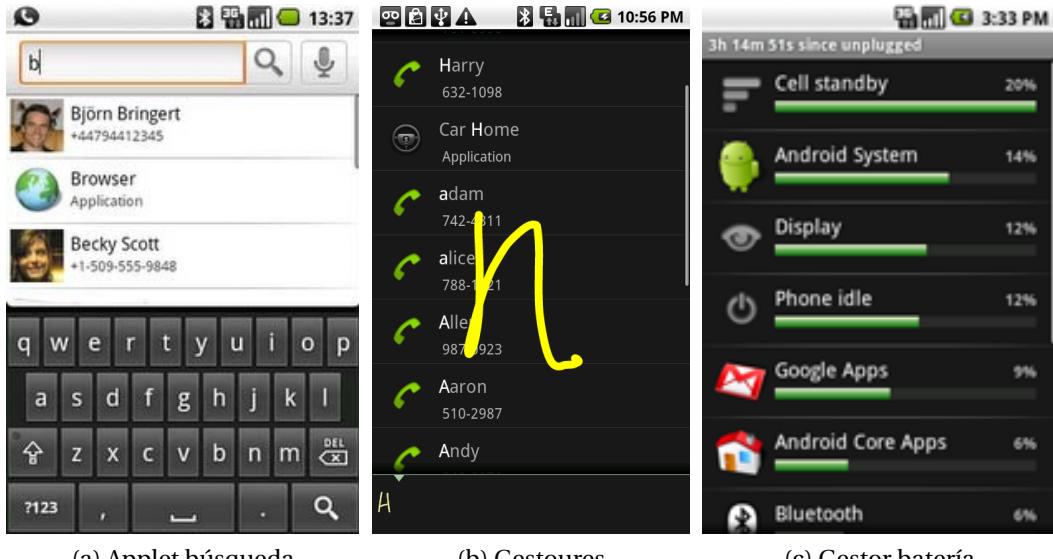


Figura 2.7: Novedades de la versión Donut



Eclair

El 26 de octubre del 2009 se presenta la versión 2.0 de Android, conocida como Eclair. Esta versión se caracteriza por:

- Sincronización con Exchange.
- Sincronización con múltiples cuentas.
- Soporte HTML 5.
- Novedades en el uso de cámara que incluye flash, zoom digital...

Froyo

El 20 de mayo del 2010 se presenta la versión 2.2 de Android, conocida como Froyo. Esta versión se caracteriza por:

- Optimizaciones en velocidad, memoria y rendimiento.
- Soporte Flash.
- Instalación de aplicaciones en la SD en el almacenamiento interno de algunos dispositivos móviles.
- Anclaje por USB, permitiendo al dispositivo al que se conecte el móvil usar la red de datos.
- Soporte C2DM (Cloud to device Messaging) que permite sincronizar datos de una forma más efectiva. En vez de pooling es el servidor quien notifica al cliente que los datos han sido modificados.
- Soporte OpenGL ES 2.0 para generación de gráficos tridimensionales.

Gingerbread

El 6 de diciembre del 2010 se presenta la versión 2.3 de Android, conocida como Gingerbread. Esta versión se caracteriza por:

- Aplicación Google Wallet para realizar pagos mediante móvil.
- Soporte NFC (Near Field Communication). Esta tecnología permite comunicar de forma inalámbrica de corto alcance y alta frecuencia entre dispositivos para el intercambio de datos.
- Soporte VoIP para realizar telefonía IP (Llamadas por Internet).

- Mejora de rendimiento en el recolector de basura de Davilk.
- Soporte nativos para sensores como el giroscopio y barómetro.



Figura 2.8: Google Wallet

Honeycomb

El 22 de febrero del 2011 se presenta la versión 3.0, conocida como Honeycomb. Esta versión consiste en una adaptación de la interfaz gráfica para tabletas. También se incluye un mayor soporte USB para conectar dispositivos a la tableta.



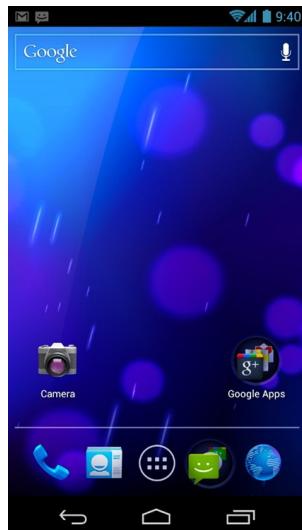
Figura 2.9: Nueva interfaz gráfica



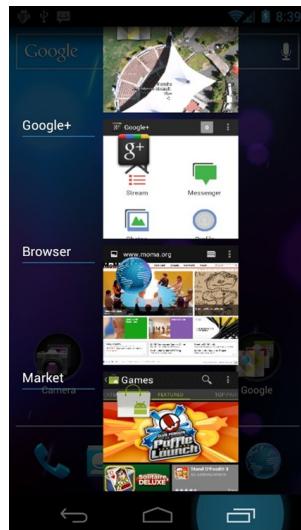
Ice Cream Sandwich

El 19 de octubre del 2011 se presenta la versión 4.0 de Android, conocida como Ice Cream Sandwich. Esta versión unifica las dos anteriores, siendo tanto para móviles como para tabletas. Esta versión se caracteriza por:

- Se basa en el kernel 3.0.1 de Linux.
- Soporte WiFi direct, que permite conectar dispositivos por WiFi sin necesidad de puntos de acceso.
- Grabación de vídeos en FullHD.
- Interfaz gráfica de usuario acelerada mediante hardware.
- Aplicación Android Beam para intercambio de ficheros mediante NFC.
- Desbloqueo facial.
- Soporte para fotos panorámicas.
- Barra multitarea que permite ver todas las aplicaciones activas.



(a) Nueva interfaz



(b) Teclado deslizante

Figura 2.10: Novedades de la versión Ice Cream Sandwich

Jelly Bean

El 9 de julio del 2012 se presenta la versión 4.1 de Android, conocida como Jelly Bean. Esta versión se caracteriza por:

- Mejora la interfaz, integrando el proyecto Butter, ofreciendo una sensación de fluidez mayor al manejar el dispositivo.
- Agrega nuevas funcionalidades a la aplicación de la cámara. La posibilidad de crear imágenes panorámicas en 360 grados es la más destacable.
- Soporte multi-usuario.
- Aplicación de inteligencia artificial, Google Now, la cual responderá a algunas preguntas típicas relacionadas con el tiempo, eventos, vuelos, lugares, tráfico, resultados deportivos...
- Teclado predictivo deslizante.



Figura 2.11: Novedades de la versión Jelly Bean

Kit Kat

Junto con el nuevo móvil de Google, el Nexus 5, el 31 de octubre de 2013 se presenta la versión 4.4 de Android, conocida como Kit Kat. El objetivo principal de esta versión es resolver el problema de fragmentación de la plataforma, en la cual existe muchas versiones de Android. Para cubrir esta meta, se ha optimizado el uso de batería y la RAM, pudiendo ejecutarse en la mayoría de los móviles actuales.

CAPÍTULO

3

Computación gráfica

El objetivo de este capítulo es explicar los conceptos técnicos necesarios para poder comprender cómo se crean y representan animaciones en la pantalla del smartphone o cualquier otro dispositivo.

Comenzaremos explicando cuáles son los datos necesarios para definir animaciones y las operaciones necesarias para poder representarlas en la pantalla, teniendo en cuenta propiedades de la luz como son la reflexión, refracción y absorción.

Para finalizar, hablaremos de OpenGL, una librería para el tratamiento de gráficos en tres dimensiones, que será utilizada en el proyecto.



3.1. ¿Qué es la computación gráfica?

La computación gráfica es el arte de producir imágenes con una computadora, proyectándolas sobre una superficie digital, como un televisor, un móvil o de un cine.

La información necesaria, procesada por un algoritmo, para generar cada imagen es almacenada en un formato gráfico. Una clasificación que podemos realizar respecto a los formatos es si representan la información en dos o tres dimensiones, es decir, formatos bidimensionales o tridimensionales.

La rasterización consiste en el algoritmo que procesa los datos almacenados en un formato gráfico, para generar una imagen, atendiendo al tamaño y resolución correspondiente a la superficie de pantalla donde se va a visualizar.

No hemos de confundir formatos tridimensionales con las pantallas 3D. El proceso de rasterización para pantallas 3D crea imágenes manipuladas de tal forma que nuestro cerebro las procesa como tridimensionales directamente o a través de algún tipo de periférico como gafas polarizadas.

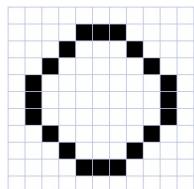
3.2. Formatos bidimensionales (2D)

Cuando hablamos de formatos bidimensionales, nos referimos directamente a las imágenes. A su vez, este formato puede ser representado de dos formas:

- **Mapa de bit** (jpg y bmp): La imagen es almacenada como una cuadricula, en la que se indica qué color tiene cada celda. Lo más común es representar los datos usando métodos de compresión que reducen significativamente su tamaño. La resolución¹ de la imagen está directamente relacionada con el número de celdas capaz de representar.
- **Vectoriales** (ps y svg): La imagen es guardada utilizando fórmulas matemáticas básicas como son círculos, cuadrados, elipses... por lo que no está definida su resolución. Su tamaño viene determinado por la complejidad de la imagen y no por su resolución.

Existen formatos tan comunes como pdf que son híbridos, es decir, algunas regiones del documento son representadas de forma vectorial y otras como mapas de bit.

.....
¹Tamaño de la imagen expresado en pixeles de alto por ancho



(a) Mapa de bit

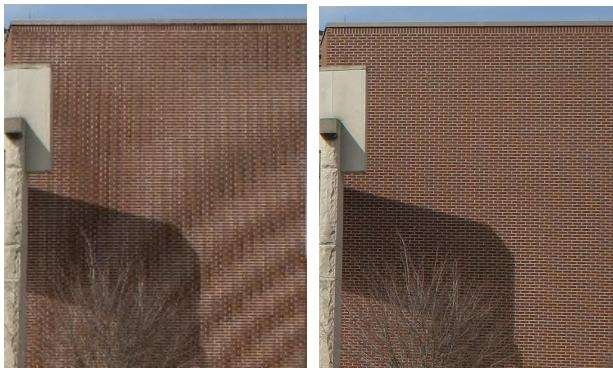
```
<?xml version="1.0" standalone="no"?>
<svg width="200" height="250" version="1.1" >
  <circle cx="25" cy="75" r="20"/>
</svg>
```

(b) Vectorial

Figura 3.1: Representación de un círculo

Si nos centramos en el proceso de rasterización, los formatos en mapa de bits son mucho más ligeros que los vectoriales, esto se debe a que existe una relación casi directa entre las celdas del mapa de bit y los píxeles de pantalla.

Sin embargo, como la resolución de una mapa de bit está implícito en el formato, en algunos casos pueden provocarse errores en la rasterización conocidos como aliasing. Estos errores se producen cuando se adapta a la resolución de la pantalla donde se va a visualizar. Para corregir esta problemática, existen distintos tipos de filtros conocidos como **antialiasing**, los cuales son capaces de detectar aquellos píxeles conflictivos y cambiar su color a uno más apropiado.



(a) Aliasing I

(b) Anti-Aliasing

Figura 3.2: Efectos del aliasing y antialiasing sobre una pared de ladrillos

La computación gráfica también abarca las animaciones o secuencias de imágenes. Como hemos visto, existe una relación directa entre un formato bidimensional y una imagen, por este motivo, para crear animaciones necesitaremos tener secuencias de formatos bidimensionales.



Figura 3.3: Animación de una persona corriendo

Los videojuegos intentan crear animaciones cercanas a la realidad, proyectando los elementos en la pantalla de la forma más realista posible. Lo que implica tener en cuenta desde dónde se proyecta la imagen que estamos capturando y la iluminación existente. La única forma de poder recrear esta situación con formatos en 2D sería tener un numero infinito de imágenes, por este motivo se usan los formatos en 3D.



(a) Pacman (1980)

(b) Super Mario Bros (1985)



(c) Prince of persia (1989)

Figura 3.4: Evolución de videojuegos con formato 2D

3.3. Formatos tridimensionales (3D)

Los complejidad de los formatos 3D es muy superior comparada con los formatos 2D. Para representar una escena se ha de indicar cada uno de los elementos existentes, la iluminación y qué parte de esa escena va a ser proyectada en la pantalla. Al pensar en una escena cualquiera, por ejemplo una casa, los elementos que lo componen serían las distintas habitaciones, armarios... La iluminación variará en función de las luces que estén encendidas y la luz que entre por las ventanas. La proyección de la escena sobre la pantalla no sólo dependerá de los elementos y la iluminación, también será influida por la posición en el que se sitúe el espectador. No vemos lo mismo si estamos sentados en el sillón o tumbados en la cama.

Mallas

Cada elemento de una escena es representado mediante polígonos. El conjunto de los polígonos que componen un elemento se conoce como **malla** o en inglés, **mesh**. En computación gráfica estos polígonos se reducen a cuadriláteros o triángulos². Cada polígono, conocido por cara o **face**, está formado a su vez por tres o cuatro vértices para formar un triángulo o un cuadrado respectivamente.

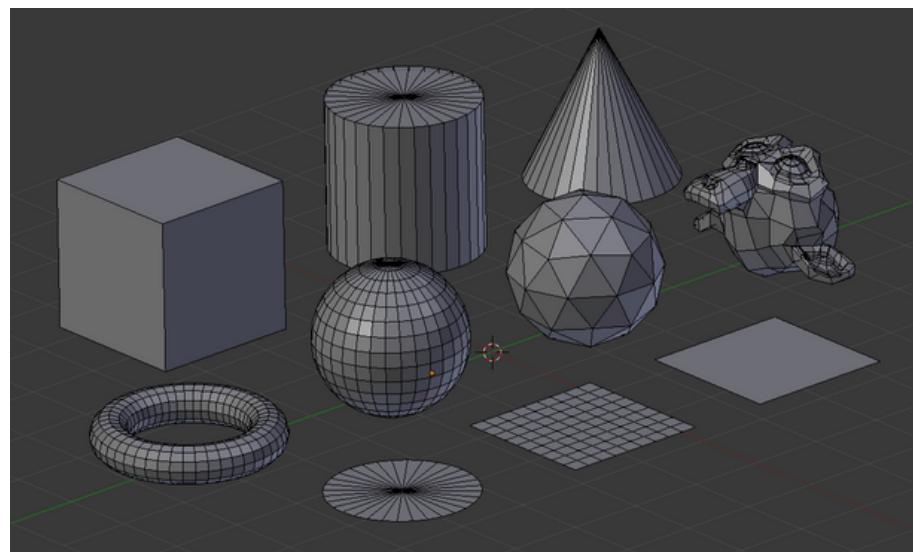


Figura 3.5: Ejemplos básicos de mallas

Ahora ya sabemos que el contorno de cada una de los elementos de la escena se realiza mediante mallas, pero desconocemos el color de la misma. Para dar color a cada una de las caras de una malla, se ha de indicar el color de cada uno de los vértices que lo componen.

.....
²Actualmente los dispositivos móviles tan sólo tratan con mallas definidas por triángulos



Indicar el color de los distintos vértices no es suficiente, también se han de indicar otras propiedades como son el sombreado y si aplican o no texturas, conceptos, que explicamos a continuación.

Sombreado: es un algoritmo mediante el cual se indica cómo se va a calcular el color de los píxeles intermedios dentro de cada polígono. Los principales algoritmos son:

- **Flat:** todos los píxeles del polígono tendrán un único color, por lo que todos los vértices también han de tener ese mismo color. Debido a su simplicidad su coste computacional es muy bajo pero los resultados son muy pixelados y por tanto de peor calidad gráfica.
- **Gouraud:** de mayor detalle que el anterior, se centra en los vértices de cada polígono, calculando el color de cada punto de la superficie interpolando los colores de cada vértice.
- **Phong:** es el algoritmo más preciso para calcular el color por cada píxel, en el cual se interpolan los vectores normales de sus vértices. Por contra su coste es bastante más elevado que Gouraud. Cuando los polígonos de los elementos de la escena son más pequeños que un píxel, los algoritmos Gouraud y Phong tienen el mismo resultado.

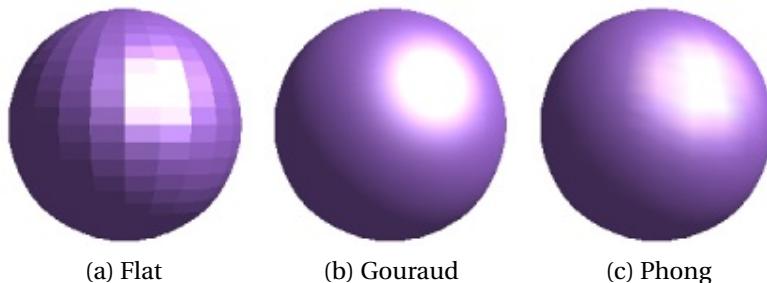


Figura 3.6: Esfera con los tres tipos de sombreado

Texturas: son imágenes que se aplican sobre la malla dándole un mayor realismo. Al aplicar una textura se ha de indicar su mapeo, es decir, cómo se traslada la imagen sobre la malla. Para ello se usan las siguientes propiedades:

- **Repeating:** indica si la imagen se ha de repetir sobre los ejes.
- **Clamping:** indica si se ha de extender el último píxel de la imagen sobre los ejes.

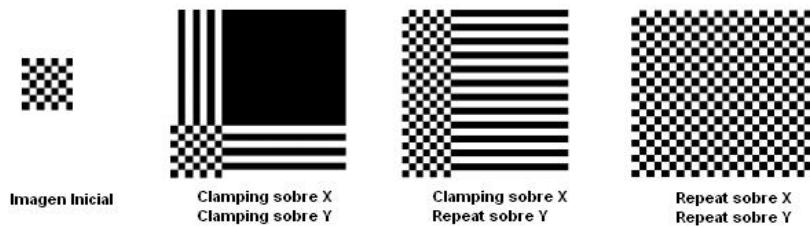


Figura 3.7: Efectos de clamping y repeating

- **UV Mapping:** indica la correspondencia entre los vértices de la malla y los píxeles de la imagen. En función de si la malla es un plano, cubo, tubo o esfera se conocen como flat, cube, tube o sphere mapping.

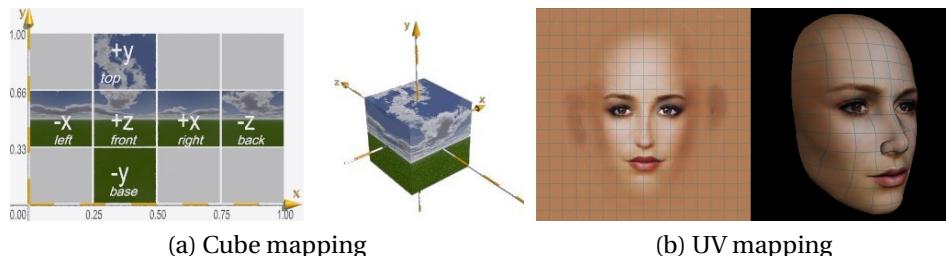


Figura 3.8: Ejemplos de mapeos

- **Función de mapeo:** determina cómo afecta la textura sobre el objeto. Entre las funciones más conocidas podemos destacar:
 - **Displacement mapping:** técnica utilizada para deformar una malla mediante una textura. Consiste en desplazar cada vértice de la malla una distancia determinada por la imagen asociada a la textura.

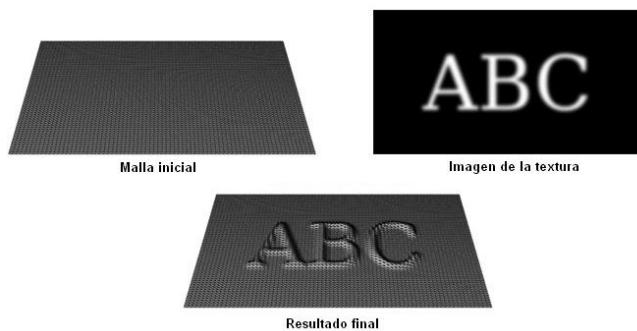


Figura 3.9: Ejemplo de displacement mapping



- **Normal mapping:** técnica que consiste en modificar las normales³ de una malla, aplicando un desplazamiento definido por una textura, de esta forma se consigue un mayor detalle con menos vértices. Hay que tener en cuenta que al no modificar la geometría del objeto tampoco lo harán sus sombras.

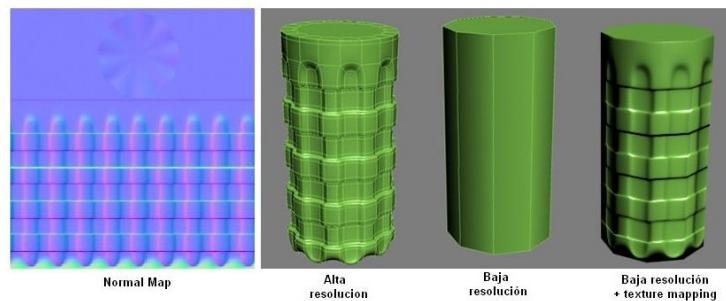


Figura 3.10: Ejemplo de normal mapping

Otras de las ventajas que nos ofrece los modelos 3D es la posibilidad de articular las mallas mediante un nuevo concepto, la armadura.

La armadura es una especie de esqueleto, compuesto por varias articulaciones, que al moverse, se traslada dicho movimiento sobre los vértices cercanos que la componen. En las películas de animación estas armaduras son creadas a partir de unos trajes que contienen multitud de sensores, los cuales transmiten información sobre cualquier movimiento que realizan.

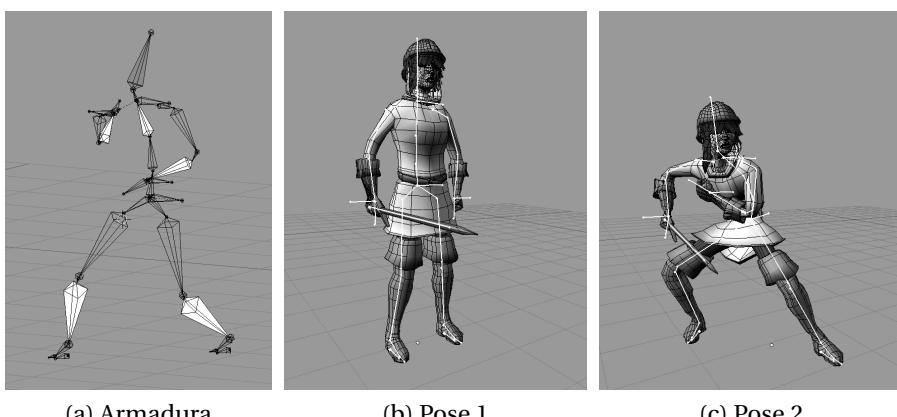


Figura 3.11: Armadura sobre un guerrero

.....
³Las normales o vector normal es el vector perpendicular al plano formado por los vértices.

Cámaras

Mediante el concepto de cámara se indica el lugar desde el cual, la escena va a ser proyectada sobre la pantalla. De cada una de las cámaras de una escena hemos de conocer su posición y orientación mediante los siguientes parámetros:

- **Eye:** define la posición de la cámara en el espacio.
- **Center:** define el punto al cual estamos mirando.
- **Up:** define el vector vertical a la cámara.

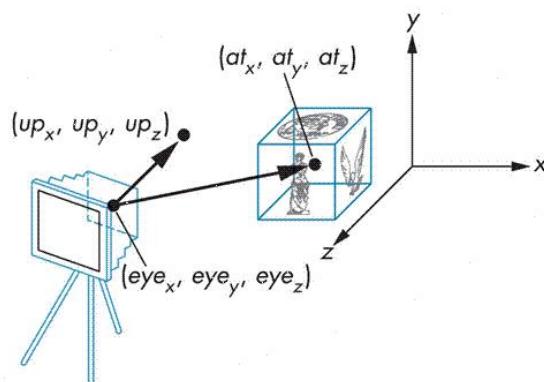


Figura 3.12: Posición y orientación de una cámara

También hemos de establecer los siguientes parámetros de la cámara, a través de los cuales se determinará el área visible de la escena.

- **Fovt:** define el ángulo de apertura de la cámara
- **Aspect:** define la proporción entre ancho y alto de la imagen (16:9, panorámico, 4:3, etc)
- **Near:** define la distancia al plano de corte cercano a la cámara
- **Far:** define la distancia al plano de corte más alejado de la cámara.

Luces

En cada escena hay que indicar los distintos focos de luz existentes, por cada uno de ellos se han de especificar las siguientes propiedades:

- **Color:** de la luz que emite el foco.

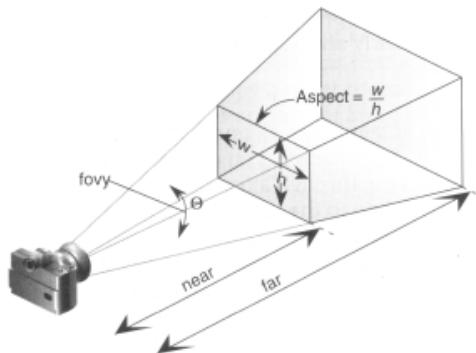
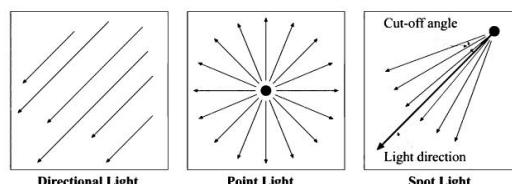


Figura 3.13: Frustrum de una cámara

■ **Intensidad:** con la que la luz es emitida y cómo se va atenuando con la distancia.

■ **Tipo de luz:**

- Directional light: es un tipo de luz ubicada en el infinito, por ejemplo el Sol.
- Point light: es un tipo de luz posicionada en un punto que emite luz en todas las direcciones, por ejemplo, una bombilla.
- Spot light: es un tipo de luz similar al Point Light pero sólo emite protones bajo una superficie conoidal, determinada por un vector direccional y el angulo de corte. Un ejemplo claro de este tipo de luz es un linterna.



(a) Tipos de luces



(b) Iluminación aplicada a un plano

Figura 3.14: Tipos de luces y el efecto sobre un plano

3.4. Modelos de iluminación

En puntos anteriores se han determinado distintos elementos que forman una escena, hemos identificado el color de las mallas, pero el color real en la escena depende de muchos factores. Si observamos un objeto de color blanco a plena luz del día, lo veremos blanco, pero si ese mismo objeto lo encontramos en una habitación con una luz roja, diremos que es rojo, por lo tanto, el color de los focos de luz existentes es un factor a tener en cuenta en la iluminación de una escena. También hay que tener en cuenta que la luz emitida por los focos puede estar siendo bloqueada por otras mallas de la escena, provocando sombras. Los modelos de iluminación son los responsables de determinar el color real del objeto en función de estos factores y muchos otros.

Para poder entender los modelos de iluminación se exponen a continuación una serie de conceptos de óptica básicos y cómo son trasladados a una computadora.

Conceptos ópticos básicos

La óptica puede ser estudiada pensando en su geométrica, física o cuántica, nos centraremos en la geométrica que es la utilizada en las computadoras para simular el comportamiento de la luz.

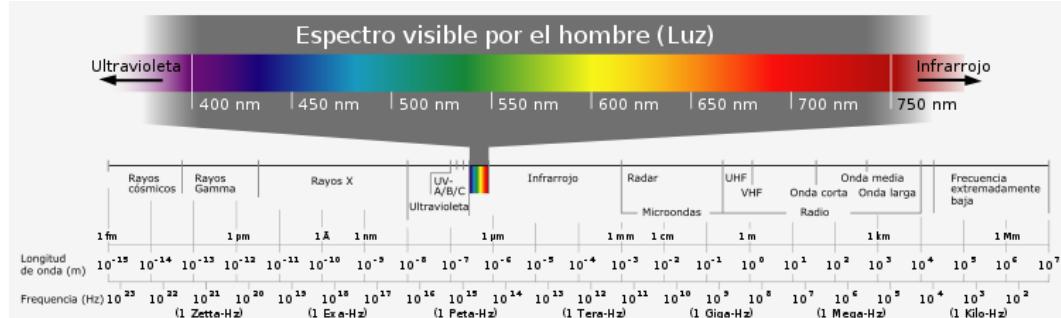


Figura 3.15: Espectro electromagnético de la luz

La óptica geométrica está fundamentada en la teoría de los rayos de luz. Esta teoría considera que todo objeto visible emite rayos rectos de luz en todas direcciones a su alrededor. Cuando estos rayos inciden sobre otros cuerpos, se presentan los siguientes fenómenos ópticos:

Reflexión: el rayo de luz es proyectado en sentido contrario al que incide sobre el objeto. Dependiendo del material del objeto existen dos tipos de reflexión:

- **Difusa:** el rayo incidente es reflejado en un amplio abanico de direcciones con intensidades equivalentes, debido a rugosidades en el material.



- **Especular:** el rayo incidente es reflejado en una única dirección y con el mismo ángulo con el que incide en el objeto.

La reflexión lleva asociada una pérdida de intensidad lumínica en función de las características del objeto con el que haya colisionado el rayo de luz. Cuanta más luz retenga el objeto, menor será la intensidad del rayo reflejado.

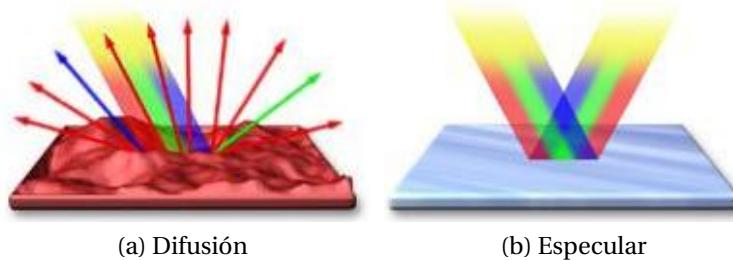


Figura 3.16: Fenómenos de reflexiones difusa y especular

Refracción: es el cambio de dirección que experimenta un rayo de luz al pasar de un medio a otro. Este cambio de dirección es lo que provoca que veamos torcido un lápiz al ser sumergido en el agua.



Figura 3.17: Fenómeno refracción sobre el agua

Absorción: cuando un rayo luminoso se propaga, va disminuyendo paulatinamente su intensidad, siendo absorbida poco a poco por el entorno, dotándole de color. Si un objeto absorbe todos los colores de la luz menos el verde, el ojo humano percibirá dicho objeto de color verde.

Aproximación a un modelo computacional

Hasta el momento hemos hablado del color de las luces y las mallas, lo cual es una simplificación. En un modelo computacional hemos de hablar de la cantidad de luz que absorben las mallas o emiten las luces. El espectro luminoso de la luz se representa por el modelo RGB, es decir, los niveles de rojo, verde y azul.

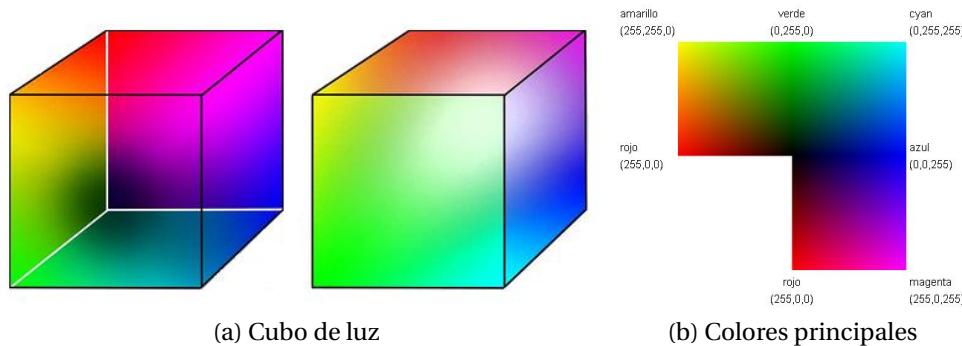


Figura 3.18: Visualización del espectro de luz sobre un cubo RGB

La luz es descompuesta en los siguientes tipos de luces:

- **Luz ambiental:** es aquella que proviene de una fuente que ha sido tan disipada por el entorno que es imposible determinar su dirección.
- **Luz de difusión:** es aquella sobre la que será aplicado el fenómeno de difusión generando objetos con un contorno suavizado y pudiendo observar su forma tridimensional.
- **Luz especular:** es aquella sobre la que será aplicado el fenómeno de reflexión especular, generando brillos en los objetos.
- **Luz de emisión:** se corresponde con la luz que emite un objeto. Por motivos de simplificación, suele tratarse como un incremento de la luz ambiental específica para dicho objeto.

Aplicando este modelo a una luz en nuestro escenario, hemos de establecer el nivel de luz ambiental, de difusión y especular sobre cada uno de los focos de luz y sobre cada objeto. Además deberemos saber cuál será su comportamiento ante el fenómeno de absorción de la luz, es decir, cuáles son los niveles de luz ambiental, difusión y especular que absorberá, además de la luz de que emite el objeto por si mismo.



Clasificación de modelos de iluminación

Ahora que conocemos las propiedades ópticas básicas, podemos centrarnos en el concepto de modelo de iluminación, a través del cual, se va a determinar cómo se va a simular en la computadora el comportamiento de la luz sobre los objetos.

Los modelos se clasifican principalmente por el tipo de iluminación. En el modelo de **iluminación directa** únicamente están implicados los rayos de luz procedentes de una fuente de luz y que colisionan con un objeto. Ahora sabemos que este rayo de luz puede sufrir el fenómeno de reflexión, modificando las propiedades del rayo, el cual puede volver a colisionar con otros objetos. Los rayos procedentes de la reflexión son descartados en modelos de iluminación directa, pero en los que se fundamentan los modelos de **iluminación global**.

Los métodos de iluminación global generan escenas difícilmente distinguible de la realidad, es lo que se conoce como foto-realismo pero no son aplicables al mundo de los videojuegos debido a su elevado coste computacional. Están surgiendo nuevos modelos de iluminación híbridos, dando origen al término de foto-realismo animado.



Figura 3.19: Imágenes del juego Crysis 2 comparadas con la realidad

A continuación se muestra un listado de modelos de iluminación más populares, entre los cuales nos centramos en Phong y Radiosity:

- **Lambert:** modelo de iluminación directa, ideal para superficies plásticas ya que esta basado en superficies difusas perfectas, por lo tanto, sobre el modelo de reflexión especular.

3.4. Modelos de iluminación

- **Oren-Nayar:** modelo de iluminación directa, ideal para superficies fibrosa como ropa, madera y pelo. Está basado en superficies difusas borrosas, es decir, un modelo de reflexión de difusión.
- **Phong:** modelo de iluminación directa basado en superficies especulares, ideal para elementos con superficies parecidas a un espejo sin llegar a serlo, como pinturas con brillo y metales.
- **Radiosity:** modelo de iluminación global basado en el nivel de calor absorbido por los elementos puramente difusos.
- **Ray-tracing:** modelo de iluminación global que simula el trazado de rayos de luz desde el observador. Por cada uno de los rayos simulados se tiene en cuenta los fenómenos de reflexión y refracción.

Modelo de iluminación de Phong

Es un modelo de iluminación directa, desarrollado en 1973 por Bui Tuong Phong en la universidad de Utah, basándose en la combinación de reflexión specular y difusión, nivel de brillo de la superficies, iluminación ambiental y atenuación de la luz.

En el modelo de iluminación de Phong, el color en un punto P en una superficie cuyo material está definido por K , donde

$$K = \{K_{ambiente}, K_{difusión}, K_{especular}\}$$

Y sobre el que se aplica una luz con una intensidad definida por L donde

$$L = \{L_{ambiente}, L_{difusión}, L_{especular}\}$$

Se determina por la suma de las intensidades de luz reflejadas, es decir:

$$\text{Color} = \text{Color}_{ambiente} + \text{Color}_{difusión} + \text{Color}_{especular}$$

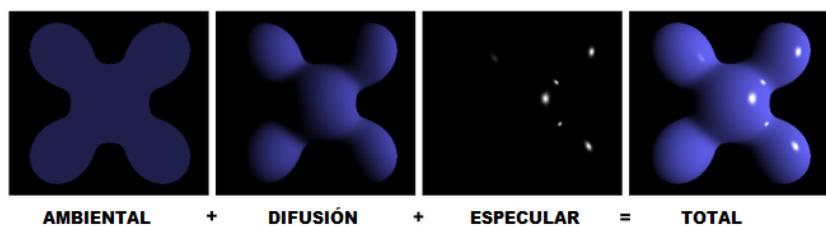


Figura 3.20: Modelo de iluminación de Phong

Para el calculo del color final del objeto se tendrán en cuenta los vectores formados por la posición de la luz, el observador y el punto de colisión entre el rayo de luz y el objeto, a este punto lo llamaremos P . A partir de estos tres puntos se han de calcular:



Capítulo 3. Computación gráfica

- El vector normal de la superficie en P se define como \vec{n} .
- El vector entre el punto donde se ubica la cámara y el punto P se define como \vec{v} .
- El vector entre el origen de la luz y el punto P se define como \vec{l} y forma un angulo θ con respecto a la normal.
- El vector de la luz reflejado en el mismo angulo θ respecto a la normal se define como \vec{r} .

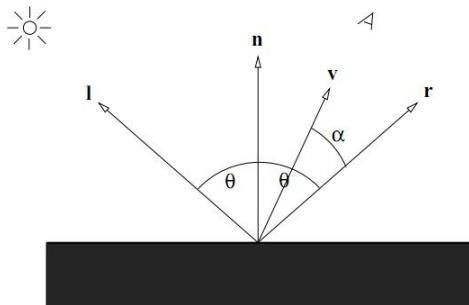


Figura 3.21: Vectores utilizados en el modelo Phong

El componente ambiental se corresponde con el producto de la intensidad de la luz ambiental sobre los materiales de emisión y ambiental, es decir:

$$Color_{ambiental} = (K_{ambiental} + K_{emision}) * L_{ambiental}$$

El componente de difusión se calcula con la intensidad de la luz de difusión, aplicada al material correspondiente, pero teniendo en cuenta el ángulo existente entre la normal y el vector de la luz. Cuanto más pequeño sea el ángulo entre los vectores, mayor será su producto escalar, por lo que recibirá más luz. Si el punto de luz está muy alejado, el producto vectorial puede resultar negativo, por lo que no llegará luz y el valor se redondeará a cero.

$$Color_{difusión} = K_{difusión} * L_{difusión} * \max(\vec{n} \cdot \vec{l}, 0)$$

El componente especular depende del punto de vista del observador sobre el rayo reflejado. El producto escalar entre ambos vectores es tratado de igual forma que en el componente de difusión.

$$Color_{especular} = K_{especular} * L_{especular} * \max(\vec{r} \cdot \vec{v}, 0)$$

Modelo de iluminación por radiosidad

La radiosidad se fundamenta en el estudio térmico de la luz debido a la vibración de los fotones, determinando que una superficie estará más iluminada cuanto más fotones choquen contra ella o lo que es lo mismo, tenga un mayor nivel de radiosidad.

Nos encontramos con un modelo de iluminación global y con un comportamiento de los objetos puramente difuso. Todos los rayos de luz son reflejados de forma homogénea y con la misma intensidad en todas direcciones, restándole parte de su calor o intensidad lumínica inicial, debido al fenómeno de absorción.

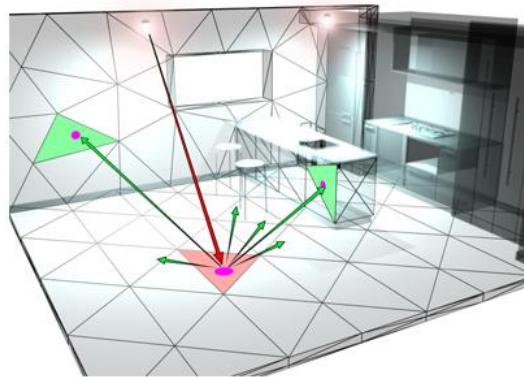


Figura 3.22: Modelo de radiosidad donde podemos observar en rojo una luz directa que es reflejada y sus rayos indirectos que se muestran en verde

La radiosidad en un determinado punto, B_i se define como la energía por unidad de área que emite una superficie por unidad de tiempo, que no es más que la suma de la luz emitida por la superficie y la energía reflejada proveniente de otras superficies:

$$B_I = E_i + R_i \sum B_j F_{ij}$$

Donde:

- E_i : energía emitida por la superficie i .
- R_i : capacidad de reflexión de la superficie i .
- B_j : energía transmitida de la superficie j sobre i .
- F_{ji} : factor de forma entre i y j que mide la cantidad de energía que emitida por j llega a i .



Finalmente, en cada punto de la escena, se calcularán los componentes ambiental, especular y de difusión del escenario, basándose en la intensidad de luz, calculada previamente en el modelo de radiosidad.

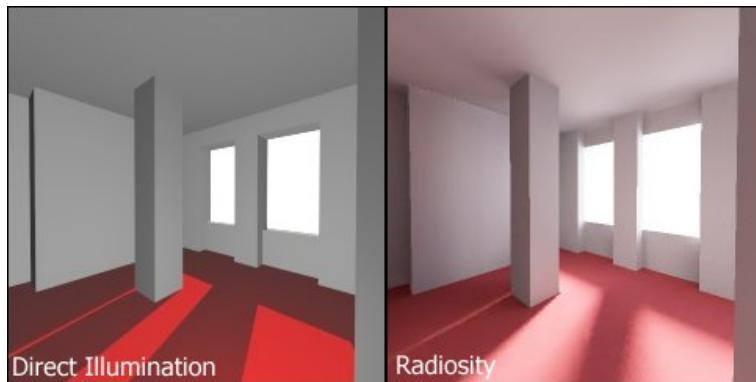


Figura 3.23: Comparativa de una escena procesada con radiosidad y phong

Las ventajas y desventajas de este modelo de iluminación son las siguientes:

- La calidad de la imagen es cercana a la realidad.
- El tiempo de procesamiento es elevado.
- Se puede generar un procesamiento progresivo, es decir, en cada paso se puede ver la escena, incrementando su realismo cuantos más pasos se realicen. Cada paso consiste en calcular los nuevos rayos generados por la reflexión.
- La radiosidad de la escena no cambia ante cambios de cámara por lo que no ha de ser recalculada por los movimientos de la cámara.
- La radiosidad de la escena cambia al cambiar la posición de cualquier elemento de la escena, pudiendo cambiar la trayectoria de algunas rayos y por lo tanto, la radiosidad de las superficies. Este es el motivo por el cual no resulta útil en el desarrollo de videojuegos.
- Para calcular la radiosidad de la escena no se usan los polígonos que componen las distintas mallas debido a su elevado número, para acelerar este cálculo se utilizan unas superficies más amplias. Debido a esta simplificación se pueden occasionar algunos errores en el cálculo de la radiosidad.

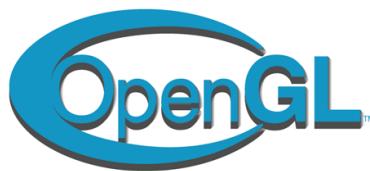
3.5. Librerías de computación gráfica

Debido a la complejidad de las escenas en tres dimensiones, surgen unas librerías de uso común para simplificar estas tareas al desarrollar aplicaciones. Las librerías más conocidas en el mercado son:

- La desarrolla por Microsoft, **DirectX**, que está limitada para entornos con un sistema operativo Windows.
- Las basadas en el API de **OpenGL**, implementadas para sistemas operativos Unix, Linux, Windows, etc.



(a) DirectX



(b) OpenGL

Figura 3.24: Logotipos de las librerías

DirectX no está soportada en Android, por lo que nos centraremos en OpenGL, para ser más exactos, sobre **OpenGL ES**, una versión más optimizada para los sistemas con un bajo nivel de procesamiento. En esta versión se han simplificado sus operaciones pero sin alterar su funcionalidad. Existen tres versiones:

OpenGL ES 1.0: procede de OpenGL 1.3 e implementa un modelo de iluminación de Phong en el pipeline fijo del microprocesador de la tarjeta gráfica.

OpenGL ES 1.1: es una evolución de la 1.0 añadiendo ciertas mejoras de OpenGL 1.5.

OpenGL ES 2.0: procede de OpenGL 2.0, que da un gran salto al incluir un pipeline dinámico, al que poder incorporar en algunos de sus pasos los shaders, a partir de los cuales se puede crear un modelo de iluminación acorde a las necesidades de cada proyecto.

Los shaders son códigos fuente compilados e interpretados por los microprocesadores de las tarjetas gráficas en tiempo de ejecución. Están escritos en un nuevo lenguaje llamado GLSL (Graphics Library Shaders Languaje). Para entender cómo funcionan hemos de conocer la estructura de OpenGL 2.0.



OpenGL 2.0

Toda tarjeta gráfica que soporte OpenGL 2.0 estará compuesta por un pipeline con una estructura similar al mostrado en el siguiente gráfico. Se han identificado con cajas sombreadas aquellas fases en las que es posible insertar shaders.

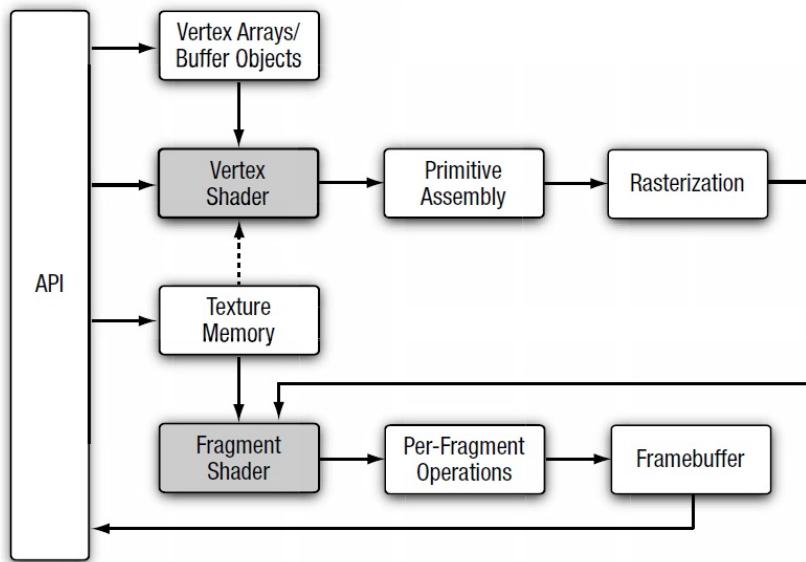


Figura 3.25: Pipeline en OpenGL ES 2.0

La primera fase `Vertex Array/ Object` se corresponde a la adaptación de los datos definidos en el API a la estructura hardware dependiente de la tarjeta, por este motivo no haremos hincapié en ella.

Vertex Shader

En esta segunda fase se realizan operaciones sobre los vértices de cada malla, de uno en uno, por lo que se desconoce la forma de la malla. Las operaciones más frecuentes son transformaciones afines, proyecciones, transformaciones de color, coordenadas de textura e iluminación. La entrada de esta fase está constituida por:

- **Atributos:** propiedades inherentes a los vértices (coordenadas de textura, vector normal...) que pueden ser definidas o creadas por la aplicación de OpenGL para cada vértice.
- **Uniforms:** propiedades generales a todos los vértices que pueden ser definidas por la aplicación de OpenGL como una constante.
- **Samplers:** se corresponden con las texturas, siendo opcionales en esta fase.

3.5. Librerías de computación gráfica

Es obligatorio definir el **GL_Position**, que se corresponde con la posición final del vértice. Las variables **GL_PointSize** y **GL_FrontFacing** son opcionales. Indican el tamaño en píxeles del vértice y la ecuación del plano de corte respectivamente. Para comunicarse con otras fases del pipeline se utilizan variables de tipo varying.

A modo de ejemplo, a partir de un vertex shaders, podríamos ser capaces de modificar los vértices de una malla junto con una textura, aplicando la técnica de displacement mapping, descrita en la página 39, cuando se definía el concepto de malla.

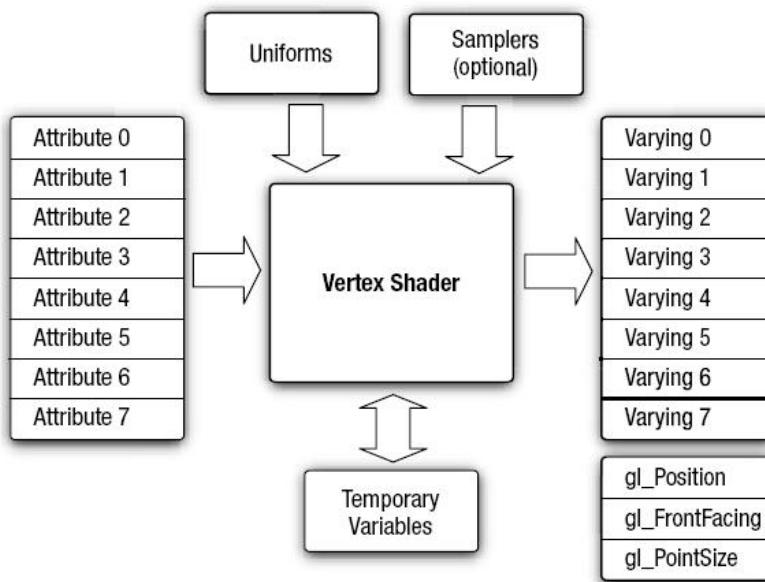


Figura 3.26: OpenGL ES 2.0 Vertex Shaders

Primitive Assembly y rasterización

La tercera fase consisten en agrupar los vértices formando puntos, líneas o polígonos para después ejecutar la rasterización y obtener los fragmentos que van a componer la imagen.

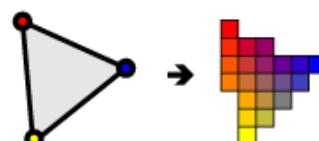


Figura 3.27: Ejemplo de rasterización



Fragment Shader

En la cuarta fase, programable mediante un shader, recibe los fragmentos obtenidos en la fase anterior. Sobre estos segmentos se pueden aplicar operaciones de mapeo de texturas, mezcla de colores, efecto de niebla...

El objetivo del fragment shaders es conseguir el color del fragmento, almacenado en la variable **GL_FragColor**, utilizando un modelo de iluminación con los siguientes datos y las variables de tipo uniform y varying, definidas previamente:

- **GL_FragCoord**: indica las coordenadas del fragmento.
- **GL_FrontFacing**: indica si el fragmento está oculto por otro fragmento.
- **GL_PointCoord**: coordenadas de dispersión aplicado sobre las partículas⁴.

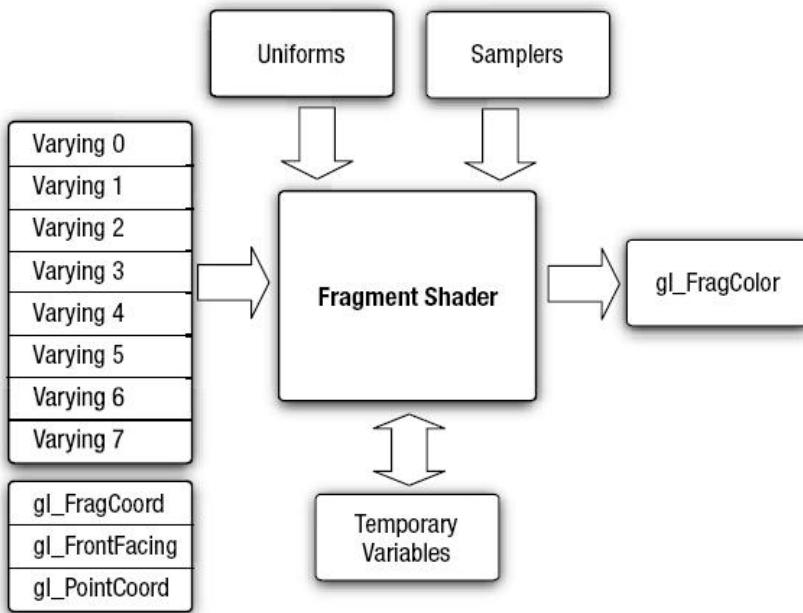


Figura 3.28: OpenGL ES 2.0 Fragmen Shaders

Algunas de las características que pueden ser implementadas en el fragment shader son la aplicación de mapeos usando texturas, simulación de efectos de niebla, etc.

.....
⁴El concepto de partículas esta fuera del alcance del proyecto y es usado para simular objetos muy dinámicos como son fuegos artificiales.

Per-Fragment

En esta última fase se genera el buffer final, equivalente a los píxeles de la pantalla. Sobre cada uno de los fragmentos se ejecutan los siguientes test, aunque inicialmente están todos deshabilitados:

- **Scissor box testing:** elimina todos los píxeles fuera del área de visualización, el cual se determina con las propiedades de la cámara.
- **Stencil buffer testing:** genera una máscara de la imagen a partir de una función `glStencilFunc`. La aplicación más conocida de este stencil buffer consiste en la generación de sombras.
- **Depth buffer testing:** elimina todos aquellos fragmentos que no están visibles al ser ocultados por otros.
- **Blending:** fusiona el color de los segmentos que se solapen en pantalla. A través de esta técnica se puede generar el efecto de transparencia de elementos como el cristal.
- **Dithering:** ajusta los colores disponibles que no se adaptan a los colores que han sido calculados con el modelo de iluminación. Éste tipo de técnica consiste en intercalar los píxeles para que visualmente ofrezca un color similar al que deseamos.

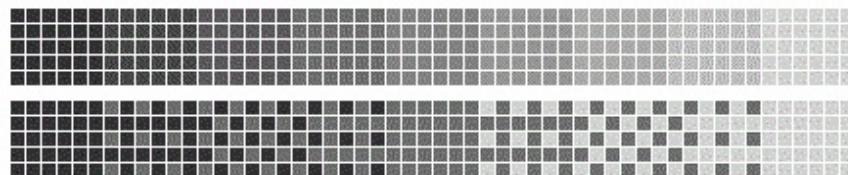


Figura 3.29: Ejemplo de dithering sobre escala de grises

CAPÍTULO

4

Leyes físicas en computadoras

En este capítulo se explicará la relación existente entre la física y los videojuegos. Las dos líneas a tratar serán:

- Conocer la nueva posición de un objeto en movimiento transcurrido un tiempo determinado.
- Saber cómo detectar cuando dos objetos han colisionado.

Estos conocimientos son necesarios para poder comprender las decisiones de diseño tomadas en la implementación del TfcGameEngine.



4.1. Aplicación de la física en un videojuego

A la hora de desarrollar un videojuego, nos encontraremos con la necesidad de simular el comportamiento de un mundo real. Esto implica incluir los efectos físicos existentes en nuestro entorno como son la velocidad, aceleración, gravedad...

Desde el punto de vista de desarrollo de un videojuego nos podríamos preguntar lo siguiente:

- ¿Cuál será la posición de un elemento, transcurridos 3 segundos, que viaja a una velocidad y dirección determinada?
- ¿Cómo sabemos si dos objetos han chocado?
- ¿Qué ocurre cuando choca una bola de billar contra otra en un determinado ángulo y fuerza?
- ¿Cómo se mueve el líquido de un vaso situado en un objeto que está acelerando?

En una computadora no existen leyes físicas, por lo tanto, han de ser recreados matemáticamente simulando las leyes físicas. De no ser así, podrían ocurrir cosas extrañas en la realidad, como que dos objetos ocuparan el mismo espacio.

En las siguientes secciones, para poder responder a las preguntas formuladas, se va a realizar un acercamiento sobre el origen del movimiento y para finalizar se estudiará el cálculo de las colisiones entre objetos.

4.2. Origen del movimiento

El movimiento es un fenómeno físico, el cual se puede definir como cualquier cambio de posición, en el espacio, que experimentan los cuerpos de un sistema con respecto a ellos mismos o a otro cuerpo, que se toma como referencia, definiendo una trayectoria.

En física, la rama que estudia y analiza el movimiento de los cuerpos es la mecánica, la cual se divide en 4 grandes ramas:

- Mecánica clásica (no relativista)
- Mecánica clásica relativista
- Mecánica cuántica
- Mecánica cuántica de campos

4.2. Origen del movimiento

Pondremos nuestra atención sobre la mecánica clásica, la cual se ajusta más al mundo físico de un videojuego, el resto de ramas se centran en el estudio del movimiento a nivel atómico o de objetos cercanos a la velocidad de la luz.

Un subconjunto de la mecánica clásica es la física Newtoniana, que nos permite realizar un estudio de objetos sólido-rígidos, es decir, objetos que no se pueden deformar, por otro lado, la física de medios continuos nos permite el estudio de los objetos son fluidos o sólidos deformables.

En del ámbito del proyecto nos centraremos en la física de objetos sólidos, aunque en los juegos de última generación incluyen física de fluidos y objetos deformables. Debido a la complejidad de éste tipo de cálculo y con el objetivo de optimizar al máximo los recursos, existen implementaciones directamente en el hardware de las tarjetas gráficas, como es el caso de PhysX de la compañía Nvidia.

El siguiente diagrama muestra un árbol con las distintas ramas en las que se divide la física mecánica. En color verde y naranja se han identificado las ramas que han sido aplicadas totalmente o parcialmente en el proyecto, sobre las cuales se profundizará en las siguientes secciones. En color amarillo se indican aquellas ramas que ya forman parte de los motores de juegos profesionales.

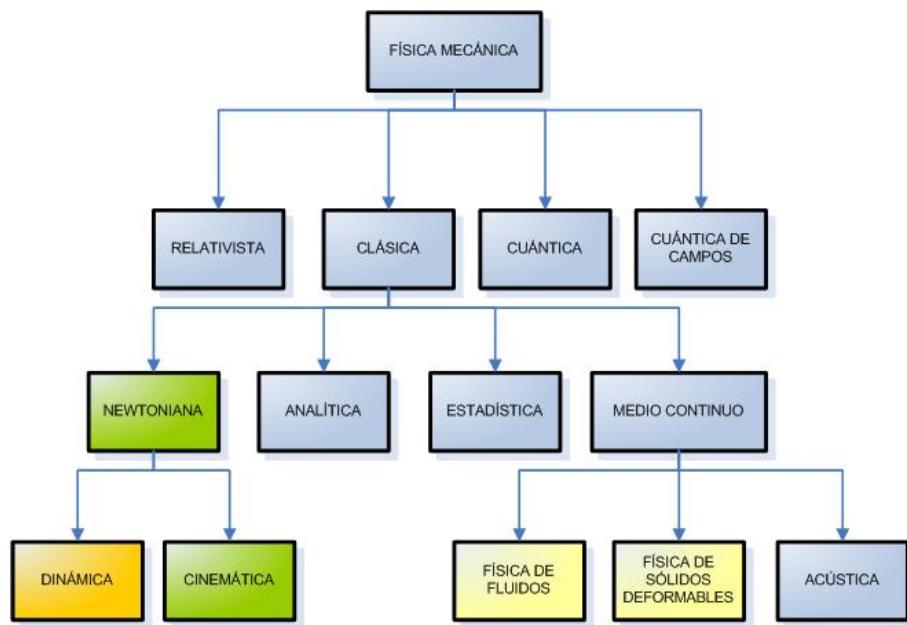


Figura 4.1: Disciplinas de la física mecánica



Cinemática

La cinemática se centra en las trayectorias de los objetos sin tener en cuenta las causas que producen el movimiento. Utiliza un sistema de coordenadas para describir las trayectorias, denominado sistema de referencia. La velocidad y la aceleración son las dos principales cantidades que describen cómo cambia su posición en función del tiempo.

- **Posición (\vec{x}):** situación del objeto respecto al sistema de referencia.
- **Velocidad (\vec{v}):** ritmo con que cambia la posición un cuerpo.
- **Aceleración (\vec{a}):** La aceleración es el ritmo con que cambia su velocidad
- **Tiempo (t)**

A continuación se muestran las trayectorias típicas en la cinemática y cómo se calcula la posición final ' x ' trascurrido un tiempo ' t '.

Movimiento rectilíneo uniforme: describe una trayectoria recta a una velocidad constante ' v ', sin aceleración y desde una posición inicial \vec{x}_0 .

$$\vec{x} = \vec{x}_0 + \vec{v} t$$

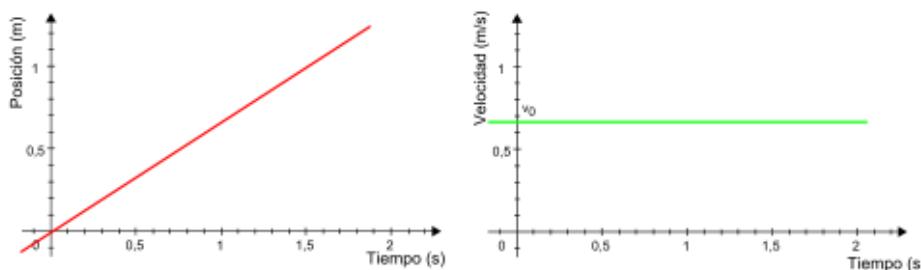


Figura 4.2: Evolución del movimiento respecto al tiempo

Movimiento rectilíneo uniformemente acelerado: describe una trayectoria recta desde un punto \vec{x}_0 , una velocidad inicial \vec{v} y una aceleración constante \vec{a} .

$$\vec{x} = \vec{x}_0 + \vec{v} t + \frac{1}{2} \vec{a} t^2$$

4.2. Origen del movimiento

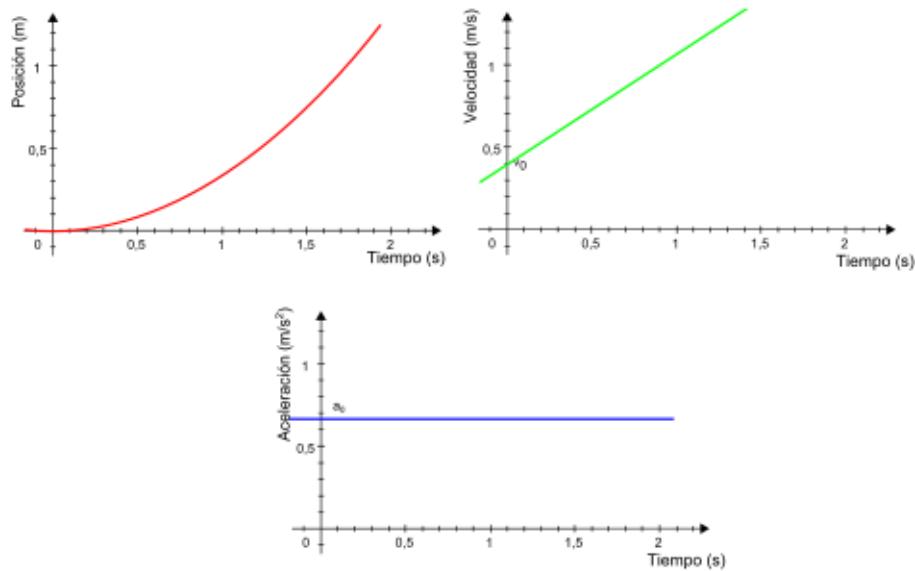
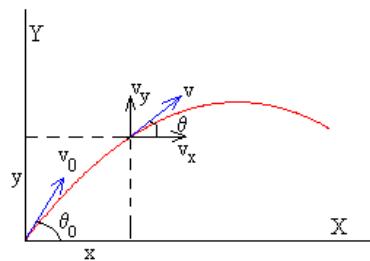


Figura 4.3: Evolución del movimiento respecto al tiempo

Movimiento parabólico: es la unión de un avance horizontal rectilíneo uniforme y un lanzamiento vertical hacia arriba, que es un movimiento rectilíneo uniformemente acelerado hacia abajo por la acción de la gravedad. Este movimiento parte de una posición inicial definida por 'x' e 'y' y un ángulo θ .

$$x = v_{inicial} * \cos\theta * t$$

$$y = v_{inicial} * \sin\theta * t + 1/2at^2$$



Existe otros tipos de movimientos como son los circulares, helicoidales y armónicos simples (muelles), pero están fuera del alcance del proyecto.

Dinámica

La dinámica se fundamenta en el estudio de las fuerzas que se aplican sobre objetos, siendo una fuerza un cambio de aceleración. Las leyes de Newton que fundamentan esta parte de la física son las siguientes:

- Todo cuerpo permanece en su estado de reposo o de movimiento rectilíneo uniforme a menos que otros cuerpos actúen sobre él.



Capítulo 4. Leyes físicas en computadoras

- $\vec{F} = m\vec{a}$ La fuerza que actúa sobre un cuerpo es directamente proporcional al producto de su masa y aceleración.
- Cuando un cuerpo ejerce una fuerza sobre otro, éste ejerce sobre el primero una fuerza igual y de sentido opuesto.

El objetivo principal que buscamos en la dinámica es conocer la implicación que tiene un choque entre dos objetos. Como podemos observar, por la segunda ley de Newton, debemos conocer el tiempo que es aplicada una fuerza sobre un objeto, para poder obtener su aceleración y con ella, aplicar los conocimientos vistos en el apartado de cinemática. Conocer el tiempo que se está aplicando una fuerza sobre un objeto implica conocer la duración de un impacto, lo cual es sumamente complejo, es por ello que surge el concepto de momento.

El momento es la magnitud física, de tipo vectorial, que mide la capacidad de ejercer una fuerza de un objeto sobre otro en función de su movimiento.

El momento lineal obedece a una ley de conservación, lo cual significa que en todo sistema cerrado (o sea uno que no es afectado por fuerzas exteriores, y cuyas fuerzas internas no son disipadoras) no puede ser cambiada y permanece constante en el tiempo.

En el mundo de los videojuegos existe un alto interés en el choque de dos objetos y la repercusión en la trayectoria del objeto. Además puede que los objetos estén conectados por una visagra o cualquier otro tipo de unión, llamada junta, implicando restricciones de movimiento, que se sumarían a las implicaciones en el movimiento de los objetos.

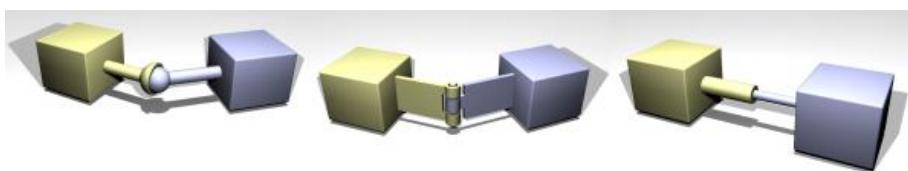


Figura 4.4: Ejemplos de tipos de juntas

Cálculo de un choque plástico

Aunque el juego a desarrollar en el proyecto no ha de calcular las implicaciones del choque entre dos elementos, se considera de especial interés el cálculo de choques plásticos, lo que permitirían desarrollar juegos tan famosos como Angry Birds.

4.3. Detección de colisiones

En un choque plástico, los objetos involucrados no se deforman, de lo contrario, sería un choque elástico. La deformación de un objeto implicaría cálculos adicionales debido a una absorción del impacto por la estructura del objeto, implicando un cambio del momento lineal.

Dados dos objetos o_1 y o_2 plásticos que se mueven a velocidades \vec{v}_1 y \vec{v}_2 y con masas m_1 y m_2 respectivamente, como resultado de la ley de conservación del momento lineal tenemos que:

$$m_1 \vec{v}_1 + m_2 \vec{v}_2 = \vec{v}(m_1 + m_2)$$

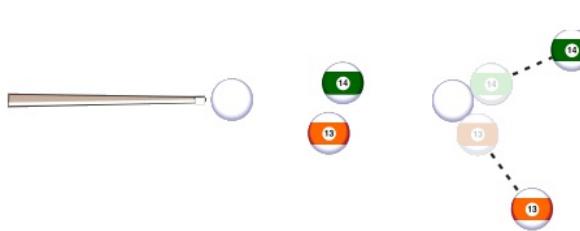
Por lo que la variación de velocidad \vec{v} debido al choque será:

$$\vec{v} = \frac{m_1 \vec{v}_1 + m_2 \vec{v}_2}{m_1 + m_2} =$$

Aplicando la variación de velocidad a ambos objetos, las velocidades finales $\vec{v}_{1_{final}}$ y $\vec{v}_{2_{final}}$ se corresponderán con las siguientes fórmulas:

$$\vec{v}_{1_{final}} = \vec{v}_1 + \vec{v}$$

$$\vec{v}_{2_{final}} = \vec{v}_2 + \vec{v}$$



(a) Juego de billar



(b) Angry Birds

Figura 4.5: Ejemplos de choque plástico

4.3. Detección de colisiones

Ya podemos calcular la posición de los objetos según transcurre el tiempo, creando en el videojuego la ilusión de movimiento. Este desplazamiento puede provocar que dos objetos choquen entre sí debido a la composición atómica de los elementos. La física atómica impide que ambos objetos ocupen el mismo espacio, pero en los videojuegos no existen estas restricciones atómicas, por lo que han de ser calculadas matemáticamente.



En un juego en dos dimensiones el cálculo de colisiones consiste en detectar si un píxel está siendo ocupado por dos elementos. Cuando usamos tres dimensiones el cálculo de colisiones se complica. Lo primero que se ha de tener en cuenta es la forma del objeto, definida mediante una malla y que en la terminología asociada a las colisiones, se define como la **forma explícita** del objeto.

Debido a la complejidad inherente de las mallas surge un problema de eficiencia en el cálculo de colisiones, por ese motivo, las mallas de los elementos son simplificadas, creando lo que se define como la **forma implícita** del objeto.

Forma implícita de un objeto

La forma implícita está compuesta por figuras básicas que permiten simplificar los cálculos para detectar las colisiones. Dependiendo de cómo se ajuste la forma implícita a la explícita, el cálculo de colisiones será más preciso, pero posiblemente más costoso.

Las figuras básicas se componen la forma implícita, llamadas **superficies de contorno** (Bounding Volumenes), son volúmenes geométricos cuyo tamaño requerido para ser representados en memoria es mínimo y es fácil verificar si existe o no colisión entre ellos. A continuación se muestran las superficies de contorno más comunes.

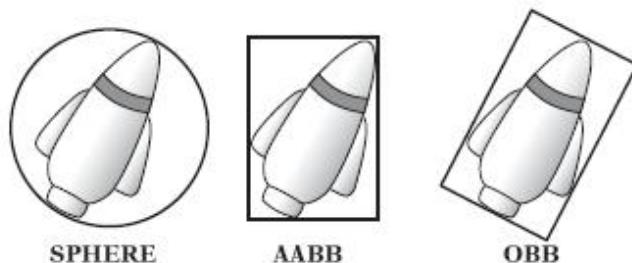


Figura 4.6: Volúmenes de contorno: Esferas, AABB (axis-aligned bounding box): cajas alineadas con los ejes y OBB (oriented bounding box): cajas orientadas respecto a los ejes.

En un videojuego, los objetos no son estáticos, sobre ellos se pueden aplicar distintas transformaciones como traslaciones, rotaciones o movimientos derivados de una armadura¹. Todos estos cambios implican volver a calcular la forma implícita del objeto. A continuación se puede ver cómo afectaría una rotación sobre un avión cuyas formas implícitas se han generado con esferas o AABB.

¹El concepto de armadura se definió en el capítulo 3 sobre computación gráfica.

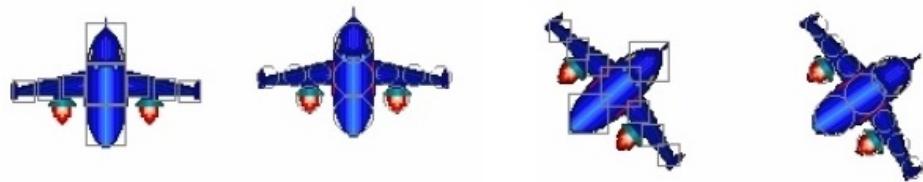


Figura 4.7: Formas implícitas transformadas tras una rotación

Aunque estén fuera del alcance del proyecto, hay que mencionar la existencia de distintos algoritmos mediante los cuales es posible convertir mallas en formas implícitas. El siguiente gráfico muestra un ejemplo del resultado obtenido al aplicar uno de ellos.



Figura 4.8: Simplificación de una malla de un conejo mediante esferas

Fases en la detección de colisiones

A pesar de esta simplificación, un objeto puede colisionar con cualquier otro de la escena, por lo tanto, el cálculo de colisiones para una escena con n objetos requeriría comprobar $(n - 1) + (n - 2) + \dots + 1 = \frac{n(n-1)}{2}$ posibles colisiones, es decir, existe una complejidad de $O(n^2)$.

Para simplificar esta situación el cálculo de colisiones se divide en dos fases:

Broad phase: es la fase preliminar en la que se crea una división espacial de la escena en áreas que permitan descartar colisiones entre objetos de la escena rápidamente. Las técnicas de división más habituales son:

- **BSP (Binary Space Partitioning):** se divide la escena recursivamente en dos particiones, generando un árbol binario, donde cada nodo está compuesto por dos hijos.



Un objeto podría ubicarse entre dos particiones, como es el caso del objeto *E* del gráfico. En vez de modificar la forma de la partición, se indica que el objeto esté en ambos nodos.

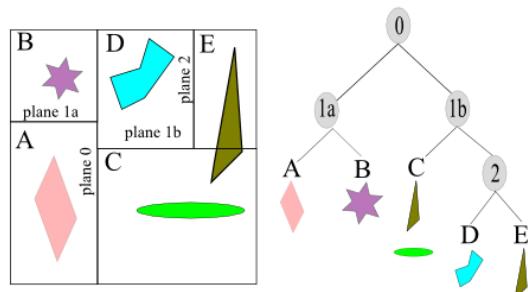


Figura 4.9: BSP

- QuadTree: es un árbol similar al BSP donde cada nodo contiene cuatro hijos en vez de dos.

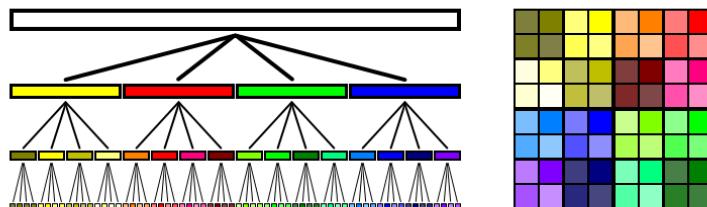


Figura 4.10: QuadTree

- OcTree: es un árbol donde cada nodo contiene 8 hijos.

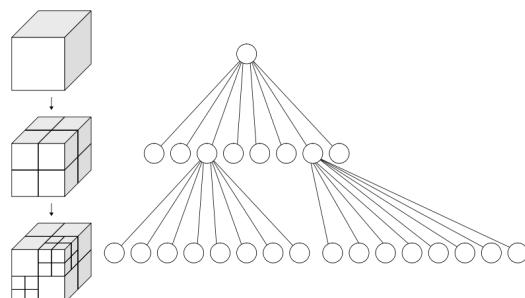


Figura 4.11: OcTree

Narrow phase: en esta segunda fase se determina si dos objetos, que están ubicados en la misma partición, están colisionando. Se aplicará un test de colisión que dependerá de la superficie de contorno elegida para crear las formas implícitas de los objetos. En la siguientes secciones se detallarán cómo se detecta si existe o no colisión para esferas y AABB.

Colisiones entre esferas

Una esfera es un cuerpo geométrico, limitado por una superficie curva cerrada, cuyos puntos equidistan de otro interior llamado centro de la esfera. Para representarla, tan sólo necesitamos en punto \vec{c} que representa el centro y la distancia r de la que equidistan los puntos que definen la superficie.

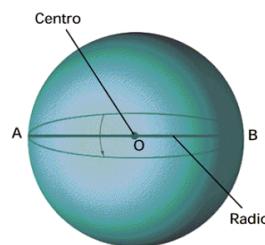


Figura 4.12: Representación de una esfera

Para determinar si dos esferas colisionan tan sólo hemos de verificar, como se indica a continuación, que la distancia entre sus centros es menor que la suma de sus radios.

Dadas dos esferas, e_1 y e_2 con radios \vec{r}_1 y \vec{r}_2 y centros \vec{c}_1 y \vec{c}_2 respectivamente, están colisionarán si:

$$distancia(e_1, e_2) > r_1 + r_2$$

$$distancia(e_1, e_2) = \sqrt{(c_{1x} - c_{2x})^2 + (c_{1y} - c_{2y})^2 + (c_{1z} - c_{2z})^2}$$

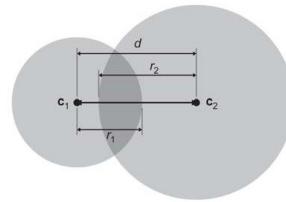


Figura 4.13: Colisión de esferas

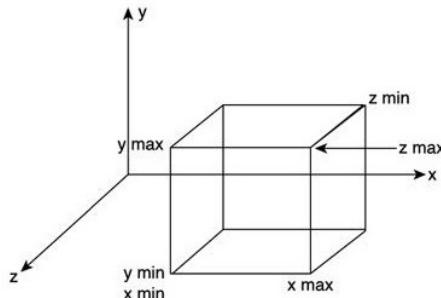


Colisiones AABB

Las AABB (Axis-aligned bounding box) son cajas alineadas con los ejes. Debido a ésta característica es posible definir su geometría definiendo únicamente dos puntos \vec{p}_{max} y \vec{p}_{min} donde:

$$\vec{p}_{max} = \{x_{max}, y_{max}, z_{max}\}$$

$$\vec{p}_{min} = \{x_{min}, y_{min}, z_{min}\}$$



Para comprobar si dos objetos a y b , con formas implícitas de tipo AABB, colisionarán hay que verificar que no se dan estas condiciones:

1. $a_{max_x} < b_{min_x}$ o $a_{min_x} > b_{max_x}$
2. $a_{max_y} < b_{min_y}$ o $a_{min_y} > b_{max_y}$
3. $a_{max_z} < b_{min_z}$ o $a_{min_z} > b_{max_z}$

Errores en el cálculo de colisiones

Una vez implementado un detector de colisiones es posible que nos encontremos con dos posibles errores. El primero de ellos viene derivado de la simplificación realizada al usar las formas implícitas como una aproximación a la forma explícita. Es posible que la formas implícitas de los elementos colisionen, pero no lo harían si lo hicieramos el cálculo con su forma explícita.

En el siguiente gráfico muestra de forma más clara este tipo de error. Está formado por varios elementos delimitados por una línea en negro, sus forma implícita por una líneas rojas, en verde un error al detectar una colisión inexistente.

Para evitar este tipo de error se ha de elegir una forma implícita más parecida a la forma explícita. Si en el ejemplo, en vez usar un cuadrado para componer la forma implícita, usamos varios, ajustándolo más a su forma explícita, el error desaparecerá. Por contra, al ser la forma implícita más compleja, se requerirán más operaciones matemáticas para detectar la colisión. Por lo tanto, es aconsejable llegar a un punto intermedio.

4.3. Detección de colisiones

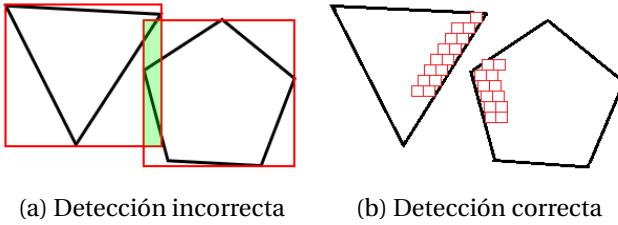


Figura 4.14: Ejemplo de error de colisión

Otro posible error en la detección de colisiones es conocido como el efecto túnel. Se debe a que los objetos están en movimiento y las colisiones se calculan por intervalos de tiempo, cuanto más rápida sea la computadora menor será el tiempo transcurrido entre detecciones y menor será la probabilidad de percibir un error.

Si este cálculo se realiza en los instantes t_0 y t_2 , puede existir un instante t_1 , entre ambas iteraciones, donde estemos pasando por alto una colisión, tal y como ocurre en el siguiente gráfico.

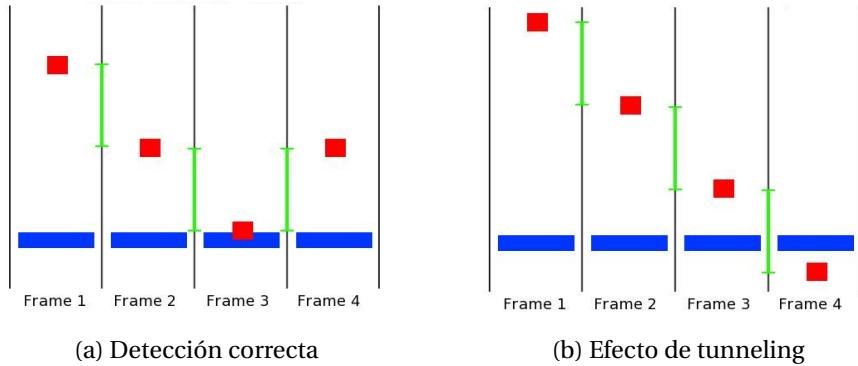


Figura 4.15: Ejemplo de posible error de tunneling mostrada en 2D

Para evitar el efecto túnel de forma efectiva se realizan unos test de colisiones más sofisticados, que quedan fuera del alcance del proyecto. Son los llamados algoritmos continuos que calculan los volúmenes de colisión que son generados por los objetos en movimiento, por ejemplo, una esfera generará una especie de cilindro.

Parte III

Desarrollo del proyecto

CAPÍTULO

5

Requisitos del proyecto

El objetivo que persigue este capítulo es definir todos los requisitos que ha de cumplir el proyecto. Se establecerá la nomenclatura utilizada para nombrar cada requisito de forma inequívoca. Después, para cada una de las partes en las que ha sido dividido el proyecto, la librería y el videojuego, se detallarán uno a uno dichos requisitos.



5.1. Nomeclatura

Cada requisito del proyecto vendrá identificado mediante la siguiente notación:

Nombredelproyecto : Módulo : Índice

- El nombre del proyecto se corresponderá con ‘TFC’, las siglas de ‘Trabajo Fin de Carrera’.
- El módulo se corresponde con ‘PACMAN’ si es un requisito del videojuego o con ‘LIB’ si es un requisito de la librería.
- Por último, el índice será un valor número autoincremental.

Tras haber definido cómo se van a identificar los requisitos, en las siguientes secciones se irán describiendo uno a uno. El objetivo que se persigue con estos requisitos es fijar el alcance final de proyecto con la mayor exactitud posible.

5.2. Requisitos del videojuego

TFC:PACMAN:001 El flujo de pantallas de la aplicación se define en el gráfico de la figura 5.1 que se muestra en la página siguiente.

TFC:PACMAN:002 Al iniciar la aplicación se mostrará la pantalla de ‘Inicialización’ en la cual, se irá indicando las acciones realizadas por la aplicación, entre ellas, lecturas de ficheros, cargas de imágenes...

TFC:PACMAN:003 Al finalizar en proceso de carga de la aplicación, se cambiará automáticamente de la pantalla de ‘Inicialización’ a la pantalla de ‘Menú principal’. Sobre el menú principal se podrán realizar las siguientes acciones:

- | | |
|-------------------|----------------------|
| ▪ Acerca de | ▪ Marcadores |
| ▪ Configuración | ▪ Salir del programa |
| ▪ Iniciar partida | |

Cada una de las acciones tiene asociado un cambio de pantalla, que se describirá en posteriores requisitos. El flujo de estas acciones está descrito en el gráfico primer requisito funcional TFC:PACMAN:001.

TFC:PACMAN:004 La opción ‘Acerca De’, ubicada en el menú principal, abrirá una ventana emergente que mostrará la siguiente información.

5.2. Requisitos del videojuego

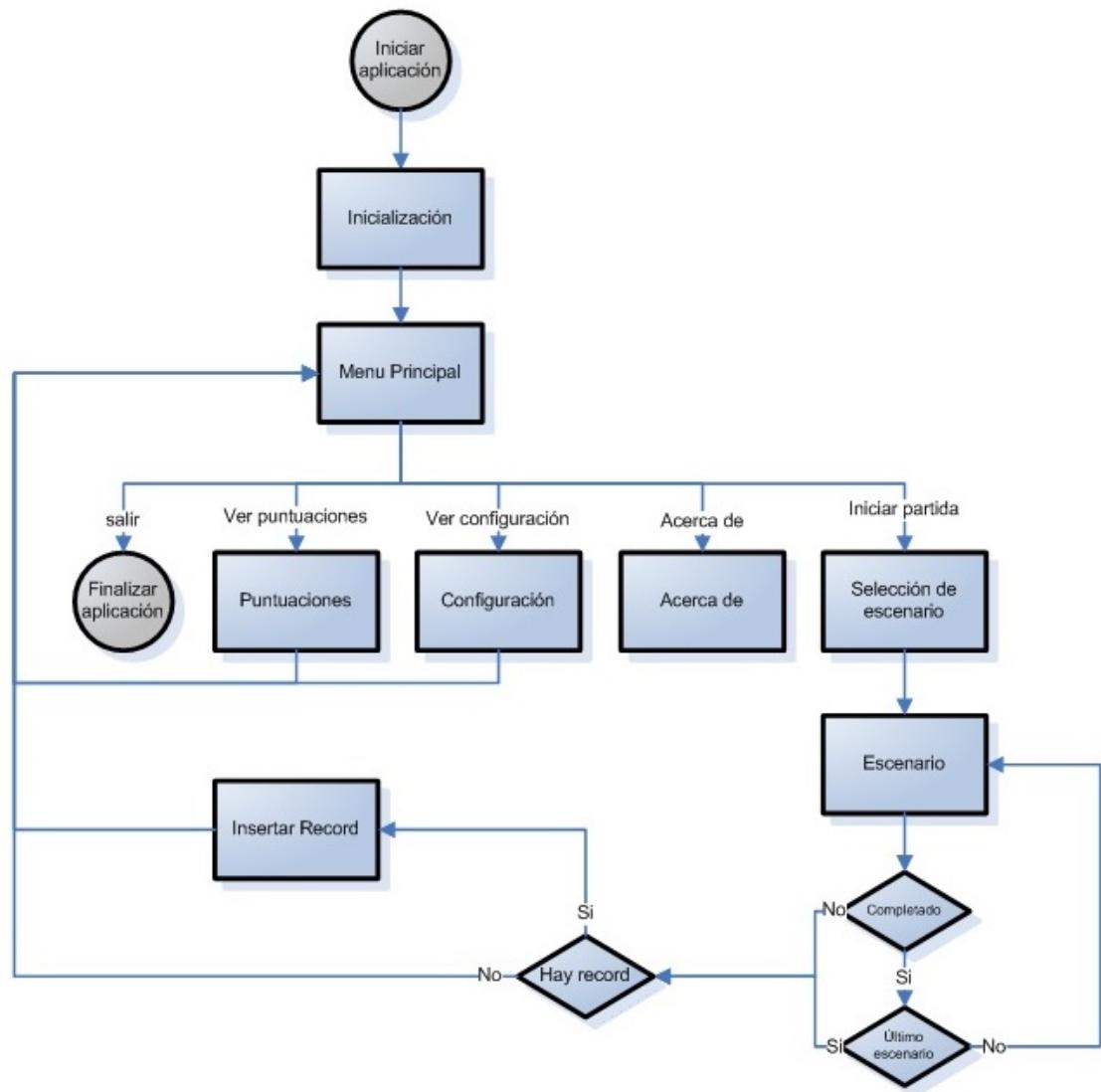


Figura 5.1: Estructura general de la aplicación



Capítulo 5. Requisitos del proyecto

- Autor del proyecto
- Tutor del proyecto
- Universidad
- Facultad
- Año de realización

TFC:PACMAN:005 La opción 'Marcadores', ubicada en el menú principal, mostrará una nueva ventana con las diez mejores partidas realizadas en el videojuego y en el smartphone donde ha sido instalada la aplicación.

Cuanto mejor ha sido una partida mayor puntuación habrá obtenido, por lo tanto, se ordenaran de mayor a menor puntuación.

En cada posición se indicará un texto con el nombre de la persona que consiguió el récord, y la puntuación lograda.

TFC:PACMAN:006 La opción 'Configuración', ubicada en el menú principal, mostrará una nueva ventana en la cual se permitirá establecer los siguientes criterios:

- Sonido: indica si el sonido del juego está encendido o apagado.
- Path: directorio donde se encuentran los ficheros de configuración.
- Módulo: listado de escenarios que compondrán el videojuego. Los posibles módulos a cargar están ubicados en el directorio de configuración.

TFC:PACMAN:007 La opción 'Iniciar Partida', ubicada en el menú principal, mostrará una nueva ventana, en la cual se mostrarán los distintos escenarios que contiene el modulo cargado. El usuario podrá seleccionar cualquiera de ellos, al hacerlo, se cargará el escenario seleccionado.

TFC:PACMAN:008 Al finalizar el escenario el sistema comprobará si se ha realizado con éxito, si es así, se cargará el siguiente escenario definido en el módulo.

En el caso de no haber completado con éxito el escenario actual o de no existir más escenarios se volverá al menú principal.

Si la puntuación obtenida está entre las diez mejores, antes de volver al menú principal, aparecerá una ventana para indicar el nombre asociado a ese nuevo récord.

TFC:PACMAN:009 Cada escenario contendrá los siguientes elementos:

- Pacman: elemento dinámico que será manejado por el usuario.
- Fantasmas: elementos dinámicos gestionados por la computadora cuyo objetivo es capturar al Pacman.
- Pastillas: elementos estáticos que han de ser recogidos por el Pacman.

5.2. Requisitos del videojuego

TFC:PACMAN:010 El Pacman puede realizar dos tipos de movimientos, los cuales se pueden simultanear:

- Desplazamientos sobre el plano XY, es decir, sobre una cuadrícula, por lo tanto siempre son horizontales o verticales.
- Desplazamientos sobre el plano Z que se corresponden con saltos y junto con un desplazamiento en el eje X o Y implicará un tiro parabólico.

TFC:PACMAN:011 En cualquier momento el Pacman podrá cambiar el sentido del movimiento, incluso cuando se este produciendo un salto.

TFC:PACMAN:012 La dirección de un elemento sólo podrá cambiar en los cruces de la cuadrícula.

TFC:PACMAN:013 Ninguno de los elementos del juego podrán atravesar las paredes del escenario.

TFC:PACMAN:014 Los fantasmas únicamente tendrán la posibilidad de realizar movimientos sobre el eje XY, por lo tanto no saltarán.

TFC:PACMAN:015 Los posibles algoritmos asociados al comportamiento automático de un fantasma son los siguientes:

- Aleatorio: se elegirá una dirección al azar en la que se moverá el fantasma en el siguiente cruce.
- Persecución: seguirá el camino mínimo existente en el escenario entre su posición y la del Pacman.

TFC:PACMAN:016 Los fantasmas podrán cruzarse entre sí, por lo que no se realizará ninguna acción cuando colisionen.

TFC:PACMAN:017 Al iniciar una partida el usuario tiene cero puntos y tres vidas u oportunidades para superar todos los escenarios.

TFC:PACMAN:018 Cuando el Pacman choca contra una pastilla, ésta desaparece del escenario y se incrementa la puntuación en el marcador en 10 unidades.

TFC:PACMAN:19 Cuando el Pacman choca contra una fantasma se pierde una vida.

TFC:PACMAN:20 El escenario se dará por finalizado cuando el Pacman haya recogido todas las pastillas contenidas en él.

TFC:PACMAN:21 La partida finaliza bajo dos situaciones:

- El usuario pierde todas las vidas.



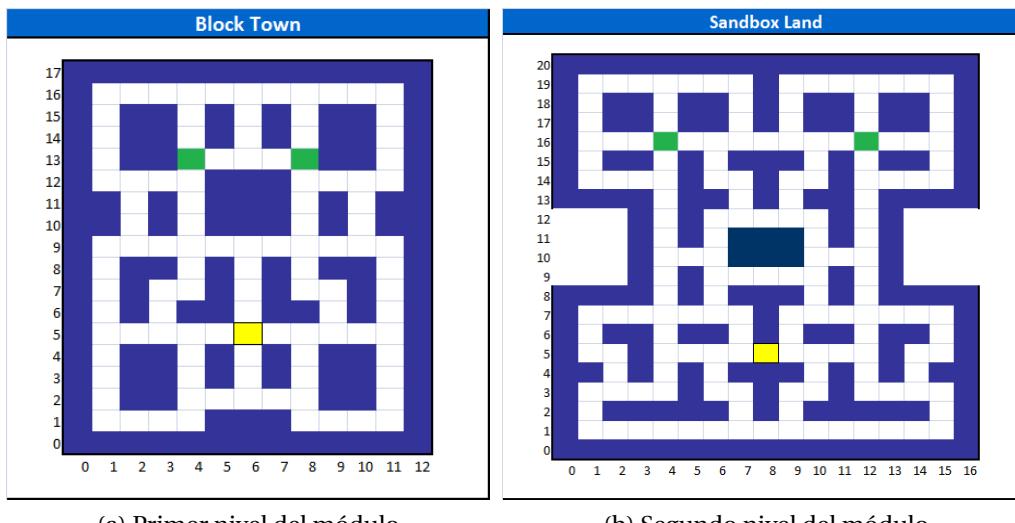
Capítulo 5. Requisitos del proyecto

- El usuario completa todos los escenarios del módulo.

TFC:PACMAN:22 La partida podrá pausarse en cualquier momento.

TFC:PACMAN:23 Se permitirá pausar el videojuego en el transcurso de un escenario. Si el videojuego está en pausa, el escenario cambiará su color a escala de sepias.

TFC:PACMAN:024 Ha de crearse un módulo llamado "Pacmania", compuesto por los siguientes niveles:



(a) Primer nivel del módulo

(b) Segundo nivel del módulo

Sobre cada uno de los escenarios se identifica:

- El cuadrado amarillo identifica la posición inicial de Pacman.
- Los cuadrados azules son las paredes del escenario
- Los cuadrados verdes son las posiciones iniciales de los fantasmas.

TFC:PACMAN:025 Al finalizar todos los niveles del módulo se mostrará un vídeo de fin de partida completada.

TFC:PACMAN:026 Al perder todas las vidas se mostrará un vídeo distinto de fin de partida no completada.

TFC:PACMAN:027 Antes de mostrar cada escenario se procederá a mostrar un vídeo introductorio.

TFC:PACMAN:028 Cada escenario tendrá asociada una melodía que se escuchara de fondo, durante el transcurso de la partida, en dicho escenario.

5.3. Requisitos de la librería

TFC:PACMAN:029 Cada módulo ha de tener una melodía que se escuchará desde el principio del juego hasta seleccionar un escenario. A partir de ese momento será sustituida por la melodía del escenarios seleccionado.

TFC:PACMAN:030 Al chocar con una pastilla se realizara un sonido.

TFC:PACMAN:031 Al perder una vida se reproducirá un sonido.

TFC:PACMAN:032 Durante la partida, la cámara irá siguiendo los movimientos del Pacman.

TFC:PACMAN:033 Al empezar cada escenario se realizará un efecto de acercamiento con la cámara hacia el Pacman.

TFC:PACMAN:034 Al perder una vida se realiza una animación girando la cámara 720 grados.

5.3. Requisitos de la librería

TFC:LIB:001 Se adaptará el componente gráfico de OpenGL para Android, llamado GLSurfaceView, dotándole de funciones específicas de un videojuego. Cuando el sistema operativo refresque la pantalla se realizarán las siguientes acciones.

- Calcular el tiempo transcurrido desde la última actualización.
- Obtener nuevas posiciones de los distintos elementos del escenario. Se han de tener en cuenta las propiedades de los distintos elementos y el tiempo transcurrido.
- Ejecutar un pre-render que permita al programador ampliar con nuevas funcionalidades el componente gráfico.
- Actualizar el escenario en pantalla con la nueva información calculada.

TFC:LIB:002 La estructura de datos a partir de la cual se definirá el escenario estará compuesta por:

- Nombre y descripción de la escena.
- Una cámara.
- Malla del escenario.
- Listado de elementos que componen la escena.

TFC:LIB:003 Los comportamientos posibles de una cámara son:

- **Estático:** La cámara no se mueve.



Capítulo 5. Requisitos del proyecto

- **Programada:** La cámara sigue una trayectoria previamente configurada mediante rotaciones sobre los ejes y los puntos inicial y final. Para dicho comportamiento también se han de indicar la velocidad lineal y angular.
- **Persecutoria:** La cámara sigue a un elemento del juego desde una distancia predeterminada.
- **Compuesta:** La cámara realiza una composición de los comportamientos mencionados.

TFC:LIB:004 La estructura de datos que determina la información de un elemento de la escena esta compuesta por:

- Las coordenadas de su posición en el escenario.
- Propiedades físicas de cinemática para creación de movimiento rectilíneos uniformes en un espacio tridimensional.
- Aspecto gráfico del elemento.
- Forma implícita.
- Comportamiento del elemento, en el cual se determina si es gestionado de forma manual o bajo inteligencia artificial.
- Detección de colisiones automáticas habilitado.

TFC:LIB:005 La algoritmos de inteligencia artificial disponibles, determinando el comportamiento de un elemento, son los siguientes:

- **Manual:** el usuario interactúa con el juego describiendo trayectorias sobre la pantalla, las cuales indican hacia donde debe moverse el elemento.
- **Aleatorio:** el elemento se mueve sobre el escenario cambiando su dirección y sentido de forma aleatoria.
- **Dijkstra:** el elemento se mueve por la ruta mas corta hacia un determinado punto.

Todos estos comportamientos se basan en movimientos sobre una cuadrícula.

TFC:LIB:006 Las formas implícitas de los elementos del juego se definen mediante superficies de contorno de tipo AABB o la composición de las mismas.

TFC:LIB:007 El comportamiento ante colisiones de un elemento puede ser **Fuerte** o **Débil**. El comportamiento fuerte sólo ha de asociarse a elementos que, cuando existas una colisión con ellos, no tengan acciones asociadas en el videojuego, pudiendo ser delegadas directamente a la librería. El ejemplo más claro son las paredes de un escenario, cuando un elemento choque con ellas, la única acción a realizar es resolver la colisión.

5.3. Requisitos de la librería

TFC:LIB:008 El motor del videojuego calculará las colisiones existentes cada vez que se actualice el componente gráfico de Android. Las colisiones que tengan un elemento cuyo comportamiento sea fuerte, se resolverán situando el elemento con comportamiento débil, justo antes de que se produjera la colisión.

TFC:LIB:009 El aspecto gráfico de cada elemento ha de estar determinado por uno de los siguientes tipos de mallas:

- **TextureShaderMesh:** se corresponde con elementos cuya apariencia está representada por una malla, teniendo en cuenta los vértices, las caras, los vértices de la textura y las texturas.
- **ParticleShaderMesh:** se corresponde con un elemento que esta compuesto por partículas¹, cada una de ellas equivale a un vértice de la malla que se representará con un punto.
- **AnimatedShaderMesh:** se corresponde con elementos cuya apariencia se asocia con una secuencia de mallas, de esta forma es posible la creación de animaciones sin necesidad de usar esqueletos.

TFC:LIB:010 La librería tendrá una utilidad que permitirá leer ficheros Wavefront (.obj y .mtl) para generar mallas de tipo TextureShadersMesh.

Este tipo de fichero son generados a partir del la aplicaciones de diseño 3D como Blender y 3D Studio, las cuales son usadas a nivel profesional para crear películas de animación y videojuegos.

TFC:LIB:011 El motor de juegos no implementará ningún modelo de iluminación. Los efectos provocados por las luces sobre los elementos del videojuego, serán procesados previamente en sus texturas, a partir de las aplicaciones de diseño 3D.

La generación de texturas que incluyen la iluminación de la escena se conoce como ‘cocinar las texturas’ (Texture Bake).

TFC:LIB:012 La librería permitirá reproducir dos tipos de sonidos:

- Una única melodía, que se corresponder con la música de fondo de una pantalla.
- Los efectos especiales que se corresponden con los distintos sonidos que se producen en el videojuego, como por ejemplo colisiones entre los distintos elementos, alcanzar ciertas puntuaciones...

¹Todas las pastillas del juego son representadas por un único elemento de tipo ParticleShaderMesh. Cada pastilla se corresponde con un vértice o partícula de este tipo de mallas, permitiendo, con una sola invocación a la tarjeta gráfica, dibujar todas las pastillas del escenario de forma óptima.

CAPÍTULO

6

Diseño TfcGameEngine

El objetivo del capítulo es comprender el diseño de la librería TfcGameEngine. Para facilitar la comprensión del mismo, se mostrarán diagramas en notación UML. En el primer punto se realiza una introducción sobre los módulos que componen la librería. Después se explicará en detalle cada módulo, haciendo hincapié en aquellos puntos donde existe una vinculación entre la librería y la tarjeta gráfica a través de OpenGL.



6.1. Visión general

En capítulos anteriores se ha hablado de Android y del concepto Actividad. Al instanciar una clase de tipo Actividad, se han de indicar los componentes gráficos que la componen. GLSurfaceView es el componente gráfico que nos va a permitir comunicarnos con la tarjeta gráfica mediante el lenguaje OpenGL ES 2.0.

GLSurfaceView contiene un objeto de tipo Renderer al que envía eventos. La misión del renderer es responder a estos eventos dibujando sobre la superficie de la pantalla que ocupa el GLSurfaceView.

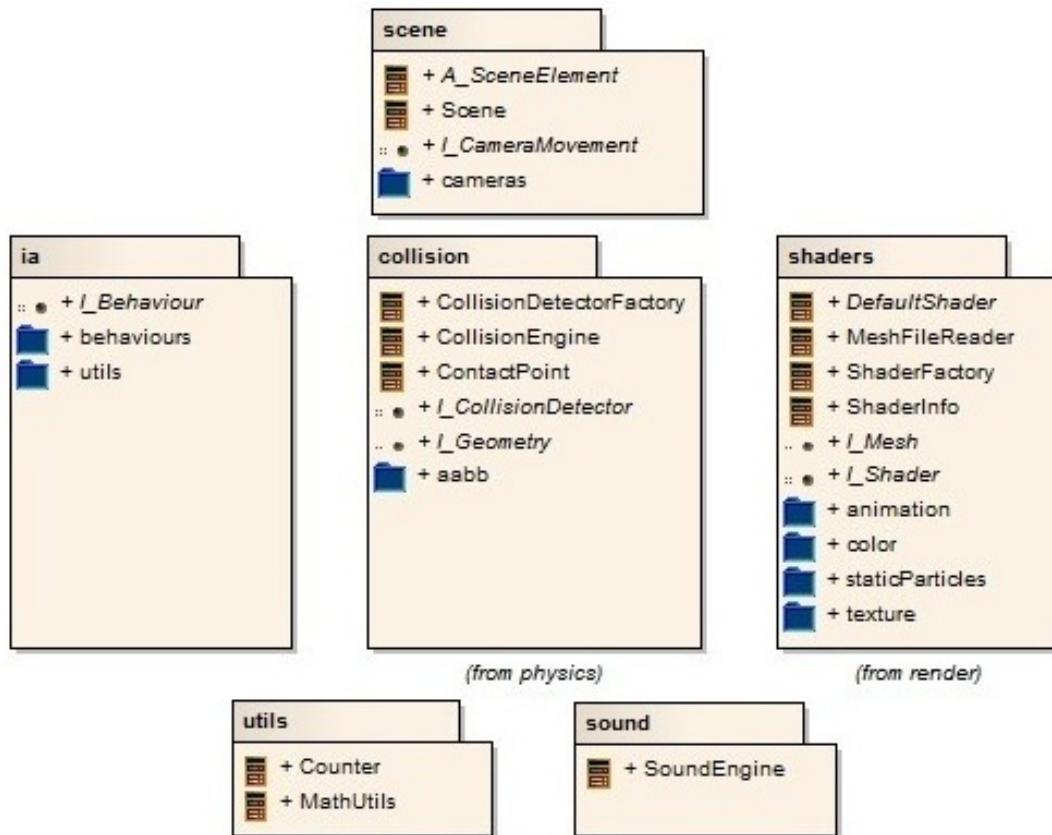


Figura 6.1: Diagrama principal de la librería TfcGameEngine

El componente principal de la librería es la clase **GameEngine** que implementa las funcionalidades de un Renderer, adaptado a las necesidades de un videojuego. Para cumplir esta tarea, se apoya en los siguientes módulos:

- **Scene:** contiene las clases que definen la estructura de datos con la información de la escena.
- **IA:** contiene las clases que implementan los algoritmos de inteligencia artificial que se aplican a los elementos de un juego.
- **Collision:** contiene las clases que definen las formas implícitas de los elementos de la escena y detectan las colisiones entre ellos.
- **Shaders:** contiene las clases que definen las mallas y los shaders.
- **Sound:** contiene las clases que simplifican las funcionalidades de sonido de Android, permitiendo sintetizar melodías y efectos de sonido más fácilmente.
- **Utils:** contiene un conjunto de utilidades adicionales que nos permiten realizar operaciones matemáticas, realizar contadores...

Una vez definidos los distintos módulos que componen la librería, el siguiente punto a tratar antes de poder adentrarnos en el comportamiento de la clase GameEngine, es el módulo Scene. De esta forma se facilita la comprensión del GameEngine ya que conoceremos toda la información que va a gestionar sobre la escena.

6.2. Módulo Scene

Para mostrar la información que contiene los datos de una escena se han utilizados dos diagramas UML de clases. **Scene** es la clase principal del módulo y da nombre al mismo. Cada escena del juego es representada por una instancia de la clase Scene y las propiedades que contiene son:

- Nombre asignado a la escena.
- Descripción de la escena.
- Path o ruta donde se encuentra los ficheros de configuración de la escena.
- El estado en que se encuentra la escena, por ejemplo ejecutándose o en pausa.
- Cámara desde la cual se va a renderizar la escena.
- Propiedades físicas de la escena, como son la gravedad y el tamaño de una celda para juegos basados en casillas.
- Listado de elementos que han de renderizarse.
- Listado de elementos que pueden colisionar.



Capítulo 6. Diseño TfcGameEngine

- Listado de elementos dinámicos, es decir, que tienen la capacidad de moverse por el escenario.

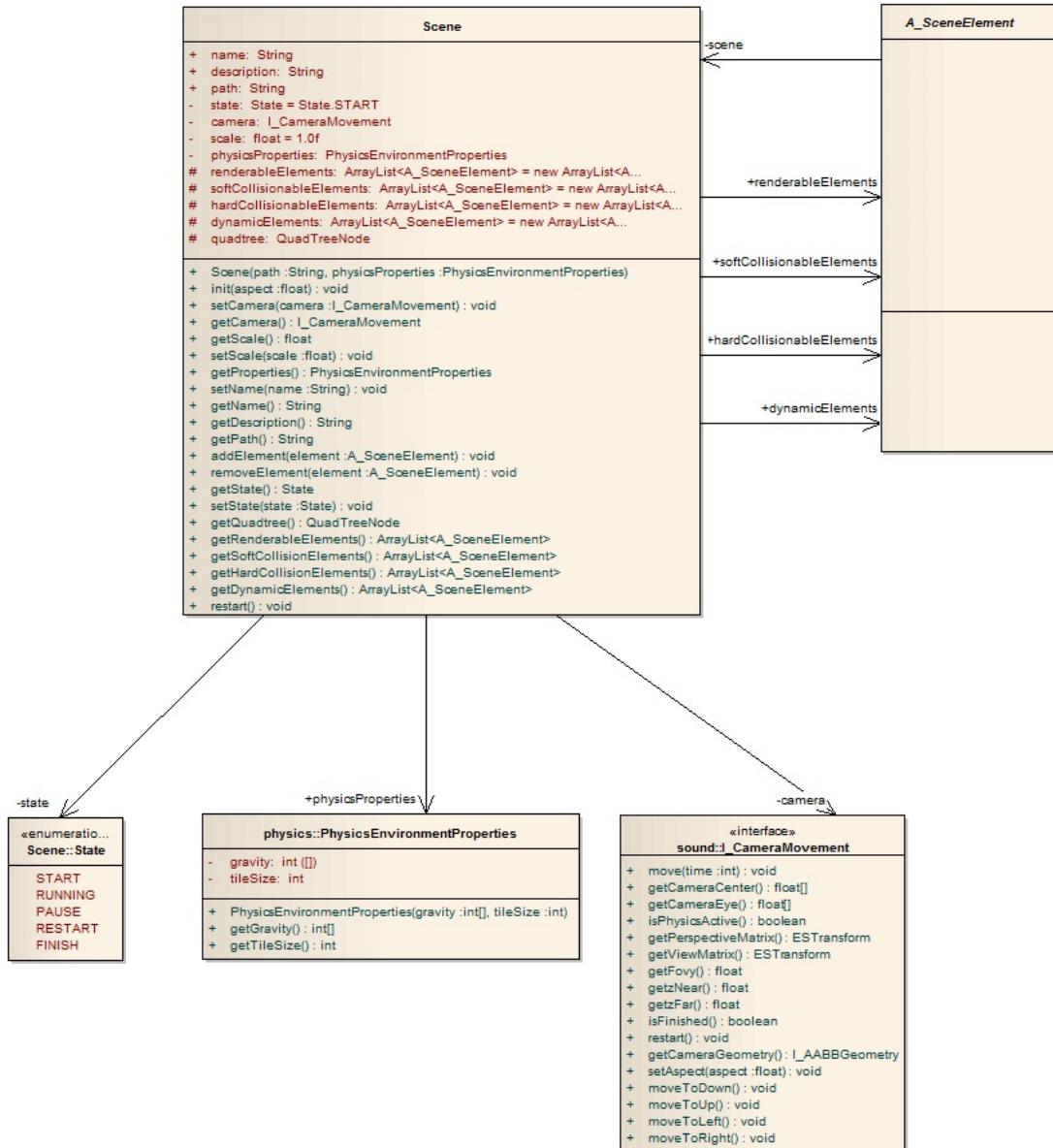


Figura 6.2: Diagrama de clases -Scene

Los elementos

Cada uno de los elementos contenidos en la escena son instancias de la clase **A_SceneElement** y sus propiedades principales son:

- Posición en el espacio tridimensional mediante las coordenadas (x, y, z).
- Vector velocidad (V_x, V_y, V_z).
- Vector de aceleración (a_x, a_y, a_z).
- Vector que indican el tiempo durante el cual se aplica dicha aceleración ($t_{a_x}, t_{a_y}, t_{a_z}$).
- Casilla de referencia para escenarios basados en cuadrículas.
- Si el objeto se ha de renderizar.
- Si el objeto es dinámico, es decir, si puede moverse sobre la escena.
- El tipo de colisión del elemento, existiendo dos tipos de colisiones suaves (Soft) o duras (Hard)¹.
- Forma implícita² del elemento ubicada en el centro del eje de coordenadas.
- Forma procesada del elemento que consiste en trasladar la forma implícita da a la posición actual del objeto.
- Comportamiento del elemento en el escenario³.
- La malla que representa el elemento junto con la escala y la rotación que se han de aplicar.

.....
¹Las implicaciones del tipo de colisión se explicarán en el módulo Collision.

²La clase que define la forma implícita del elemento, se explicará en detalle en el módulo Collision

³El tipo de comportamientos de un elemento se explicarán en detalle en el módulo IA



Capítulo 6. Diseño TfcGameEngine

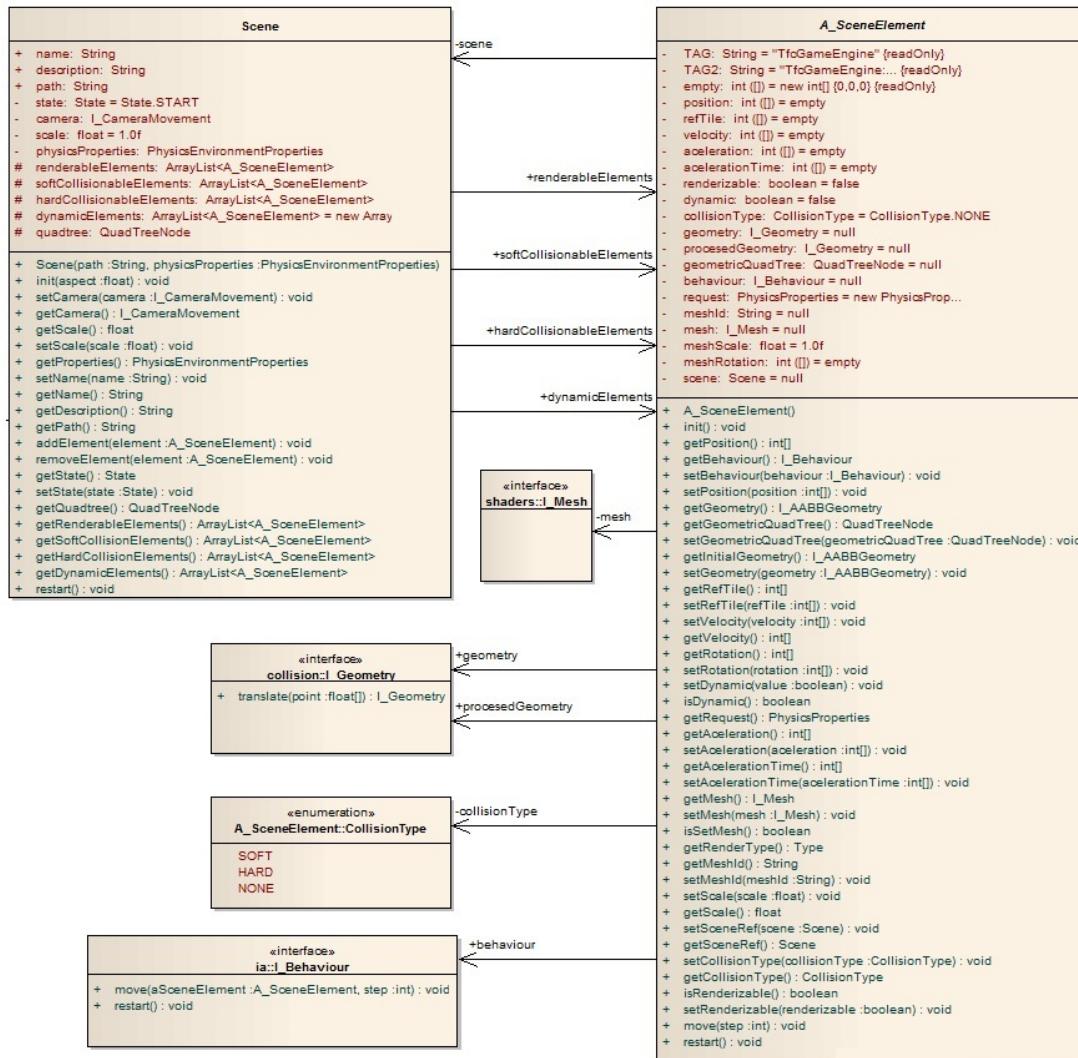


Figura 6.3: Diagrama de clases - A_SceneElement

Las cámaras

Los conceptos necesarios para comprender la implementación desarrollada han sido definidos en el capítulo 3. En este capítulo se indicaban las propiedades necesarias de las cámara para poder renderizar una escena.

Se pueden definir sobre un escenario varias cámaras, pero sólo será posible procesar la escena al mismo tiempo con una de ellas. La interfaz que implementan las distintas cámara es **I_CameraMovement** y define los siguientes métodos:

- **Move:** indica a la cámara el tiempo transcurrido, permitiendo cambiar respecto al tiempo la posición de la cámara.
- **GetCameraCenter:** se obtiene la posición actual de la cámara.
- **GetCameraEye:** se obtiene el vector de visualización de la cámara.
- **GetFovy:** se obtiene el ángulo de visión de la cámara.
- **GetzNear:** se obtiene la mínima distancia, a partir de la cual, la cámara percibe los elementos.
- **GetzFar:** se obtiene la máxima distancia hasta la cual, la cámara puede percibir los objetos.
- **SetAspect:** se establece la proporción en función del alto y ancho que ocupa el componente gráfico.
- **IsFinished:** se obtiene si el movimiento que tiene la cámara programado se ha completado.
- **restart:** coloca la cámara en la posición inicial.
- **GetViewMatrix:** devuelve la matriz de visualización que es generada a partir de la posición y el vector de visualización de la cámara. Esta matriz es aplicada a cada uno de los vértices que se van a renderizar.
- **GetPerspectiveMatrix:** devuelve la matriz de proyección, la cual es generada a partir del fovy, near, far y aspect. Del mismo modo que la matriz de visualización, es necesaria para renderizar la escena.

Se han desarrollado tres implementaciones de cámaras en la librería TfcGameEngine, en cada una de ellas se define un de los siguientes tipos de movimientos:

- **CameraZoomFunction:** esta cámara sigue una trayectoria lineal, permitiendo girar la cámara sobre la misma.
- **PersuivanCamera:** está cámara persigue un elemento del escenario.
- **ComplexCameraMovement:** esta cámara permite crear animaciones de cámara mas complejas utilizando la composición de varios movimientos de cámaras.

En el siguiente diagrama de clases UML se representan las relaciones entre las interfaz I_CameraMovement y las distintas implementaciones. También se puede apreciar que la clase ComplexCameraMovement es una agrupación de cámaras y la clase PersuivanCamera está relacionada con el elemento de la escena que persigue.



Capítulo 6. Diseño TfcGameEngine

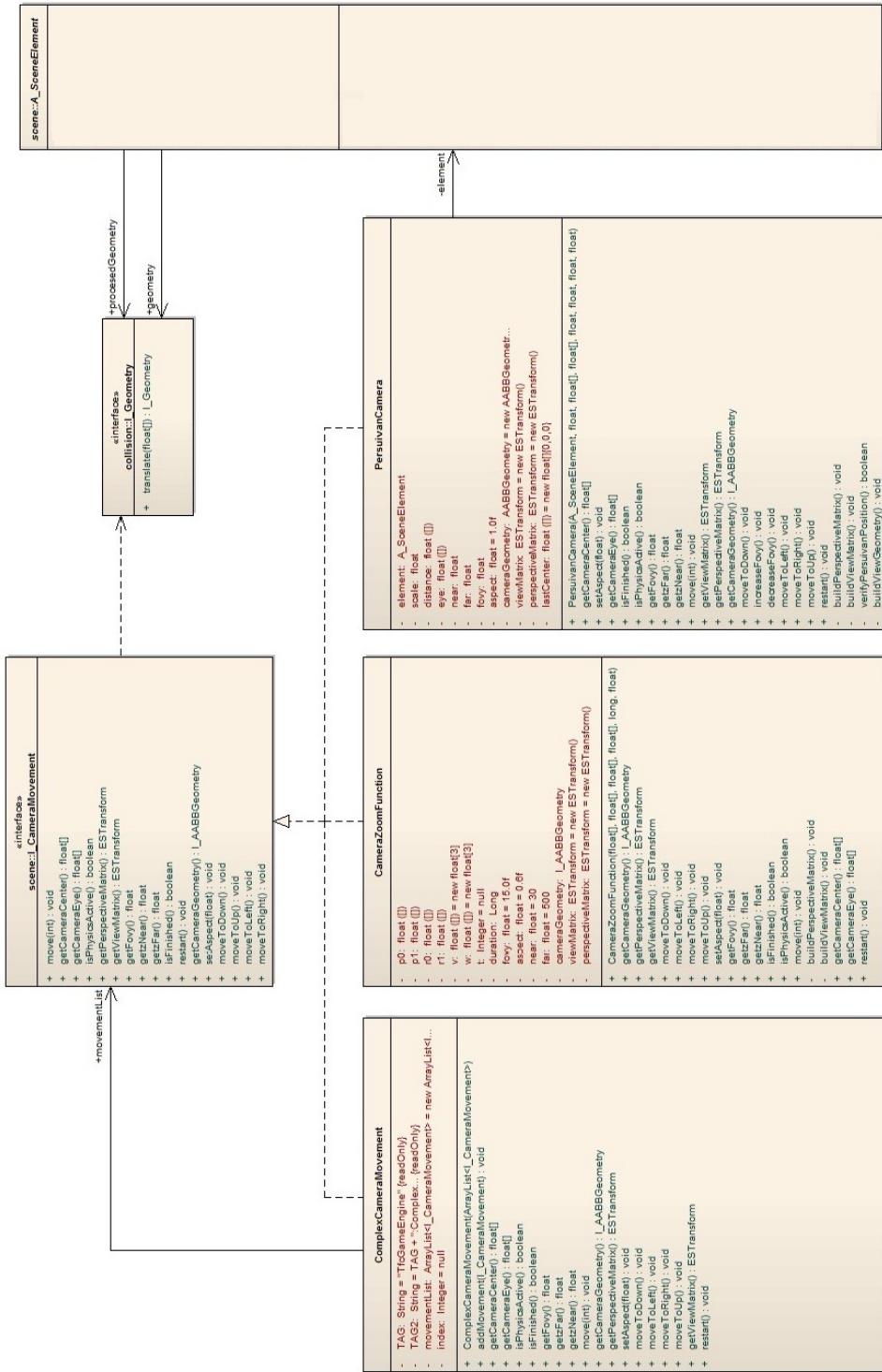


Figura 6.4: Diagrama principal de la librería TFCGameEngine

6.3. Renderer GameEngine

Una vez que han sido explicadas las clases que definen las estructuras de datos de una escena, retomamos el componente principal de la librería, la clase GameEngine.

Los métodos que ha de implementar están definidos en la interfaz Renderer y son los siguientes:

- **OnSurfaceCreated:** es invocado al crea el GLSurfaceView para su inicialización.
- **OnDrawFrame:** es invocado constantemente para re-dibujar la pantalla a través del renderer. En cada invocación es necesario conocer el tiempo transcurrido desde la última actualización de la pantalla, para calcular las nuevas posiciones de los elementos del juego, colisiones...
- **OnSurfaceChanged:** es invocado ante un cambio de tamaño del componente GLSurfaceView. Este cambio implica un tener que volver a calcular la matriz de visualización de OpenGL, la cual se ha visto afectada.

OnSurfaceCreated

La inicialización del componente desarrollado implica poner la pantalla en negro mediante el comando **glClearColor**, a continuación se activa el buffer de profundidad invocando el método **glEnable** de la librería de OpenGL. Para ello hemos de indicarle en el parámetro el valor **DEPTH_TEST**. Una vez activado el Dept Buffer, es necesario establecer su comportamiento mediante el método **glDepthFunc**. La función habitual es **GL_EQUALS**, mediante la cual se ocultarán los elementos que se encuentre detrás de otros elementos más próximos a la cámara.

La siguiente acción es inicializar, invocando el método **load**, los distintos shaders que se van a usar en el motor. La función load de los shaders se explicará más adelante en la módulo Renderer. Finalmente invocaremos el método **loadTextures** que realizará la carga en la memoria de la tarjeta gráfica las texturas que van a ser necesarias en el renderizado.

En el siguiente diagrama de secuencia en UML muestra las acciones realizadas en la inicialización del GLSurfaceView. Una curiosidad destacable radica en los parámetros que recibe la operación, los cuales proceden de OpenGL ES 1.0 pero no son necesarios en la siguiente versión, donde se accede directamente a la clase GLES20, que sigue el patrón singleton o solitario.

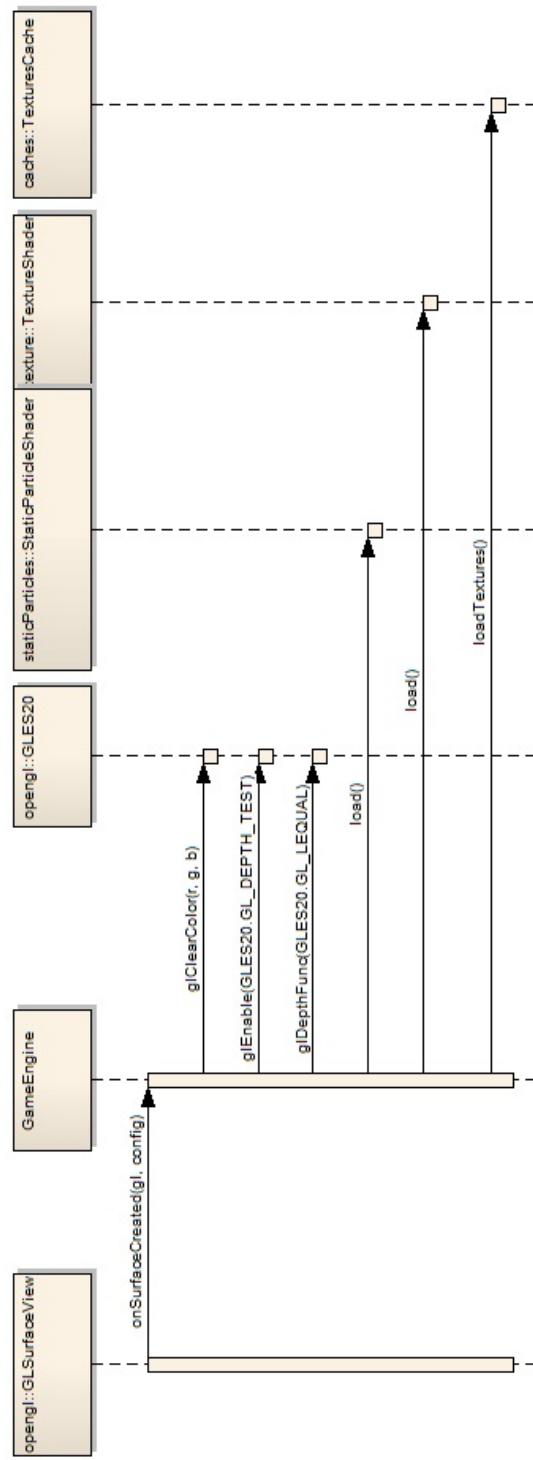


Figura 6.5: Diagrama del método OnSurfaceCreated de la clase GameEngine

6.3. Renderer GameEngine

Es importante seguir desarrollando el método loadTextures, con otro diagrama UML de secuencia, por su directa implicación con OpenGL.

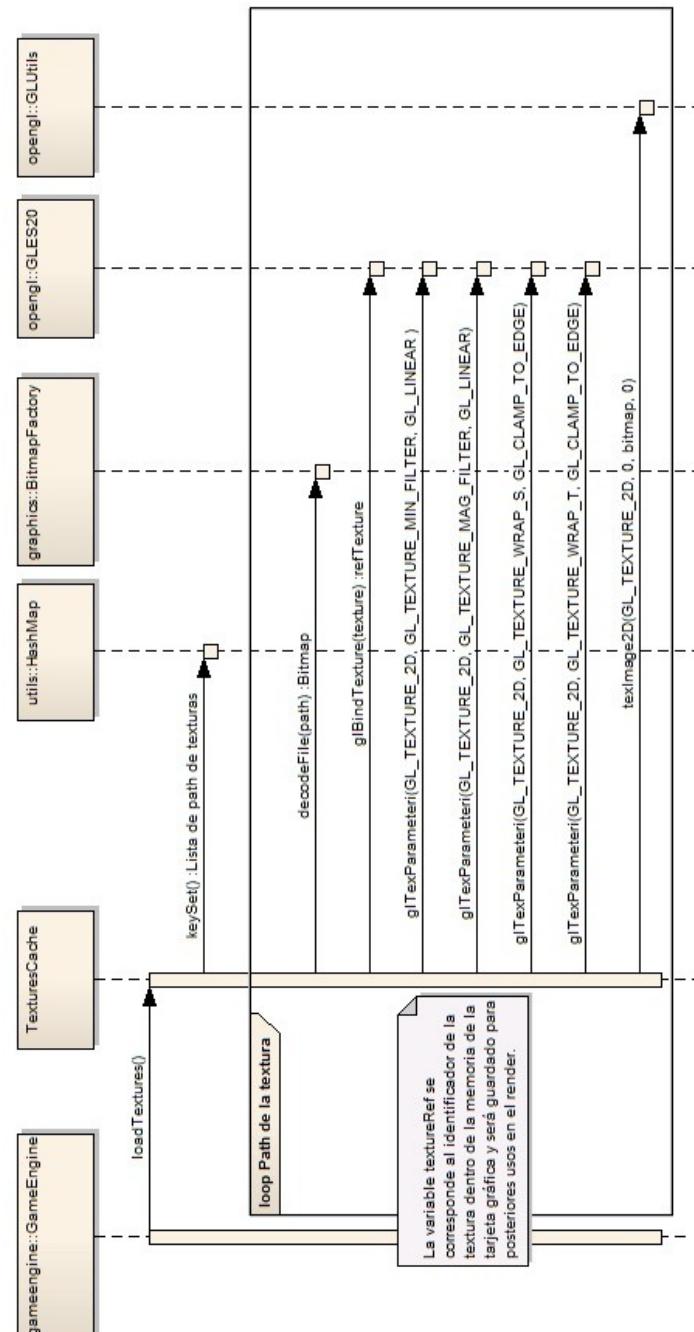


Figura 6.6: Diagrama del método loadTextures



La clase **TextureCache** contiene las rutas de los distintos ficheros que van a ser usadas en el proceso de renderizado. Todas estas imágenes han de ser cargadas previamente dentro de la inicialización del videojuego. Por cada una de ellas se han de realizar las siguientes acciones:

- Obtenemos los bytes que componen la imagen mediante el método **decodeFile**.
- Añadimos una textura a la tarjeta mediante el método **glBindTexture**, el cual nos devuelve el identificador de la textura, que se almacenará para posteriores consultas por los shader.
- Establecemos las propiedades de la textura mediante varias llamadas al método **glTexParametri**.
- Finalmente insertamos los bytes correspondientes a la imagen en la memoria de la tarjeta gráfica con ayuda de la clase **GLUtils** y el método **texImage**.

OnDrawFrame

Una vez que el componente gráfico GLSurfaceView está inicializado, comenzará a invocar continuamente al método OnDrawFrame del renderer para redibujar el espacio que ocupa. Estas invocaciones no se realizan periódicamente, ya que Android no es un sistema operativo de tiempo real, por lo tanto, se ha de calcular el tiempo transcurrido entre cada una de las iteraciones. Para obtener este valor se llamará al método **getNumberOfStep**.

A continuación, mediante el método **getDynamicElements**, se obtendrán todos los elementos del escenario que tengan la capacidad de desplazarse. No hemos de olvidar la cámara ya que tiene un comportamiento dinámico.

Por cada uno de estos elementos obtenidos, invocando al método **move** junto con el tiempo transcurrido, se calculará la nueva posición del elemento en base al algoritmo asignado para su comportamiento.

Una vez que han sido calculadas las nuevas posiciones de todos los elementos se procederá a renderizar la escena y mostrarla en pantalla mediante los shaders desarrollados. El método encargado de esta misión es **renderScene**, el cual recibe dos parámetros, la cámara con la que renderizar la escena y el tiempo transcurrido.

Para permitir incorporar funcionalidad específica en el videojuego, se incluyen los métodos **preRender** y **postRender** que se ejecutarán antes y después del renderizado.

6.3. Renderer GameEngine

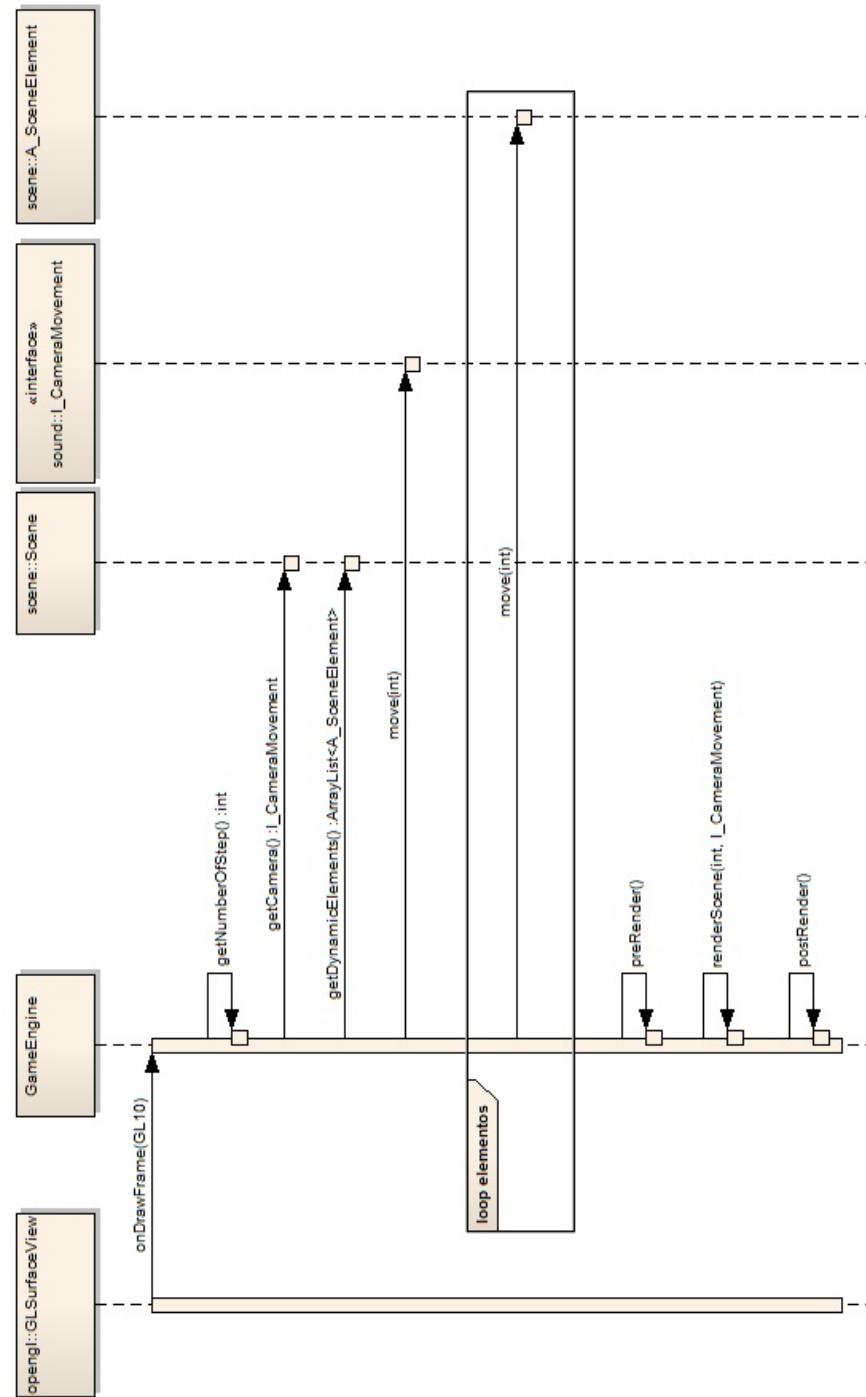


Figura 6.7: Diagrama del método `OnDrawFrame` de la clase `GameEngine`



Tras conocer el flujo principal del GameEngine, los métodos move y renderScene han pasado de soslayo debido a su complejidad. El primero de ellos, el método move, se fundamenta en el algoritmo de inteligencia artificial asignado, el cual ha de tener en cuenta las colisiones existentes. El segundo de ellos requiere conocer los shaders, por lo tanto, los métodos se explicarán en el módulo de inteligencia artificial y shaders respectivamente, tras haber comprendido el módulo de colisiones.

6.4. Módulo de colisiones

El módulo de colisiones contiene la lógica necesaria para detectar colisiones entre los objetos del escenario. Su clase principal es **CollisionEngine** y mediante los siguientes métodos detectaremos las colisiones:

- **CalculateCollisions:** Detecta las colisiones existentes entre una geometría y un listado de geometrías.
- **ExistCollision:** Detecta si existe colisión entre dos geometrías.
- **DetectCollision:** Detecta si existe colisión entre dos elementos de la escena.

Cada elemento del escenario ha de definir su forma implícita. Al trasladarse sobre el escenario, la forma procesada ha de recalcularse. Debido a esta situación surge la interfaz **I_Geometry** que contiene un único método **translate**, para generar la forma implícita procesada.

En el ámbito del proyecto se ha decidido realizar la detección de colisiones en función de formas implícitas de tipo AABB y la composición de las mismas. Como resultado de esta decisión, se han desarrollado las siguientes clases:

- **AABBGeometry:** que representa una geometría de tipo AABB.
- **ComplexGeometry:** representa una geometría compleja, compuesta por la unión de varias geometrías.

La detección de colisiones implica usar distintos algoritmos que dependen del tipo de geometría de los elementos, por este motivo, es creada la interfaz **I_CollisionDetector**, que detecta las colisiones entre dos geometrías. Las implementaciones desarrolladas son las siguientes:

- **ComplexComplexCollisionDetector:** detecta las colisiones entre geometrías de tipo ComplexGeometry.
- **AABBCollisionDetector:** detecta las colisiones entre dos geometrías de tipo AABBGeometry.

6.4. Módulo de colisiones

- **AABBComplexCollisionDetector:** detecta las colisiones entre una geometría de tipo AABBGeometry y otra de tipo ComplexGeometry.

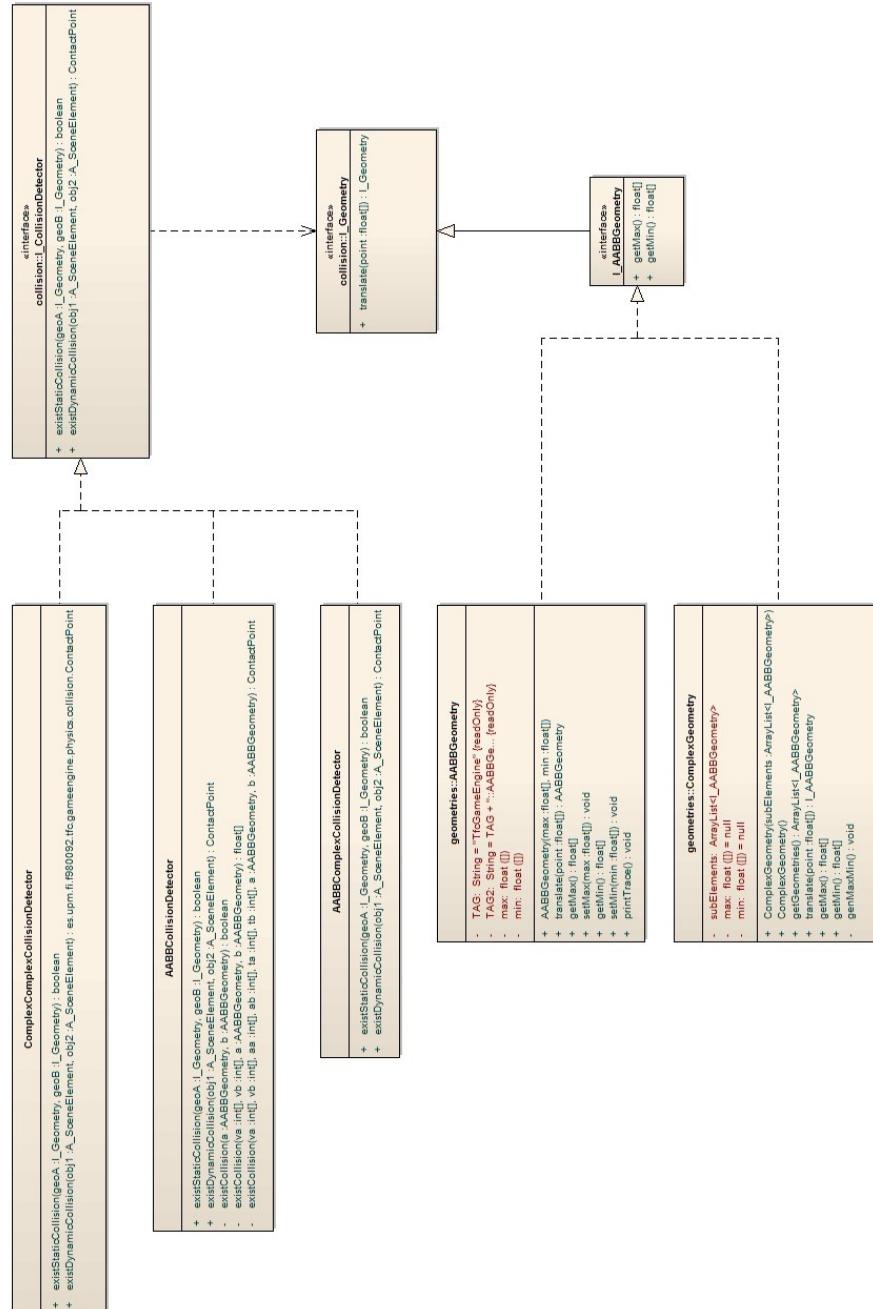


Figura 6.8: Geometrías del módulo de colisiones



Capítulo 6. Diseño TfcGameEngine

Al invocar a uno de los métodos de la clase CollisionEngine, la primera acción que se ha de realizar es detectar cuál es el I_CollisionDetector adecuado, basándose en las geometrías de los elementos. Aplicando el patrón de diseño factoría mediante la clase **CollisionDetectorFactory**. Esta clase contiene el método **getDetector** que nos devolverá cuál es el detector de colisiones indicado.

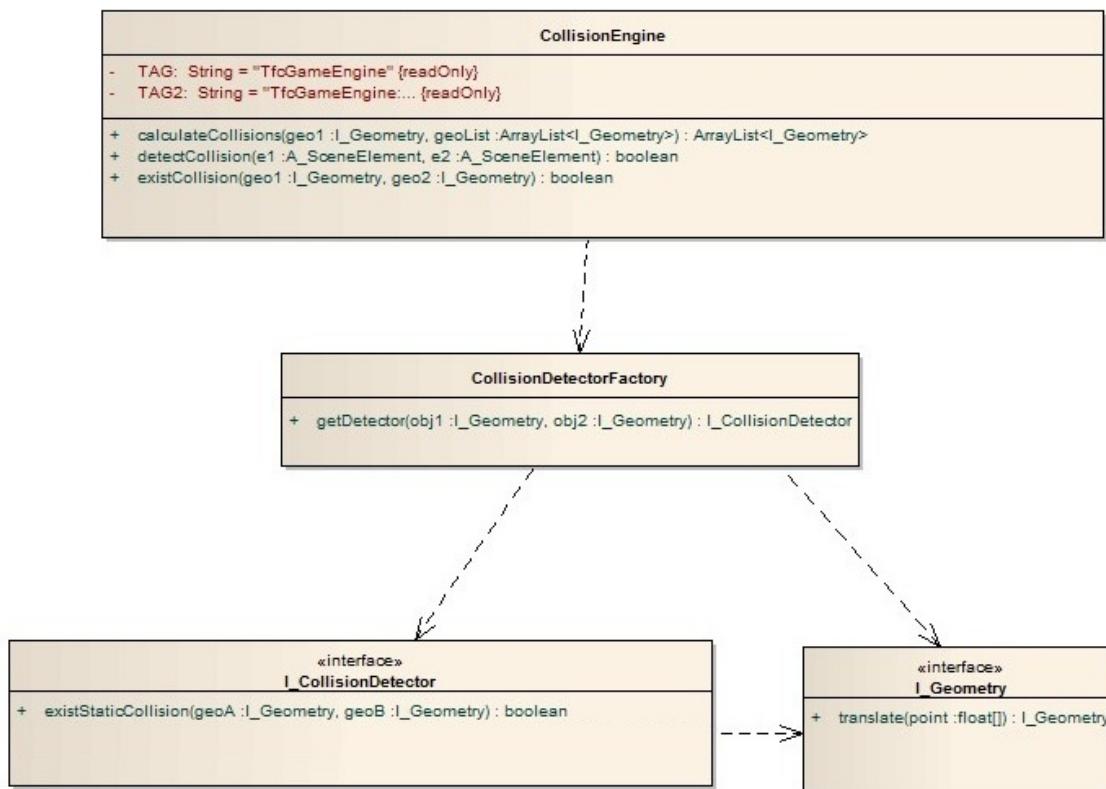


Figura 6.9: Diagrama de clases del CollisionEngine

Por último, mediante un diagrama de secuencias de UML, se explicará cuál es el funcionamiento interno del método `calculateCollisions`. En este método de la clase `CollisionEngine`, se puede apreciar la relación entre las distintas clases involucradas en la detección de colisiones.

Los otros métodos descritos en la clase `CollisionEngine` son bastante parecidos por lo que se han omitido sus diagramas de secuencia.

6.4. Módulo de colisiones

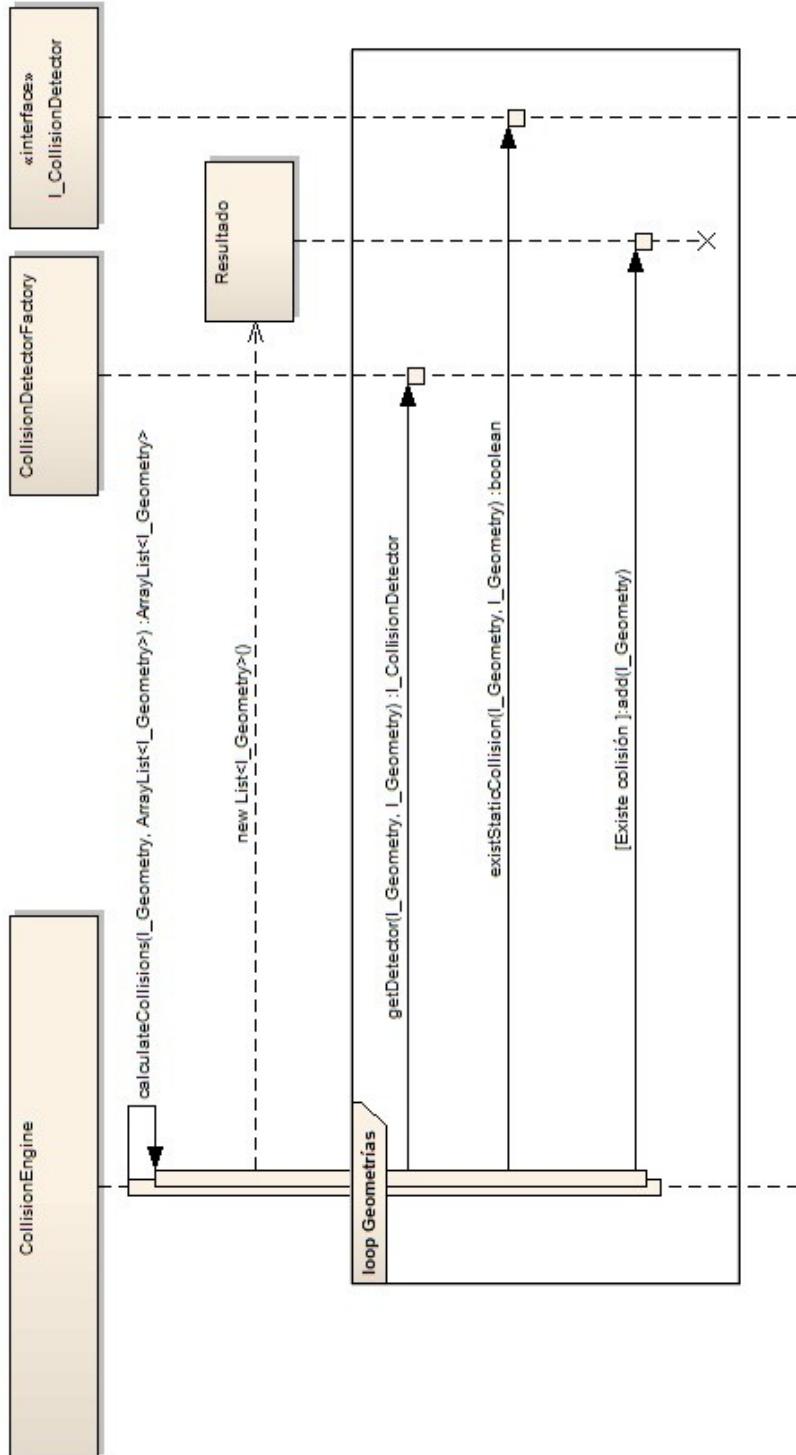


Figura 6.10: Diagrama de secuencia del calculo de colisiones



6.5. Módulo de inteligencia artificial

Una vez comprendido cómo se detectan las colisiones dentro de la librería, vamos a ver cómo se implementan los algoritmos que determinan el comportamiento de los distintos elementos de un juego.

Para definir un comportamiento dentro de la librería, se ha de implementar la clase **I_Behaviour**. Esta interfaz contiene los métodos **move** y **restart**, utilizados para obtener la nueva posición de un elemento en función del tiempo transcurrido y reiniciar el comportamiento del elemento al estado inicial.

Los algoritmos que se han programado en la librería se basan en movimiento sobre una cuadrícula. Si un elemento se desplazara durante un tiempo t , recorrería una distancia que ha sido llamada en el gráfico "distancia total".

Si el objeto se moviera en la misma dirección y sentido, obtendríamos el primer punto P_1 donde puede situarse. El segundo punto, P_2 , se corresponde con la misma dirección pero distinto sentido.

Al conocer la posición actual del elemento y la distancia total recorrida en un instante de tiempo, podemos calcular si el elemento pasa por un punto de intersección entre cuadriculas. Si esto ocurre, el elemento puede que cambie de dirección, dando lugar a dos posibles puntos adicionales P_3 y P_4 . La distancia en la dirección actual más la distancia en la nueva dirección, mostradas en el gráfico como D_1 y D_2 , será igual a la distancia total.

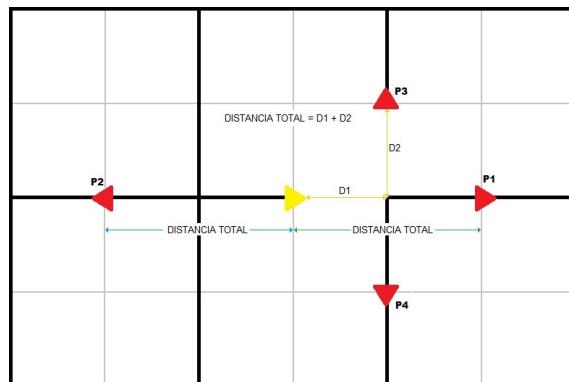


Figura 6.11: Ejemplo de movimiento sobre una cuadrícula

Antes de fijar la posición final del elemento, se ha de verificar que no existe ninguna colisión de tipo Hard. En el caso de que se produzca, el algoritmo es el responsable de resolver esta situación. Las colisiones de tipo Weak no son tratadas por la librería,

6.5. Módulo de inteligencia artificial

sino a posteriori en el método de postRender, por tener implicaciones dependientes del tipo de videojuego que se esté desarrollando.

PacmanBehaviour es la primera de las tres implementaciones de la interfaz **I_Behaviour**. Al asociar este comportamiento a un elemento, éste pasa a ser controlado de forma manual por el usuario, desplazando el dedo sobre la pantalla. Un movimiento de izquierda a derecha implicará que el elemento debe desplazarse hacia la derecha. Si el movimiento es de izquierda a derecha el elemento debe desplazarse hacia la izquierda. Del mismo modo ocurre con desplazamientos hacia arriba o hacia abajo.

Al calcular la nueva posición hay que tener en cuenta dos factores, la existencia de colisiones y si se cruza una intersección cuando se ha indicado un cambio de dirección. Si al realizar un cambio de dirección el elemento colisiona con otro elemento de tipo Hard, la posición ha de descartarse y continuar siguiendo la trayectoria actual. Si siguiendo la misma trayectoria vuelve a colisionar con un elemento de tipo Hard, el elemento se quedará quieto y junto con el elemento que colisionó.

RandomBehaviour establece un comportamiento aleatorio por la cuadrícula que compone el escenario. Al llegar a un punto de intersección en la cuadrícula, se elige al azar una nueva trayectoria, impidiendo al objeto volver sobre sus pasos o chocar con otros elementos cuyo tipo de colisión sea Hard.

SearchBehaviuor implica un comportamiento de persecución sobre otro elemento del escenario, buscando el camino más corto entre ambos con el algoritmo de Dijkstra, implementado en la clase **PathFinder**. Cuando el elemento atraviesa una intersección se invoca a la clase PathFinder para volver a calcular cuál es el camino mínimo. Con el fin de minimizar al máximo los tiempos de respuesta, son precalculados todos los posibles caminos del escenario en su carga inicial.

Para finalizar la explicación del módulo, se muestran a continuación cuatro diagramas UML. El primero de ellos contiene las clases mencionadas y sus dependencias. Los tres siguientes se corresponden con diagramas de secuencia del método move, uno por cada comportamiento implementado.



Capítulo 6. Diseño TfcGameEngine

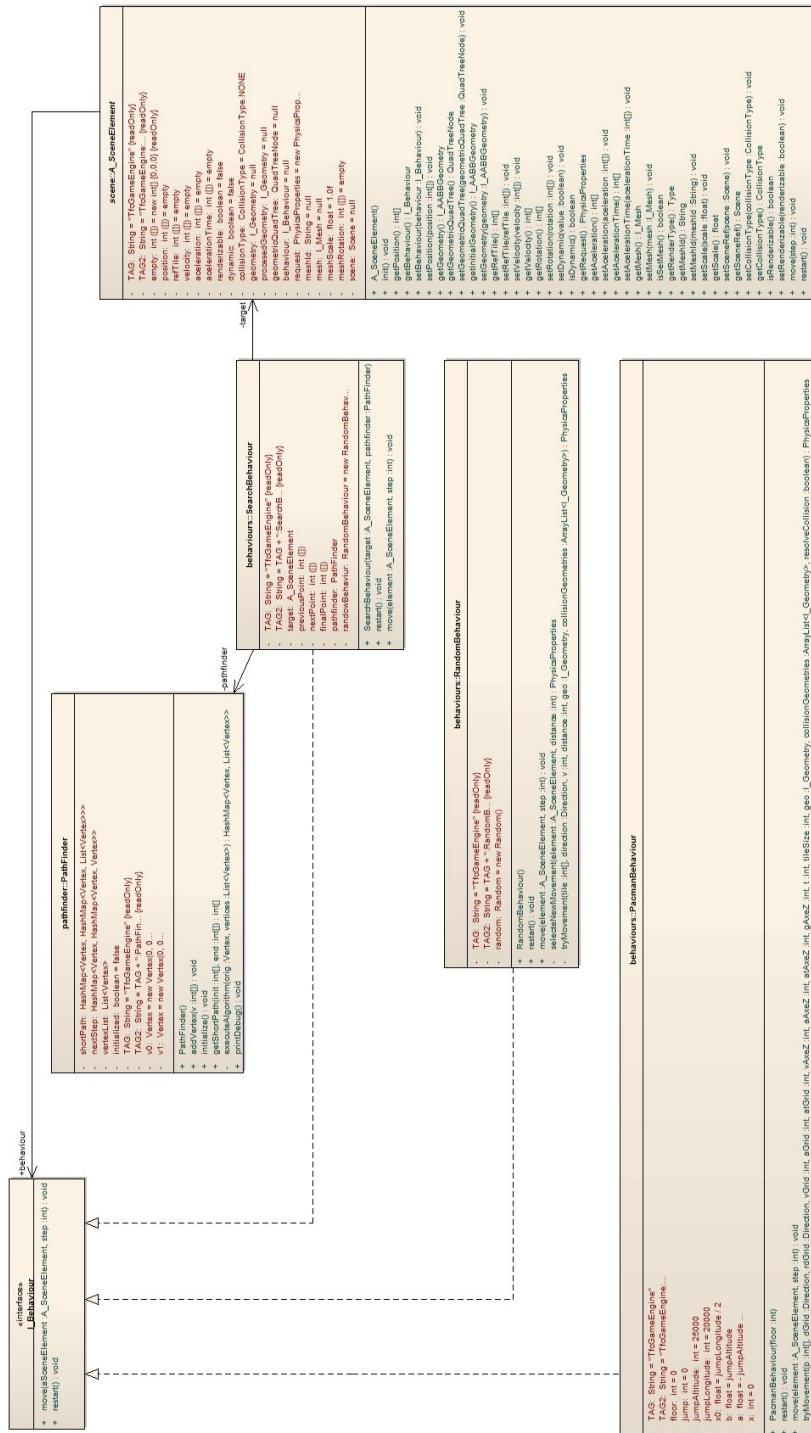


Figura 6.12: Diagrama de clases del módulo de inteligencia artificial

6.5. Módulo de inteligencia artificial

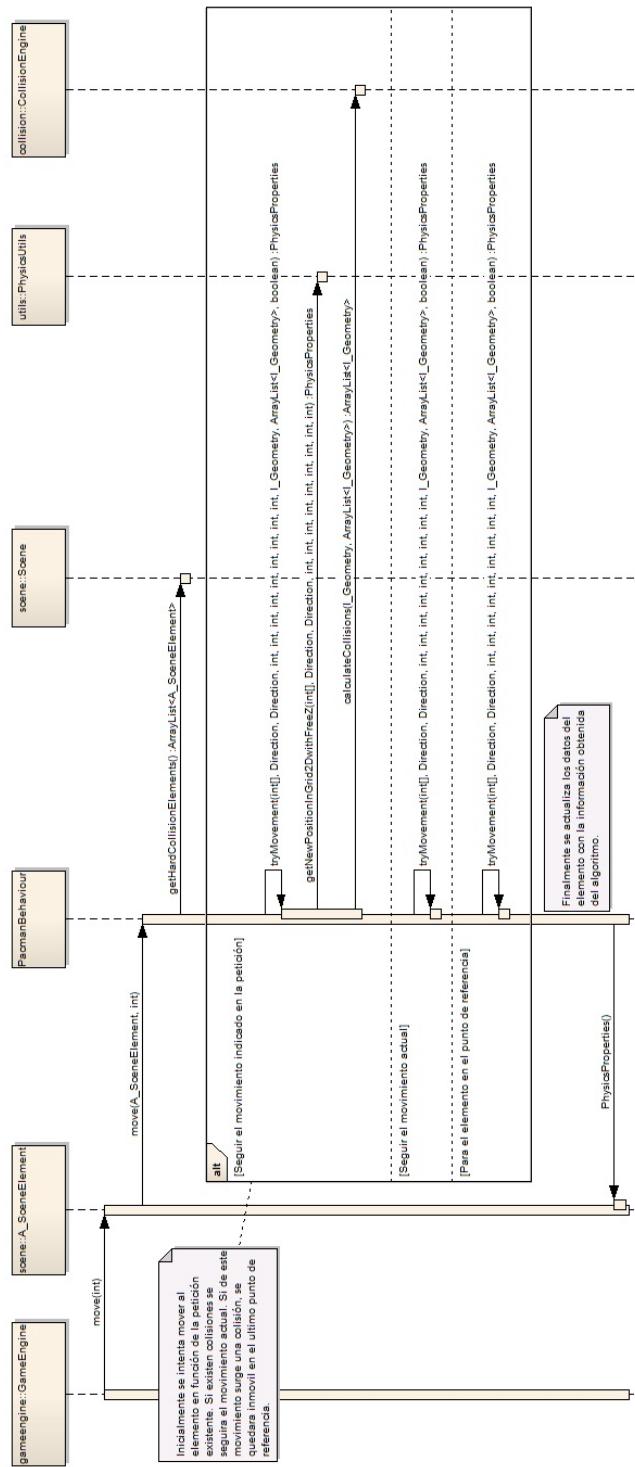


Figura 6.13: Diagrama de secuencia del método move del PacmanBehaviour

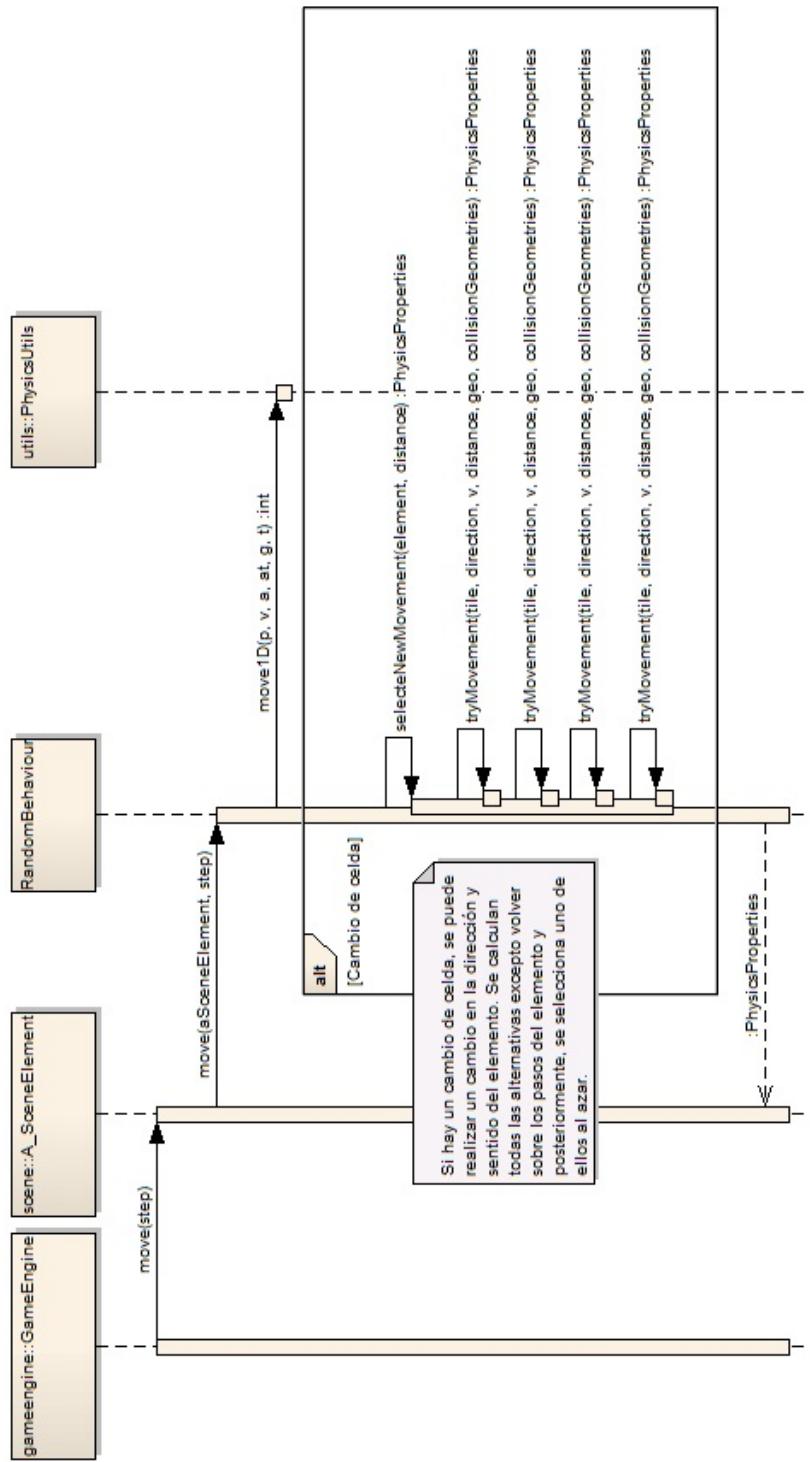


Figura 6.14: Diagrama de secuencia del método move del RandomBehaviour

6.5. Módulo de inteligencia artificial

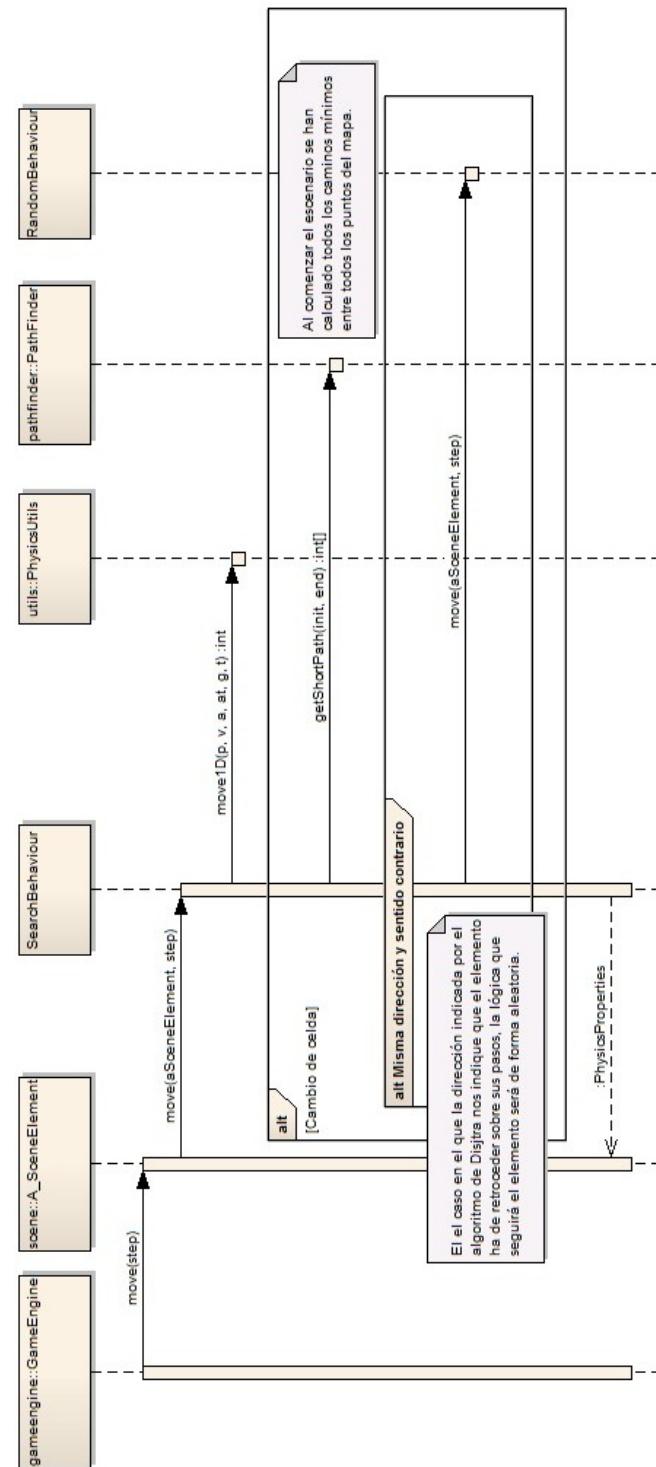


Figura 6.15: Diagrama de secuencia del método move del SearchBehaviour



6.6. Módulo de renderizado

El módulo de renderer contiene la lógica necesaria para poder realizar el proceso de renderizado, transformando la información de la escena en la imagen que se mostrará en pantalla.

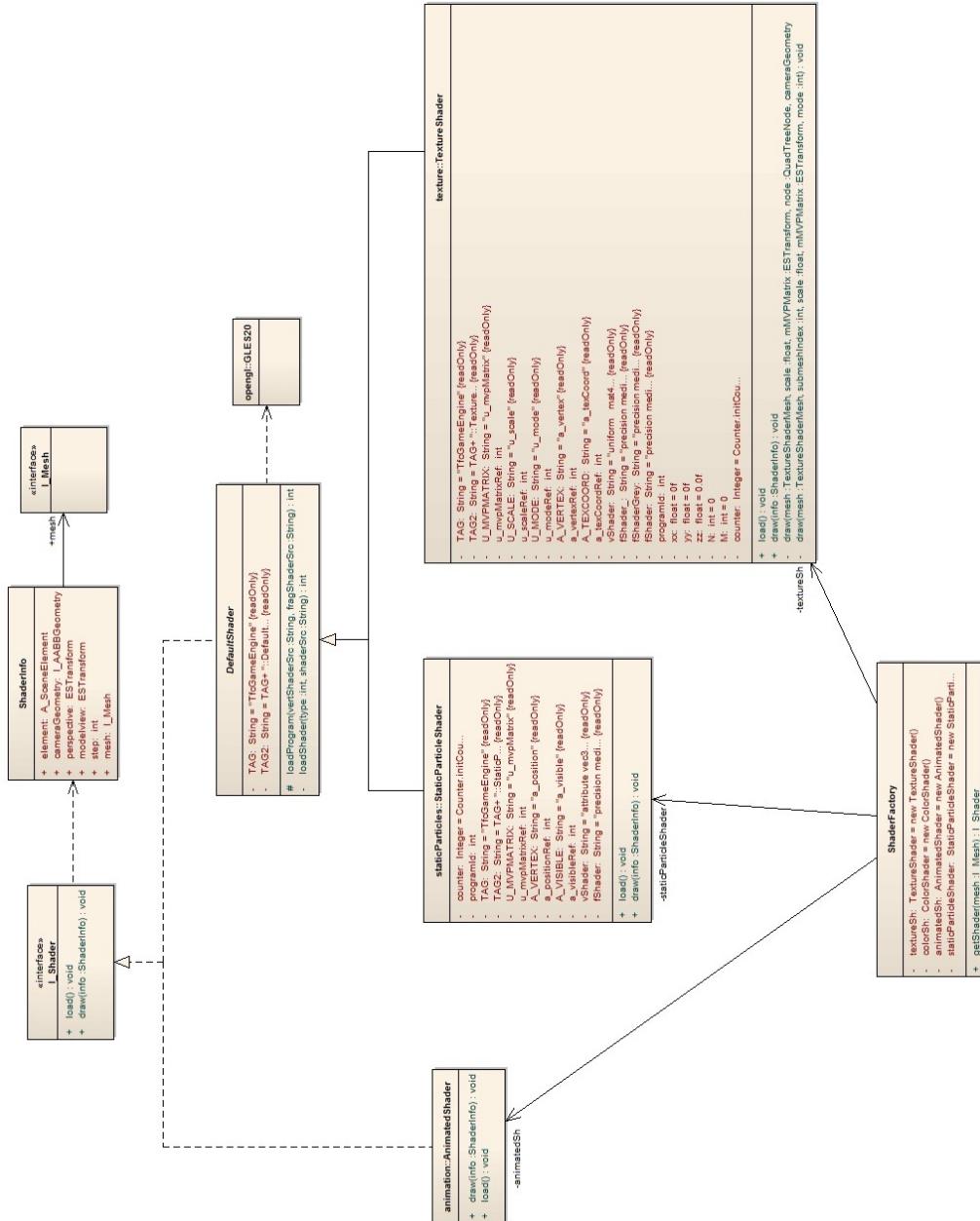


Figura 6.16: Diagrama del clases del módulo Renderer

Los elementos principales de este módulo son los shaders, los cuales se comunican con la tarjeta gráfica mediante la librería de OpenGL ES 2.0. Podemos diferenciar dos fases, las inicialización y el renderizado, que se corresponden con los métodos **load** y **draw** respectivamente, siendo estos métodos los que se definen en la interfaz **I_Shader**.

Método load

Antes de proceder a renderizar la escena, es necesario cargar los programas que vamos a usar en el proceso de render. Un programa está compuesto por un vertex shader y un fragment shader. La carga inicial del GameEngine es el lugar idóneo para cargar los distintos programas que serán utilizados en la tarjeta gráfica.

En proceso de carga de un programa es genérico para cualquier shader, por ese motivo se crea la clase **DefaultShader**, que contiene dicho método de carga, el cual tiene las siguientes fases:

1. Creación vertex shader que implica:
 - a) Indicar a la tarjeta gráfica la creación de un nuevo shader (**glCreateShader**).
 - b) Asociar el código GLSL al shader (**glShaderSource**).
 - c) Compilación del shader (**glCompileShader**).
2. Creación del fragment shader siguiendo los mismos pasos que se han realizado en el vertex shader. Al invocar dichas funciones, se identifica el tipo de vertex que se está creando.
3. Mediante el método **glCreateProgram** se indica a la tarjeta gráfica la creación de un nuevo programa, que contendrá los shaders que han sido creados previamente.
4. Se adjuntan el vertex shader y fragment shader al programa (**glAttachShader**).
5. Se indica a la tarjeta gráfica qué ha de enlazar el código fuente de los shaders que hemos adjuntado (**glLinkProgram**).
6. Por último, se libera la memoria de los shaders creados invocando el método **glDeleteShader**.

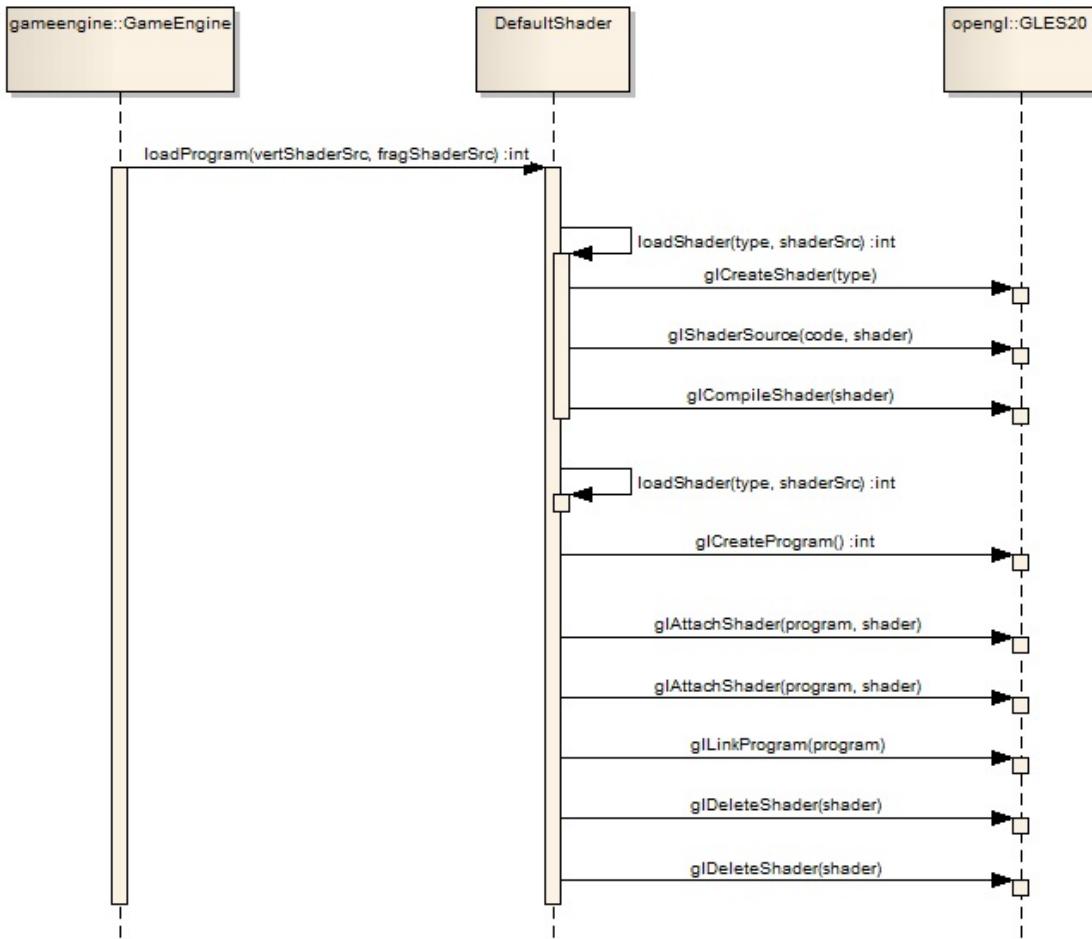


Figura 6.17: Diagrama del método load de la clase DefaultShader

Método draw

Una vez cargados en la tarjeta gráfica los programas necesarios para el renderizado, retomamos el método renderScene de la clase GameEngine, que hará uso de ellos.

En el diagrama de secuencias se explica cómo se desarrolla el proceso de renderizado. La primera acción a realizar es obtener todos aquellos elementos visibles de la escena, invocando el método **getRenderableElements**.

El proceso de render requiere realizar transformaciones matriciales sobre cada uno de los vértices de las mallas con las propiedades de la cámara. Éstas matrices son obtenidas invocando los métodos **getPerspectiveMatrix** y **getViewMatrix**.

6.6. Módulo de renderizado

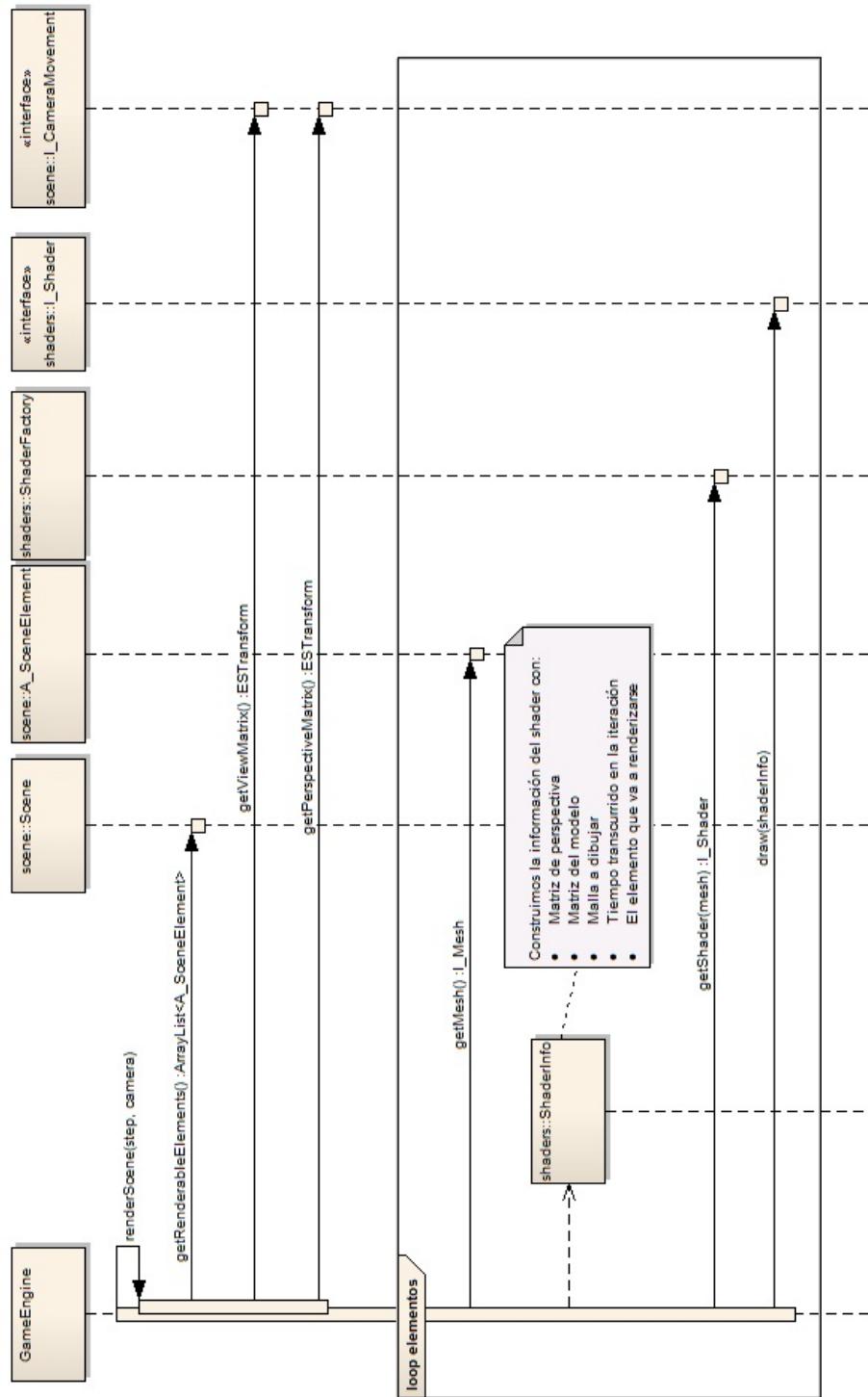


Figura 6.18: Diagrama del método RenderScene de la clase GameEngine



El siguiente paso es ir dibujando cada uno de los elementos de la escena. Cada elemento del escenario a renderizar tendrá asociada una malla, este concepto es representado por la interfaz **I_Mesh**. Cada implementación de una malla tendrá asociado un shader específico para renderizarla y todo shader debe implementar la interfaz **I_Shader**. Utilizando el patrón fábrica, a través de la clase **ShaderFactory**, desacoplamos la lógica del shader del proceso de renderizado del motor del juego, permitiendo añadir con facilidad nuevos shaders en un futuro.

El shader obtenido de la invocación al método `getShader` de la clase `ShaderFactory`, contiene el método **draw** mediante el cual se realizarán las invocaciones necesarias para renderizar la malla. Para su invocación se ha de construir una instancia de la clase **ShaderInfo**, la cual contiene la información necesaria para el renderizado y que se ha obtenido en los pasos anteriores.

Al finalizar el método `onDrawFrame`, el componente gráfico actualiza la región que ocupa en pantalla, después se dispone realizar otra iteración del bucle principal.

Tras haber detallado el comportamiento general de la fase de renderizado, el próximo paso consiste en explicar las tres implementaciones de los shaders. Por cada uno de ellos se explicará la información que gestiona, así como el código GLSL que contiene.

Texture Shader

Este shader, el más común de las implementadas, es utilizado para renderizar mallas compuestas por vértices, caras y texturas. Esta clase necesita invocar a los métodos de la librería de OpenGL ES para cargar en la memoria de la tarjeta gráfica un programa. Al extender la clase `DefaultShader`, por herencia, puede acceder a determinados métodos que facilitan éstas tareas.

La estructura de información que maneja el `TextureShader` para el renderizado está almacenada en la clase **TextureShaderMesh**, por lo tanto, al solicitar el render adecuado a la fábrica de render para una malla de tipo `TextureShaderMesh`, nos devolverá un `TextureShader`.

El método `draw` es explicado mediante un diagrama de secuencias UML. La primera acción a realizar es cargar el programa, referenciado en la clase `TextureShader`, en la tarjeta gráfica con el método **glUseProgram**.

6.6. Módulo de renderizado

Posteriormente se obtiene las matrices relacionadas con la posición de la cámara, se le aplica una traslación en función de la posición del elemento, junto con una rotación en cada eje en función de los ángulos de rotación del elemento que estamos renderizando y por último la matriz de transformación de la cámara.

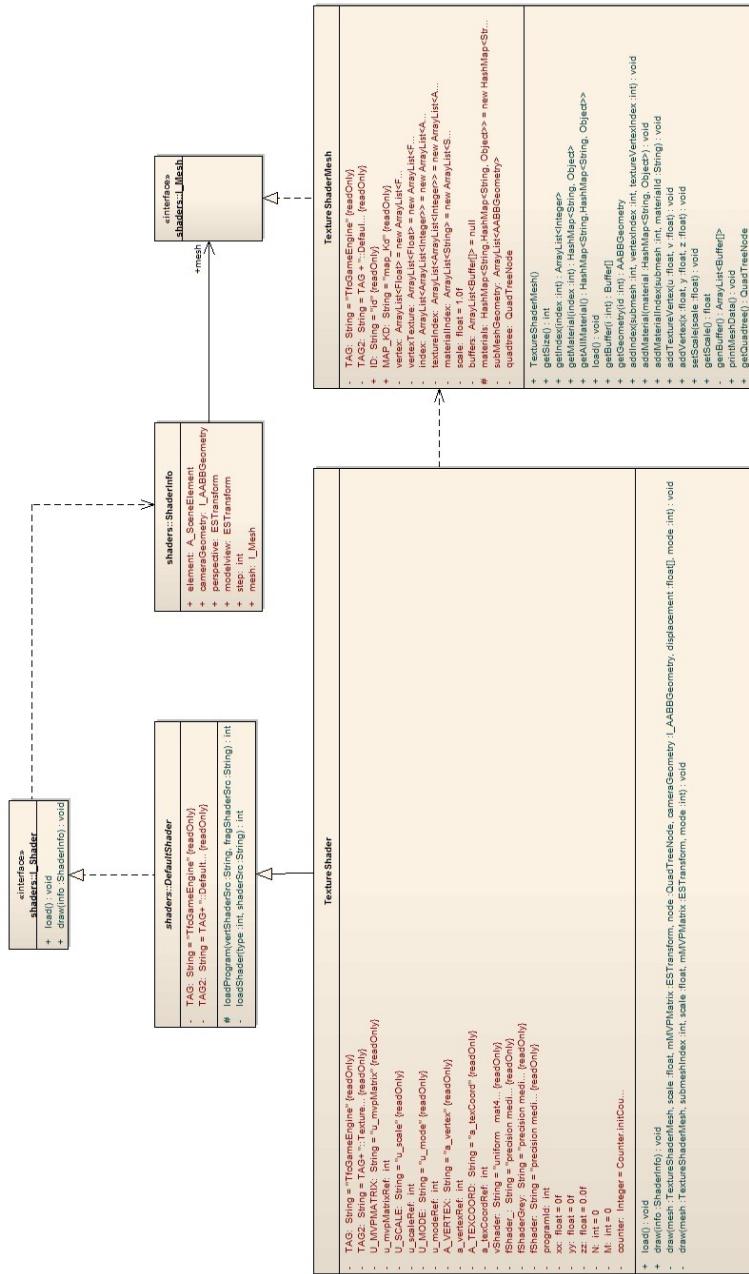


Figura 6.19: Diagrama clases del Texture Shader



Capítulo 6. Diseño TfcGameEngine

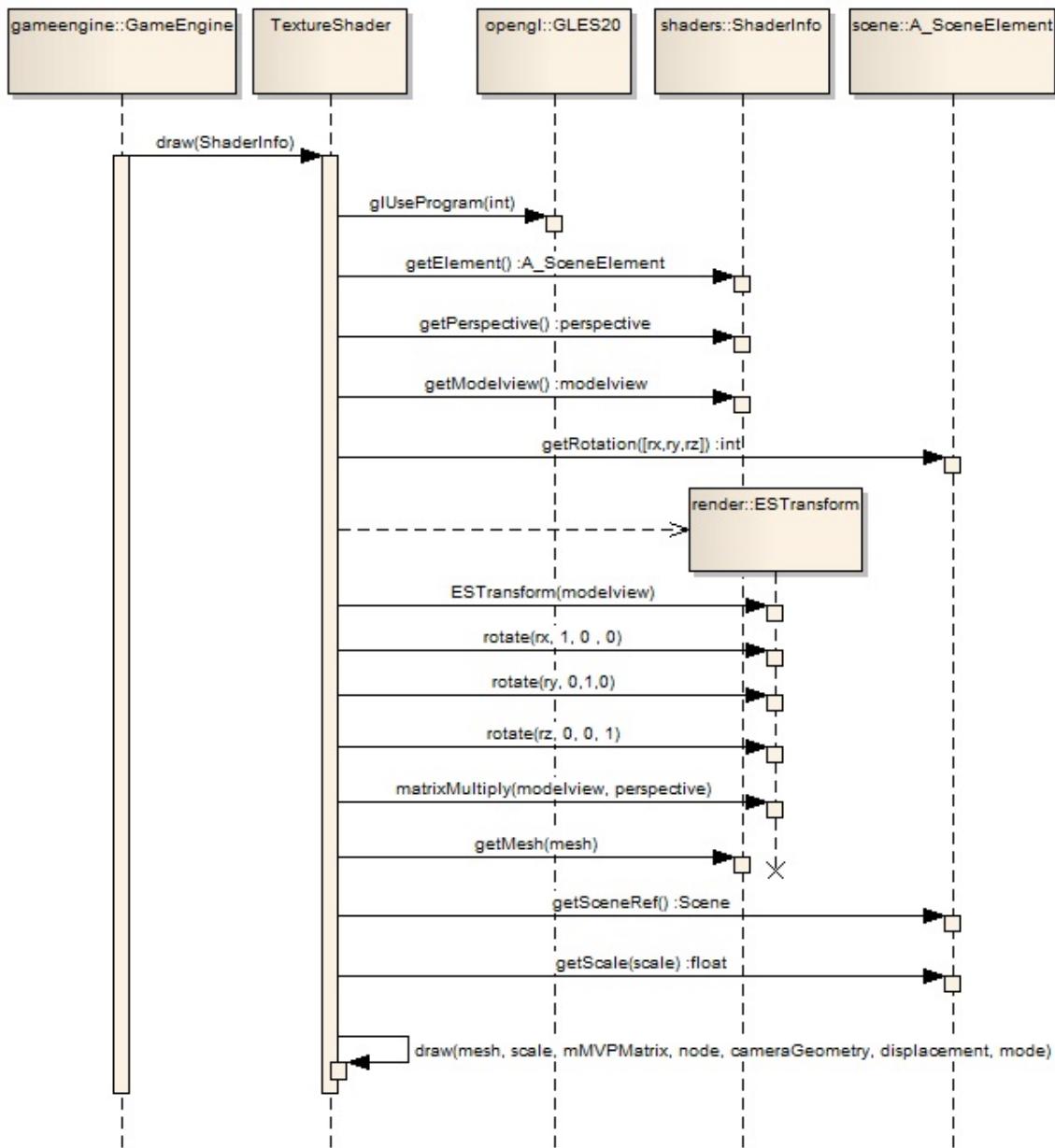


Figura 6.20: Diagrama de secuencia del método draw (Parte 1)

También se obtiene la información sobre la escala del elemento y una referencia a la escena, desde la cual, podremos saber si esta en pausa. En el caso de estar en pausa indicaremos al fragment shader que renderice las escenas en tonos sepia. Finalmente, invocamos el método privado `draw` que contiene el tratamiento de los parámetros.

6.6. Módulo de renderizado

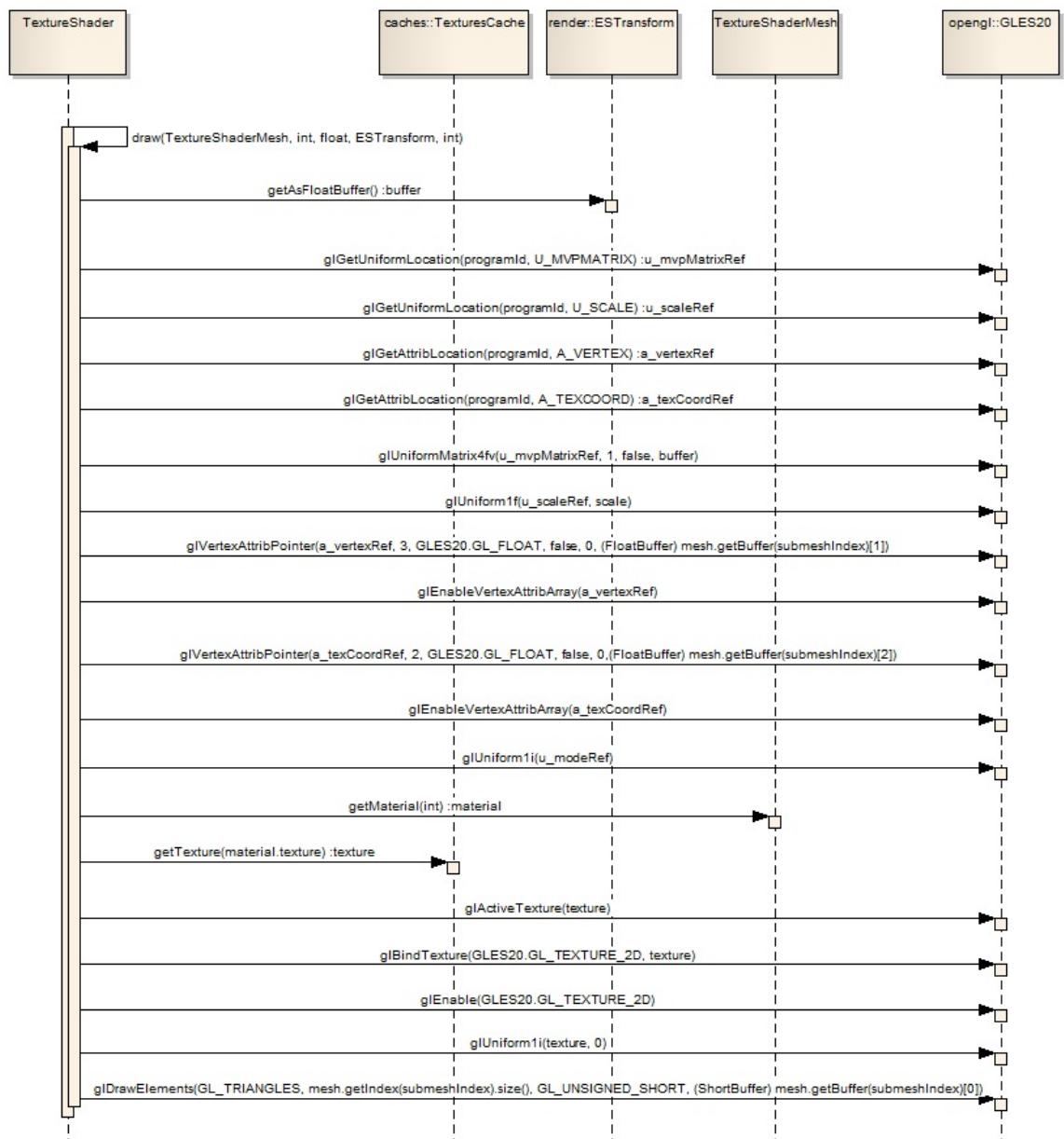


Figura 6.21: Diagrama de secuencia del método draw (Parte 2)

En este diagrama se puede observar cómo se van cargando los distintos tipos de variables en OpenGL, llamando a varios métodos dependiendo del tipo de variable y los datos que contiene.

Para poder cargar un dato en la memoria de la tarjeta gráfica, es necesario obtener sus referencias, para ello usamos los métodos con sufijo location. A partir de esta



referencia es posible cargar los datos correspondientes al elemento que estamos renderizando. No hay que olvidar la llamada al método **glEnableVertexAttribArray** para indicar que los datos introducidos se corresponden con una lista.

En el caso de las texturas, las cuales ya están insertadas en memoria, se ha de indicar la textura que vamos a manejar mediante las funciones **glActiveTexture**, **glBindTexture** y **glEnableTexture**.

Finalmente, una vez cargada en memoria toda la información, se invoca el método **glDrawElement**, responsable de dibujar cada uno de los triángulos que componen la malla del elemento.

Para completar la explicación de la clase TextureShader se detallará a continuación el código GLSL que contiene, junto con una breve explicación.

```
GLSL Vertex Shader
1 uniform    mat4 u_mvpMatrix;
2 uniform    float u_scale;
3 attribute   vec4 a_vertex;
4 attribute   vec2 a_texCoord;
5 varying     vec2 v_texCoord;
6
7 void main()
8 {
9     vec4 scale_vertex = a_vertex;
10    scale_vertex.xyz *= u_scale;
11    gl_Position = u_mvpMatrix * scale_vertex;
12    v_texCoord = a_texCoord;
13 }
```

El código del vertex shader contiene dos constantes, la matriz de transformación (**u_mvpMatrix**) y la escala (**u_scale**), que se aplicarán a todos los vértices.

La constante **u_scale** es utilizada para que la malla se ajuste a las dimensiones requeridas, de esta forma, coincidirán las proporciones en el modelo físico y gráfico.

Al aplicar la matriz de transformación **u_mvpMatrix** sobre cada vértice, este se posicionará correctamente en la escena a renderizar en función de la posición de modelo y cámara.

La arquitectura vectorial permite aplicar de forma paralela el shader, compartiendo las variables antes mencionadas. Los parámetros de entrada son la posición del

vértice (**a_vertex**) y las coordenadas para el mapeo de la textura UV (**a_texCoord**).

La salida del shader devolverá la variable **gl_Position**, que representa la posición final del vértice tras aplicar la matriz de transformación y la escala, y también la posición de la textura sin alteración alguna.

GLSL Fragment Shader

```

1 precision mediump float;
2 varying vec2 v_texCoord;
3 uniform sampler2D s_texture;
4 uniform int u_mode;
5
6 void main()
7 {
8     if (u_mode == 0)
9     {
10         gl_FragColor = texture2D( s_texture, v_texCoord );
11     }
12     else
13     {
14         float grey = dot(
15             texture2D(s_texture, v_texCoord).rgb,
16             vec3(0.299, 0.587, 0.114)
17         );
18         gl_FragColor = vec4(grey * vec3(1.2, 1.0, 0.8), 1.0);
19     }
20 }
```

El GLSL Fragment Shader tiene dos modos de funcionamiento, dependiendo de la variable **u_mode**. Si su valor es 0, mapea las texturas directamente y en otro caso modifica el color de las texturas en tonos sepia.

Para realizar el mapeo de la textura necesitamos la textura, referenciada mediante **s_texture**, y las coordenadas de mapeo que aplicar, almacenadas en **v_texCoord**.

Una vez obtenido el color rgb correspondiente a las coordenadas de mapeo sobre la textura, se aplican las operaciones pertinentes para cambiarlo a color sepia en función de la variable **u_mode**.

Static Particle Shader

Este shader ha sido creado con el objetivo de definir una serie de puntos amarillos en pantalla, en el desarrollo del videojuego veremos que es una forma de dibujar todas las pastillas del escenario con una simple invocación a la tarjeta gráfica, lo que da una



Capítulo 6. Diseño TfcGameEngine

sustancial mejora en el tiempo de proceso. De no ser así, usando el TextureShader, deberían realizarse tantas invocaciones como pastillas existieran y definir una malla común para todas ellas. De esta forma hemos pasado de una complejidad lineal a una constante.

El shader está implementado en la clase StaticParticleShader, que extiende la clase DefaultShader para acceder a las funcionalidades de OpenGL ES. La información necesaria que utiliza este shader está contenido en la clase **StaticParticleShaderMesh**, la cual contiene el listado de números decimales que, agrupados de tres en tres, se corresponden a las coordenadas de cada partícula. Por cada agrupación también se define si dicha partícula es o no visible.

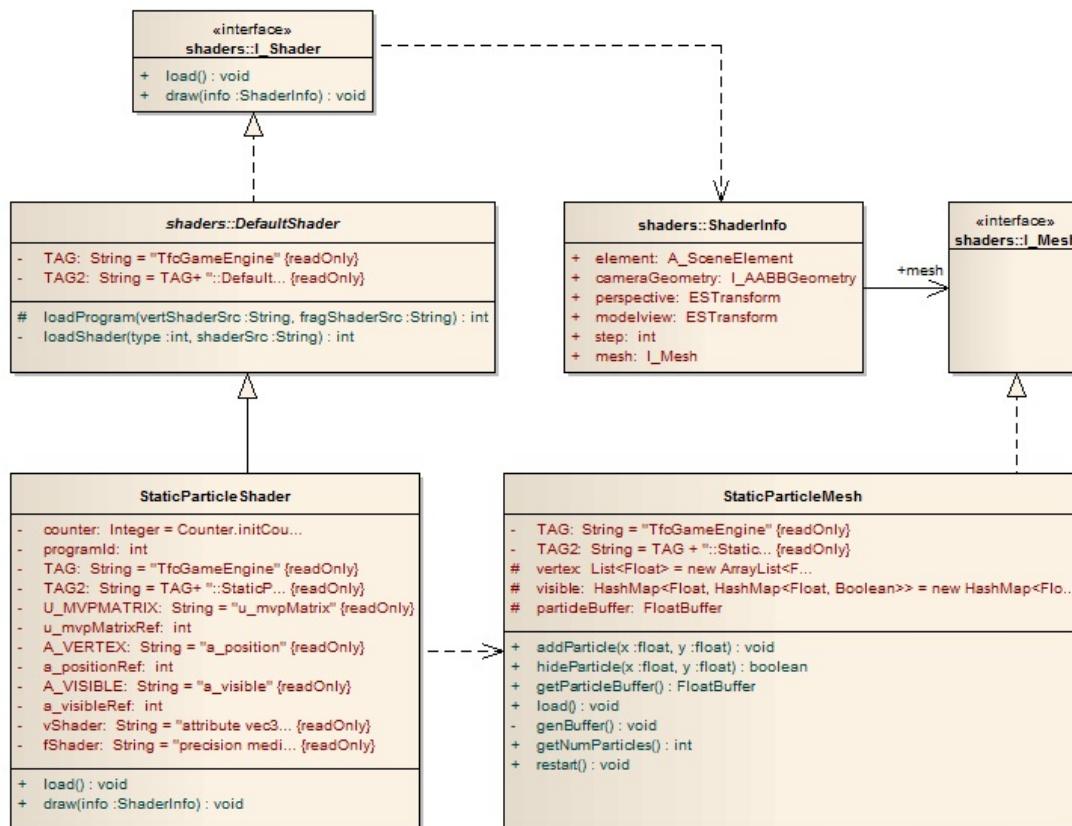


Figura 6.22: Diagrama clases del Static Particle Shader

El GLSL Vertex shader utilizado, es más simple que el anterior, tan sólo recibe como variable compartida la matriz de transformación que se aplicará a cada uno de los vértices. También se indica por cada uno de los vértices si es visible o no visible.

```
GLSL Vertex Shader
1 attribute vec3 a_position;
2 attribute float a_visible;
3 uniform mat4 u_mvpMatrix;
4 varying float v_visible;
5
6 void main()
7 {
8     gl_Position.xyz = a_position;
9     gl_Position.w = 1.0;
10    gl_Position = u_mvpMatrix * gl_Position;
11    gl_PointSize = 10.0;
12    v_visible = a_visible;
13 }
```

La salida devolverá la nueva posición de la partícula sobre la que se ha aplicado la matriz de transformación y si esta partícula ha de visualizarse o no en pantalla.

```
GLSL Vertex Shader
1 precision mediump float;
2 varying float v_visible;
3
4 void main()
5 {
6     if ( v_visible == 1.0 )
7     {
8         gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);
9         gl_FragColor.a = 1.0;
10    }
11    else
12    {
13        discard;
14    }
15 }
```

El objetivo de GLSL Fragment shader consiste en dibujar un punto de color amarillo o descartar el fragmento si la partícula no es visible. La forma de ignorar un vértice es mediante el comando **discard**.

El diagrama de secuencia que explica el método draw de este shader es muy similar al del TextureShader. La primera acción a realizar es cargar el programa correspondiente para después obtener las referencias a las variables específicas del vertex shader. A partir de estas referencias podremos cargar los valores almacenados en la clase StaticParticleShaderMesh.



Capítulo 6. Diseño TfcGameEngine

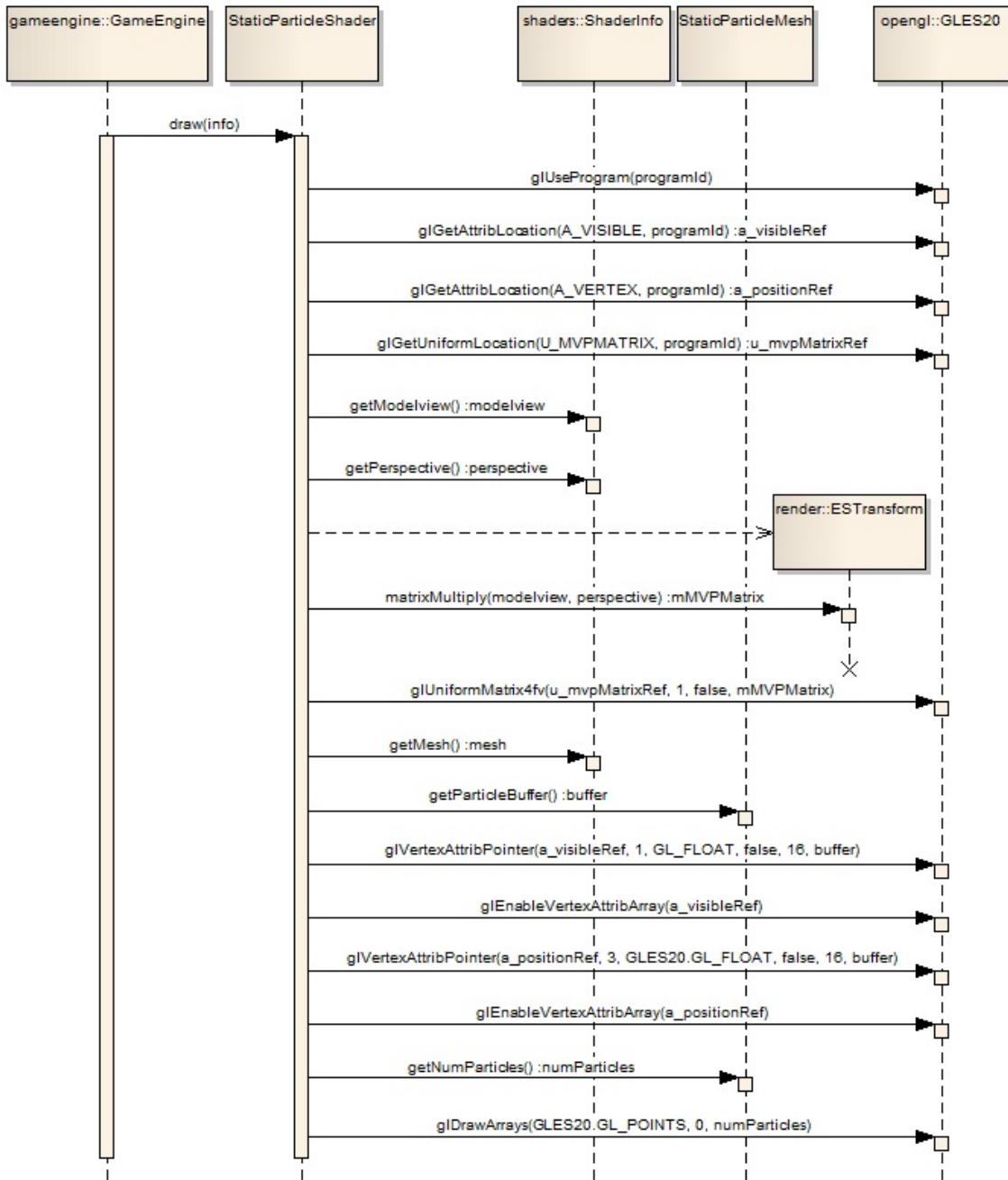


Figura 6.23: Método draw de la clase Static Particle Shader

La diferencia entre ambos shaders radica al invocar a `glDrawArrays`, en este método se indica si se van a procesar triángulos o puntos usando los valores `GL_TRIANGLES` o `GL_POINTS` respectivamente.

Animated Shader

El objetivo que persigue este shader es poder crear animaciones mediante secuencias de mallas, en vez de esqueletos. La estructura de información necesaria para crear la animación está contenida en la clase **AnimatedShaderMesh**. Sus propiedades son las siguientes:

- Meshes: contiene un listado de mallas que componen la animación.
- CurrentFrame: indica cuál es la malla que se está visualizando en ese momento.
- Durations: listado que indica la duración de cada una de las mallas.
- FrameTime: tiempo que ha de seguir visualizándose la malla actual.

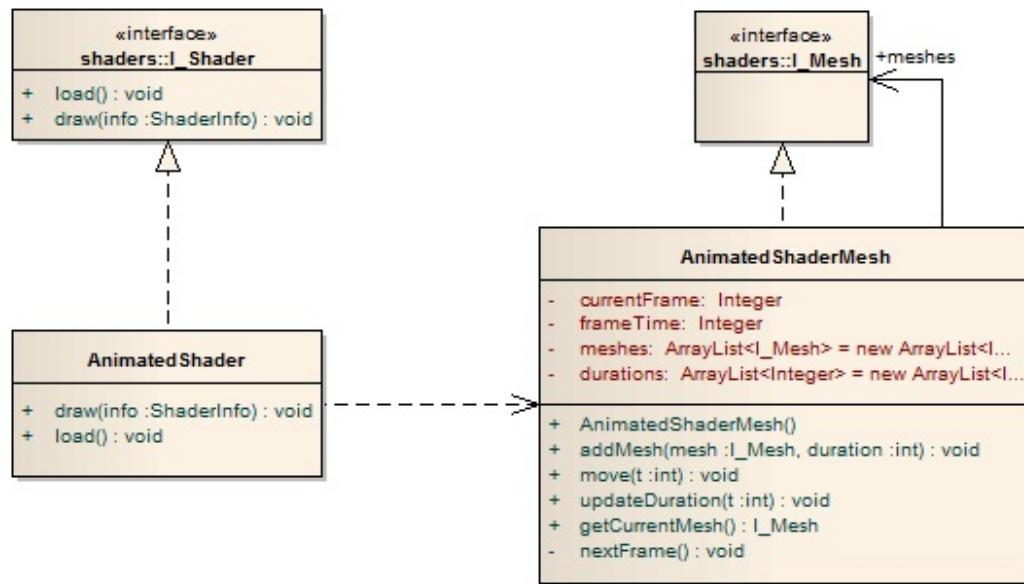


Figura 6.24: Diagrama clases del Animated Shader

Como podemos apreciar en el diagrama de secuencias, a pesar de haber denominado a esta clase como un shader, no tiene lógica de acceso a OpenGL ES, ya que es delegada a cada una de las mallas almacenadas en la propiedad meshes.

Al invocar al método move de la clase AnimatedShader, se ha de restar la duración del frame al frameTime. Si el nuevo valor de la propiedad frametime es menor o igual a cero, el tiempo que debe mostrarse la malla actual se ha agotado, por lo tanto, el currentFrame apuntará a la siguiente de la lista. En el caso de ser la última malla, se volverá a renderizar la primera malla del listado.

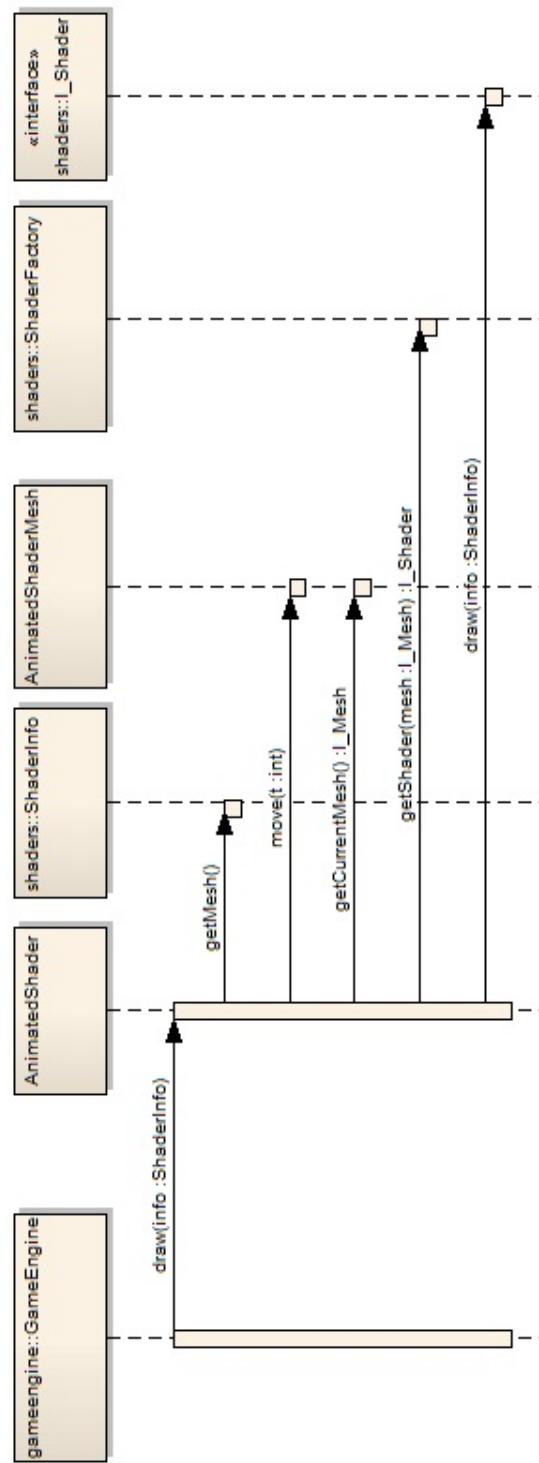


Figura 6.25: Método draw del AnimatedShader

Lector de mallas en ficheros **wavefront**

Tal y como hemos visto en la estructura de datos que maneja el TextureShader, se han de indicar todos los vértices, caras, vértices de textura y las distintas texturas que componen la malla. Definir toda esta información de forma manual es inviable debido a la cantidad de datos necesarios para definir cada una de las mallas.

Gracias a la existencia de aplicaciones de diseño gráfico como son 3DStudio, Blender, Maya... podemos crear mallas en ellos y exportarlas a un fichero, el cual, podremos leer mediante la clase `TextureShaderMeshReader` desarrollada en la librería del proyecto.

Al examinar cada uno de los programas de diseño gráfico, hemos visto que debido a su complejidad, deciden crear cada una su propio tipo de fichero, en formato binario, que se adapte a las necesidades concretas de cada uno de los fabricantes. Afortunadamente existen varios formatos estándar de texto plano para definir mallas y que son soportados, de mejor o peor manera, por estas aplicaciones.

En el ámbito del proyecto nos vamos a centrar en Blender por ser una herramienta ya consolidada y gratuita que permite exportar a distintos formatos. Tras evaluar cada uno de ellos, se ha decidido elegir debido a su sencillez el formato Wavefront que esta compuesto por :

- Un fichero `.obj` que contiene información sobre la estructura de la malla.
- Uno o varios ficheros `.mtl` que contiene información sobre los materiales que han sido aplicados en el fichero `obj`.

Dentro de la sencillez de este formato en comparación con otros, pero debido a la complejidad de la materia que estamos tratando, sólo vamos a prestar atención a un subconjunto de todas las instrucciones definidas en su especificación⁴, las necesarias, para poder generar instancias de la clase `TextureShaderMesh`.

Formato `.obj`

La forma más clara de entender cómo esta compuesto un fichero `.obj` es mediante ejemplos. Se expondrán cuatro pequeños ejemplos donde se irán incorporando nuevos conceptos de forma incremental.

En el primer ejemplo se va a definir un único triángulo, definiendo sus vértices y la única cara que lo compone.

.....

⁴Para una mayor compresión de los fichero `.obj` y `.mtl` de WaveFront es recomendable las especificaciones en http://netghost.narod.ru/gff/vendspec/waveobj/obj_spec.txt



Ejemplo 1

```
1 # Fichero Obj
2 v 0.0 0.0 0.0
3 v 0.0 1.0 0.0
4 v 1.0 0.0 0.0
5 f 1 2 3
```

La primera línea de ellas se corresponde con un comentario, las tres siguientes definen cada uno de los vértices que compondrán el triángulo.

Un vértice se especifica mediante una **v** seguida de las coordenadas $[x, y, z]$ que lo definen. El identificador de cada vértice se corresponde con su orden de creación, en el ejemplo el primer vértice v_1 se corresponde con las coordenadas $[0,0,0,0,0,0]$, el segundo $v_2 = [0,0,1,0,0,0]$ y el tercero con $v_3 = [1,0,0,0,0,0]$.

Para definir el triángulo, o lo que es lo mismo, un polígono de tres vértices, se utiliza la letra **f** junto con los índices de los vértices que lo forman, como podemos ver en la última línea del primer ejemplo.

En el siguiente ejemplo, la malla a crear será un cubo, para lo cual usaremos ocho vértices y seis caras definidas mediante polígonos de cuatro vértices.⁵.

Ejemplo 2

```
1 v 0.000000 2.000000 2.000000
2 v 0.000000 0.000000 2.000000
3 v 2.000000 0.000000 2.000000
4 v 2.000000 2.000000 2.000000
5 v 0.000000 2.000000 0.000000
6 v 0.000000 0.000000 0.000000
7 v 2.000000 0.000000 0.000000
8 v 2.000000 2.000000 0.000000
9 f 1 2 3 4
10 f 8 7 6 5
11 f 4 3 7 8
12 f 5 1 4 8
13 f 5 6 2 1
14 f 2 6 7 3
```

Ahora que somos capaces de definir la mallas, de mayor o menor complejidad según aumentemos el número de polígonos que lo componen, vamos a añadir materiales a cada una de las caras del cubo del ejemplo anterior.

.....
⁵Al exportar los ficheros de Blender a ficheros Wavefront se ha indicar que la malla está compuesta por triángulos debido a que OpenGL ES no soporta la creación de mallas basadas en cuadrados

Ejemplo 3

```
1 mtl lib master.mtl
2 v 0.000000 2.000000 2.000000
3 v 0.000000 0.000000 2.000000
4 v 2.000000 0.000000 2.000000
5 v 2.000000 2.000000 2.000000
6 v 0.000000 2.000000 0.000000
7 v 0.000000 0.000000 0.000000
8 v 2.000000 0.000000 0.000000
9 v 2.000000 2.000000 0.000000
10 usemtl red
11 f 1 2 3 4
12 usemtl blue
13 f 8 7 6 5
14 usemtl green
15 f 4 3 7 8
16 usemtl gold
17 f 5 1 4 8
18 usemtl orange
19 f 5 6 2 1
20 usemtl purple
21 f 2 6 7 3
```

Lo primero que hay que resaltar, en la primera línea, es la referencia a un fichero en formato .mtl mediante el comando `mtllib`, seguido de la ruta del fichero.

Justo antes de definir cada una de las caras del cubo, podemos ver cómo se asigna un material mediante el comando `usemtl` seguido del nombre del material. Este nombre tiene que estar definido dentro del fichero .mtl que hemos enlazado en la primera línea.

Siguiendo el código del ejemplo, el cubo generado tendrá cada cara de distinto color en función de cómo se haya definido en el fichero .mtl los material red, blue...

Ejemplo 4

```
1 mtl lib master.mtl
2 v 0.000000 2.000000 0.000000
3 v 0.000000 0.000000 0.000000
4 v 2.000000 0.000000 0.000000
5 v 2.000000 2.000000 0.000000
6 vt 0.000000 1.000000
7 vt 0.000000 0.000000
8 vt 1.000000 0.000000
9 vt 1.000000 1.000000
10 usemtl wood
11 f 1/1 2/2 3/3 4/4
```



Como ejemplo final, vamos a crear un cuadrado con la información necesaria para el mapeo uv de la textura, conocida como vertex texture. La estructura es exactamente la misma que la de un vértice pero en vez de ser el comando v se usa vt.

La forma de aplicar los mapeos de la textura a cada una de las caras del polígono es indicándola de la siguiente manera $f \nu_x/vt_x \nu_y/vt_y \nu_z/vt_z$.

Con la sintaxis explicada ha sido posible definir todas las mallas requeridas en el ámbito del proyecto. Ampliando el lector y los shader sería posible generar otros elementos como puntos, líneas, curvas y superficies.

Una dificultad añadida del proyecto al generar este tipo de fichero mediante Blender u otra herramientas de diseño 3D, consiste en conocer cómo se traducen al formato Wavefront las acciones realizadas en la aplicación y saber si están permitidas dentro del lector desarrollado. De no ser así, podríamos generar mallas que no se procesarán correctamente.

Formato .mtl

Hemos visto que desde el fichero .obj se enlaza con el fichero de .mtl que contiene distintos materiales. Por cada uno de los materiales se puede indicar⁶:

- Nombre del material (newmtl)
- Color de ambiente (Ka)
- Color de difusión (Kd)
- Color especular (Ks)
- Modelo de iluminación (ilum)
- Nivel de transparencia del elemento (d)
- Textura para la luz de ambiente (map Ka)
- Textura para la luz de difusión (map Kd)
- Textura para la luz especular (map ks)

⁶En el proyecto sólo se definen las propiedades principales de una material, para una mayor detalle consultar su especificación en <http://www.fileformat.info/format/material/>

6.6. Módulo de renderizado

Ejemplo 1: Material con una tonalidad verde

```
1 newmtl diss_green
2 Ka 0.0000 1.0000 0.0000
3 Kd 0.0000 1.0000 0.0000
4 d 0.8000
5 illum 1
```

En el siguiente ejemplo usamos una imagen (logo.jpg) la cual añadiremos al material en tono verdoso.

Ejemplo 2: Material con una textura

```
1 newmtl logoVerde
2 Ka 0.0000 0.2000 0.0000
3 Kd 0.0000 0.8000 0.0000
4 illum 1
5 map_Ka logo.jpg
6 map_Kd logo.jpg
```

Toda esta información es procesada por un motor gráfico de Wavefronten en cuyo modelo de iluminación dan cabida. En el ámbito del proyecto los materiales se han simplificado a texturas, por lo tanto sólo será necesario utilizar el parámetro `newmtl`, como identificador del material y `map_Kd` para obtener la imagen correspondiente a la la textura, el resto de la información se ignora.

Para finalizar el capítulo, vamos a mostrar el fichero .mtl resultante de exportar desde Blender al formato Wavefront uno de los escenarios desarrollados en el videojuego realizado en el proyecto.

Ejemplo 3

```
1 # Blender MTL File: 'SandboxLand.blend'
2 # Material Count: 3
3 newmtl Material_Azul_InternalWallTexture
4 Ns 96.078431
5 Ka 0.000000 0.000000 0.000000
6 Kd 0.003194 0.028797 0.640000
7 Ks 0.500000 0.500000 0.500000
8 Ni 1.000000
9 d 0.700000
10 illum 2
11 map_Kd InternalWallTexture.png
12
13 newmtl Material_Floor
14 Ns 96.078431
15 Ka 0.000000 0.000000 0.000000
16 Kd 0.640000 0.640000 0.640000
```



Capítulo 6. Diseño TfcGameEngine

```
17 Ks 0.500000 0.500000 0.500000
18 Ni 1.000000
19 d 1.000000
20 illum 2
21 map_Kd Floor.png
22
23 newmtl Material_Gris_ExternalWallTexture
24 Ns 96.078431
25 Ka 0.000000 0.000000 0.000000
26 Kd 0.640000 0.383237 0.000000
27 Ks 0.500000 0.500000 0.500000
28 Ni 1.000000
29 d 1.000000
30 illum 2
31 map_Kd ExternalWallTexture.png
```

CAPÍTULO

7

Diseño de Pacmania

En éste último capítulo de proyecto, se da una visión general de las actividades que componen el videojuego para tener una visión global de su diseño. Después se explica en profundidad el contenido del directorio que contiene los ficheros de configuración necesarios en el videojuego, especificando su formato y cómo han sido creados. Por último, se comentarán cada una de las actividades de la aplicación, completado de ésta forma la explicación del diseño del videojuego.



7.1. Visión General

Toda aplicación realizada en Android está compuesta por las actividades, por lo que la forma más coherente de explicar cómo se ha desarrollado, es explicando las actividades que lo componen y la relación existente entre las mismas. Una forma de verlo de forma esquemática es mediante el diagrama de navegación entre actividades, junto con una breve descripción de cada una de ellas.

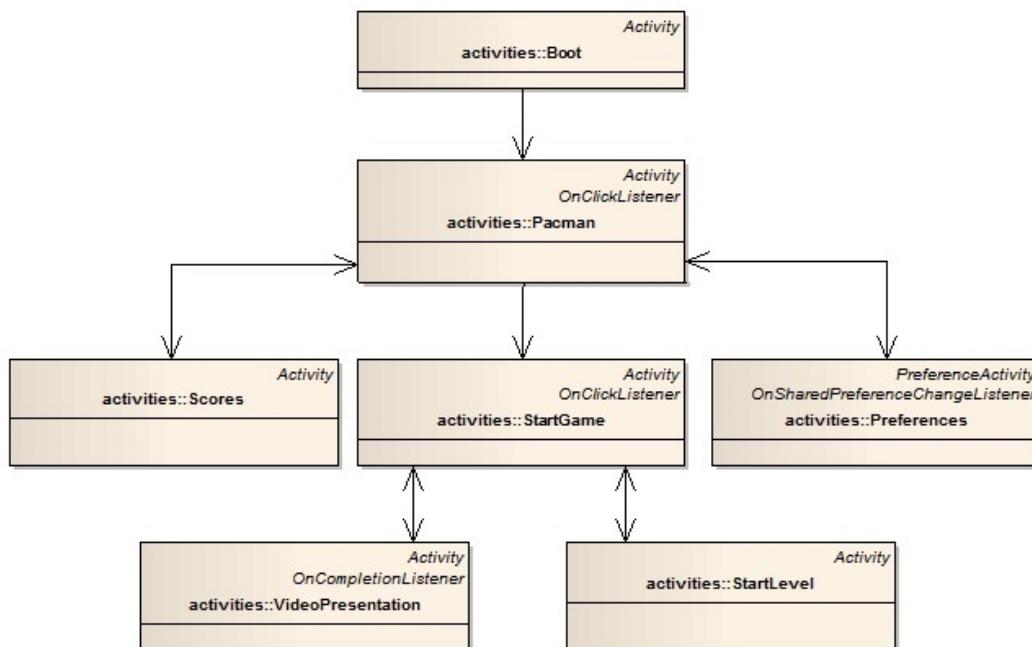


Figura 7.1: Navegación entre Actividades

- **Boot**: primera actividad de la aplicación donde se carga la información común a todas las actividades en el contexto.
- **Pacman**: comienza una vez finalizada la carga del contexto y contempla el menú general de la aplicación.
- **Preferences**: muestra un listado con los distintos parámetros configurables en el videojuego.
- **Scores**: muestra un listado con las diez mejores puntuaciones que los jugadores han obtenido en el videojuego.
- **StartGame**: accesible desde la actividad principal y permite iniciar una nueva partida, mostrando inicialmente los distintos niveles disponibles.

- **StarLevel:** se corresponde con un escenario del juego. Esta actividad es iniciada tras seleccionar en la actividad StartGame el nivel en el cual se desea jugar.
- **VideoPresentantion:** muestra un video correspondiente a un escenario.

El nombre de la primera actividad al iniciar la aplicación es Boot. Su objetivo es cargar la información del contexto de la aplicación, mostrando mensajes e informando si todo funciona correctamente o no. Para poder explicar el comportamiento interno de esta actividad, se hará hincapié en la estructura de directorios del videojuego, llamada **PacmanConfig**, donde se guarda la información necesaria para el proceso de carga de la aplicación.

7.2. PacmanConfig

Durante la ejecución del videojuego existe una gran cantidad de datos que han de ser recuperados del sistema de ficheros, algunos de estos ejemplos son las vídeos, sonidos, melodías, texturas... El nombre por defecto de este directorio es PacmanConfig y se encuentra ubicado, por defecto, en el directorio principal de la tarjeta SD. La estructura de directorios de la que está compuesta es la siguiente:

- **meshes:** contiene las distintas mallas de los elementos comunes como son los fantasmas y el pacman. Cada una de las mallas estará en su directorio con todos los ficheros necesarios en función del tipo de malla. Un ejemplo del gráfico es la malla de blueghost, que se corresponde con una TextureShaderMesh.

El fichero principal es texture.mesh y se corresponde con un fichero de tipo .obj de Wavefront. Este fichero, como vimos en el capítulo anterior, hace a su vez referencia a los materiales del objeto ubicados en el fichero blueghost.mtl. Este último enlaza con las distintas texturas, aunque en este caso sólo hay una y se corresponde con el fichero blueghost.png.

- **modules:** contiene los distintos módulos que pueden ser cargados en el juego, entendiendo por módulo una secuencia de escenarios.
- **music:** contiene la música utilizada por cada uno de los módulos. Por ejemplo, el fichero Intro.mp3 es la melodía que suena al iniciar el juego.
- **videos:** contiene los vídeos utilizados por cada uno de los módulos. Por ejemplo, GameComplete.mp4 y GameOver.mp4 se corresponden con los vídeos que se muestran al finalizar todo el juego o perder todas las vidas.



Capítulo 7. Diseño de Pacmania

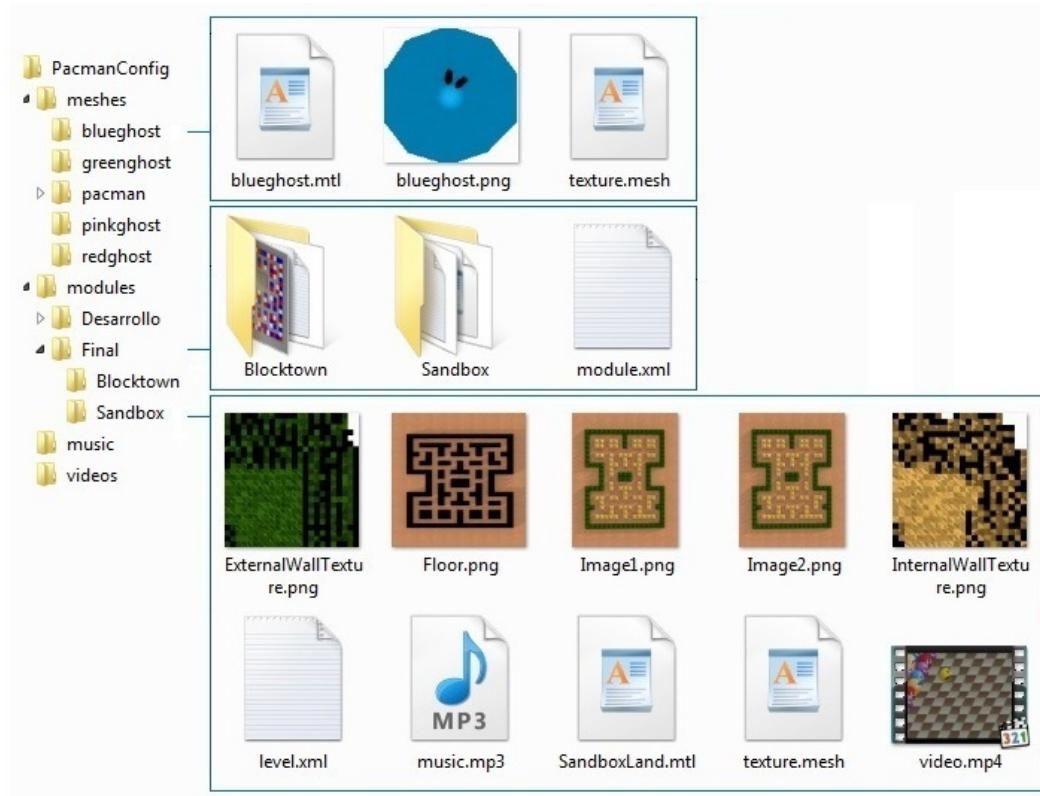


Figura 7.2: Estructura del directorio de configuración del Pacman

Módulo

Cada módulo está compuesto por un fichero **module.xml** que describe el módulo, un fichero **music.mp3** que contiene la melodía del módulo y un listado de directorios, correspondientes a las escenas referenciadas en el fichero module.xml.

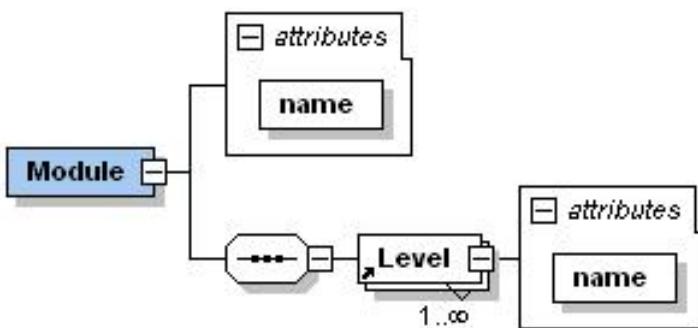


Figura 7.3: Esquema Xml del fichero module.xml

Ejemplo de module.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <Module name="Test">
3    <Level name="Blocktown"/>
4    <Level name="Sandbox"/>
5  </Module>
```

La etiqueta principal del fichero module.xml es **Module**. El atributo **name** indica el nombre del módulo y por cada sub-etiqueta **Level**, se está referenciado a un escenario del videojuego. El atributo **name** de la etiqueta **Level**, hace referencia al nombre del directorio que contiene su configuración.

Niveles

Hasta el momento se ha definido un módulo como una agrupación de escenarios, siendo éste, un concepto definido por la librería desarrollada. En el ámbito del videojuego es necesario ampliar las características del escenario, dando origen a un nuevo concepto, el de nivel, implementado por la clase **Level** y que extiende a la clase **Scene**.

Un nivel contiene nueva información como capturas de pantalla del escenario, referencias a ciertos elementos del juego como el pacman y los fantasmas, listado de cámaras utilizadas en distintos momentos del videojuego, etc.

La información del nivel es almacenada en un directorio, especificado en el fichero module.xml, que contiene los siguientes ficheros:

- **Image1.jpg**, una captura de pantalla del nivel con una resolución 64x64.
- **Image2.jpg**, una captura de pantalla del nivel con una resolución 256x256.
- **Texture.mesh** que contiene la información de la malla con todo el escenario.
- **Music.mp3** con la melodía del escenario.
- **video.mp4** con el vídeo de introducción del escenario.
- **Level.xml**: que contiene la información asociada al nivel:
 - Atributo **name**: nombre del nivel.
 - Atributo **description**: descripción del nivel.
 - Etiqueta **Pacman_Start**: posición inicial del Pac-man.
 - Etiqueta **Ghost**: información de los fantasmas sobre su posición inicial en el escenario, la velocidad, su algoritmo y su malla.



Capítulo 7. Diseño de Pacmania

- Etiqueta Small_Pill: posición de cada pastilla del escenario.
- Etiqueta Wall: posición de cada bloque de pared del escenario.

Es posible encontrarnos otros ficheros en el directorio, consecuencia de la definición de la malla. Estos ficheros pueden ser de tipo mtl e imágenes, los cuales representan los materiales y texturas definidos en el escenario.

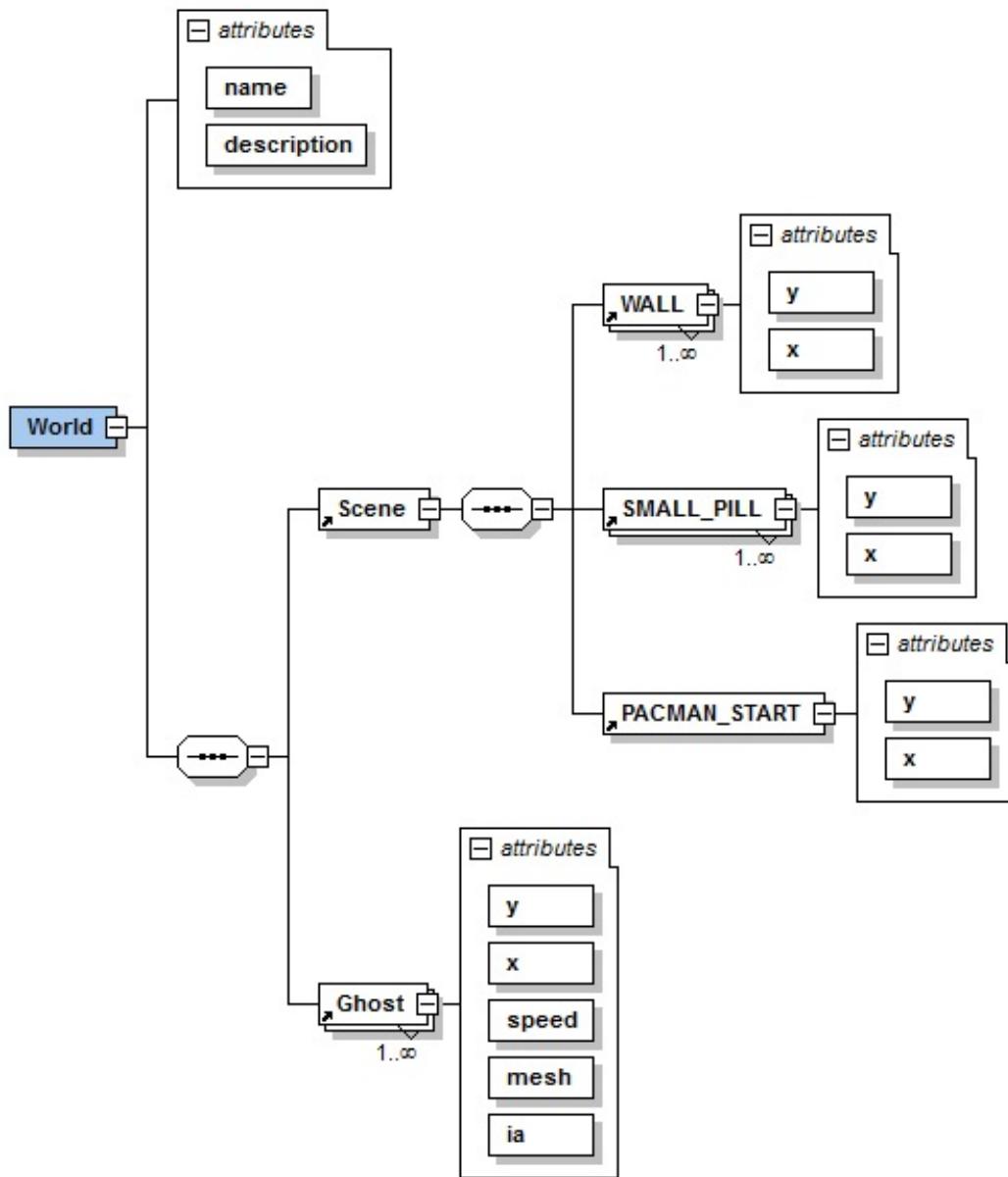


Figura 7.4: Esquema xsd de level.xml

Ejemplo de fichero level.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <World name="Blocktown" description="Escenario de lego">
3      <Scene>
4          <WALL           x="0"   y= "17"/>
5          <WALL           x="1"   y= "17"/>
6          .....
7
8          <SMALL_PILL     x="1"   y= "16"/>
9          <SMALL_PILL     x="2"   y= "16"/>
10         .....
11         <PACMAN_START   x="6"   y= "5"/>
12         .....
13     </Scene>
14     <Ghost x="6" y="13" mesh="Ghost1" speed="7" ia="0"/>
15     <Ghost x="6" y="13" mesh="Ghost2" speed="5" ia="0"/>
16 </World>

```

Lector de niveles

Los ficheros level.xml son procesados por la aplicación para obtener instancias de la clase Level. El responsable de procesar estos ficheros es la clase **LevelReader**. Según se vayan procesando las etiquetas ocurrirán las siguientes acciones:

- Etiqueta World: se obtiene el nombre y descripción del nivel.
- Etiqueta Scene: se crea la malla del escenario utilizando la clase MeshFileReader.
- Etiqueta Wall: se crea una geometría AABB en función de las coordenadas X e Y de la etiqueta y se añade a la geometría del nivel.
- Etiqueta Small_Pill: se añade una nuevo vértice a la clase PathFinder ya que es un posible punto por el que los fantasmas y el Pacman pueden pasar y debe ser procesado en el algoritmo de Dijkstra. También se añade una nueva pastilla al objeto PillsNetElement indicando su posición.
- Etiqueta Pacman_Start: se añade una nuevo vértice a la clase PathFinder y se indica al elemento Pacman de la clase Level sus coordenadas y velocidad inicial.
- Etiqueta Ghost: se añade un nuevo elemento al nivel de tipo GhostElement. En el constructor se indica su posición inicial en el escenario, su velocidad, si el comportamiento es aleatorio o de persecución y la referencia al Pacman, que es el elemento que ha de perseguir.



Capítulo 7. Diseño de Pacmania

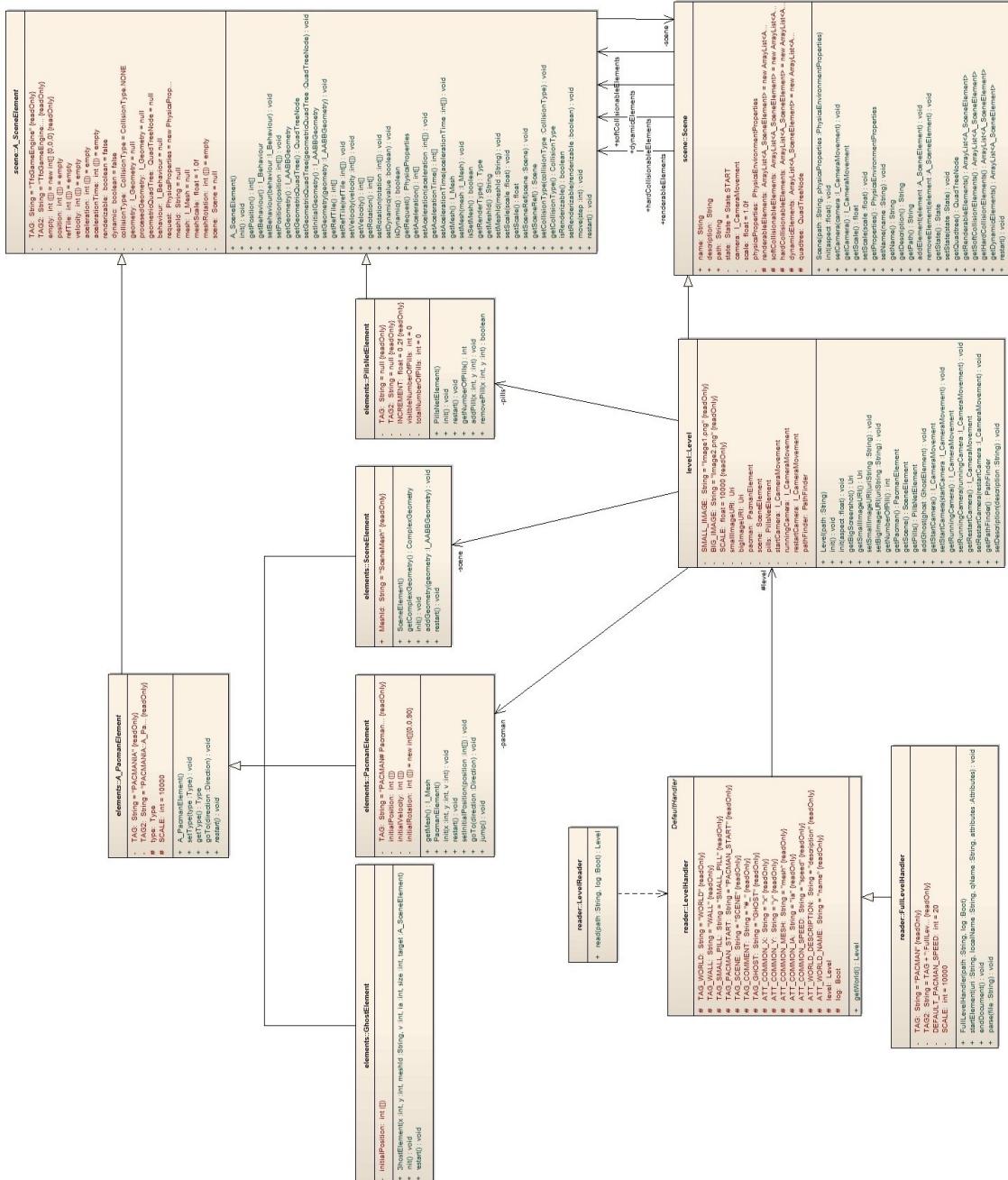


Figura 7.5: Diagrama de clases de LevelReader

7.2. PacmanConfig

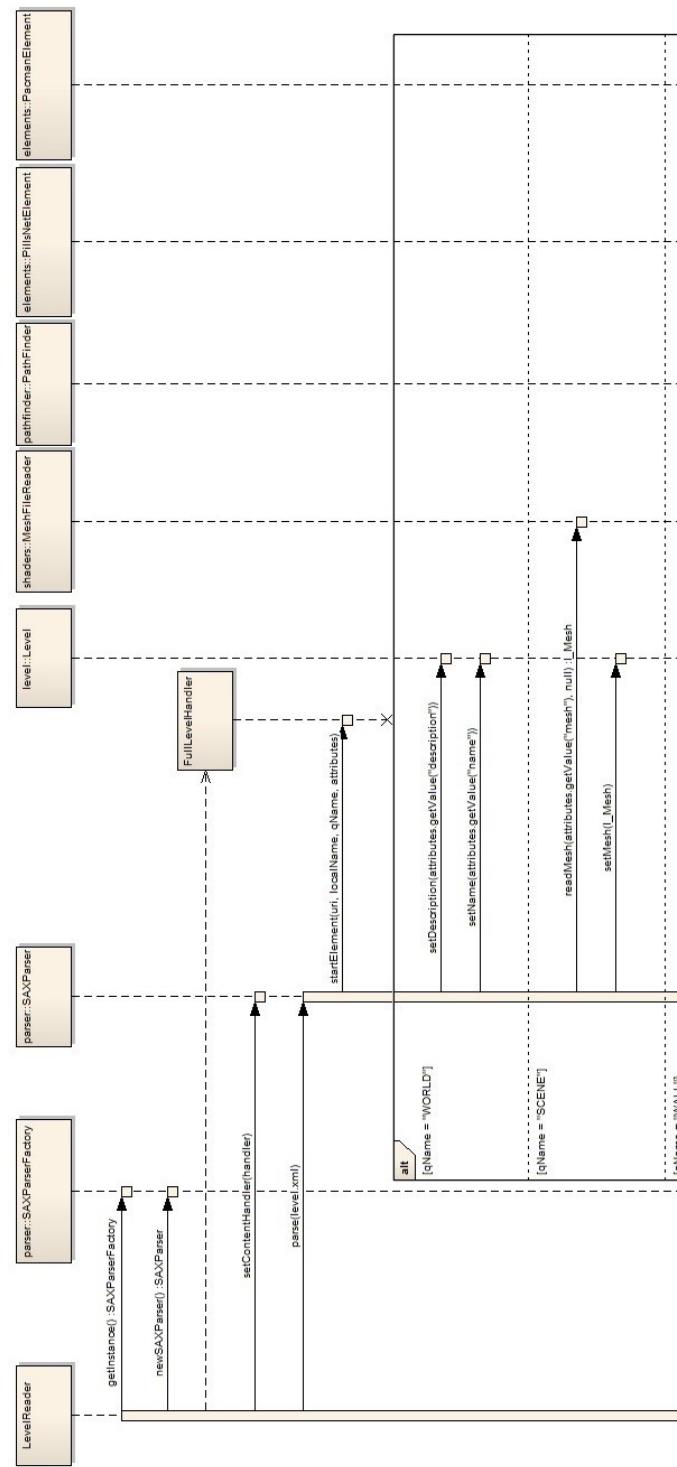


Figura 7.6: Diagrama de secuencia para procesar el fichero level.xml (Parte 1)



Capítulo 7. Diseño de Pacmania

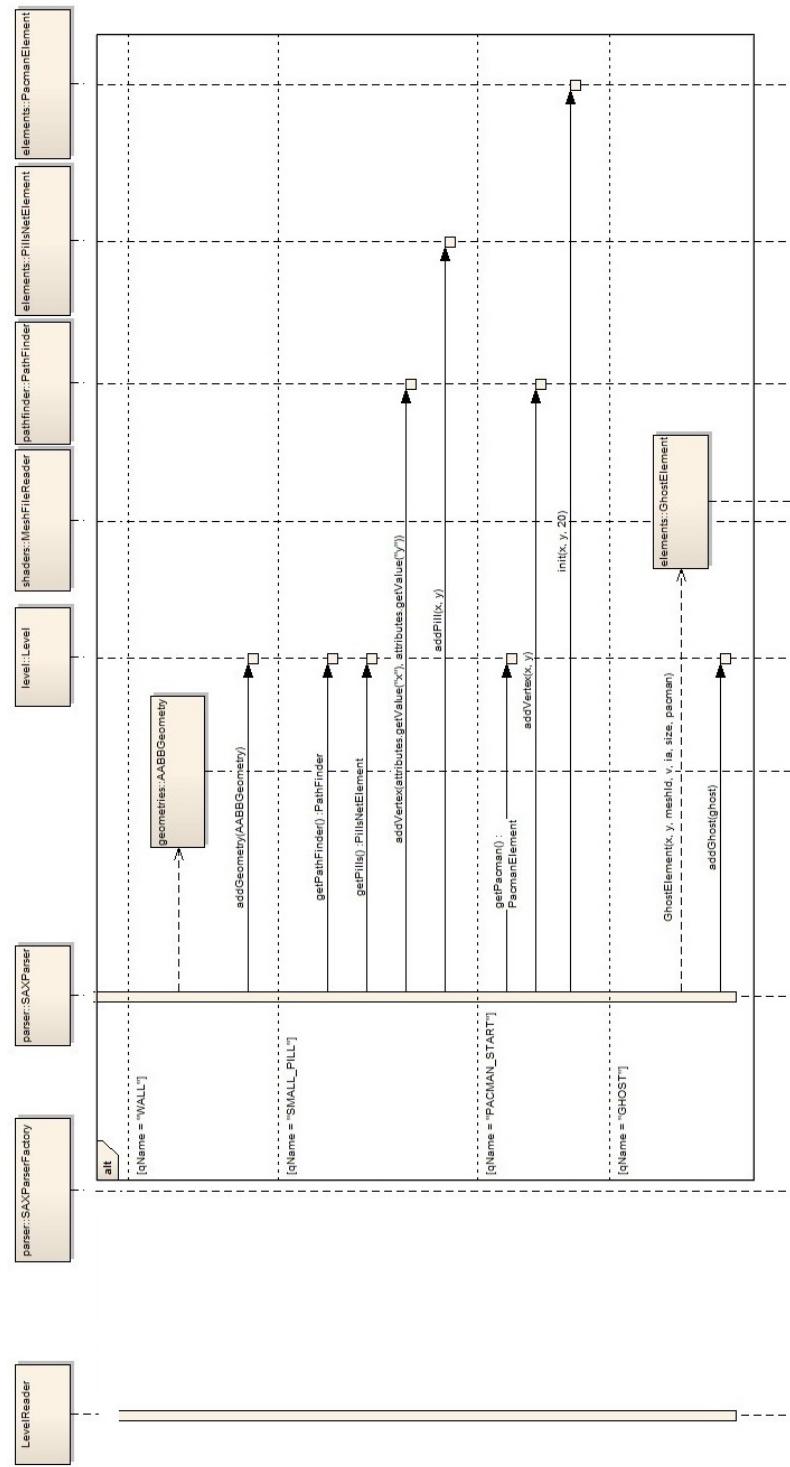
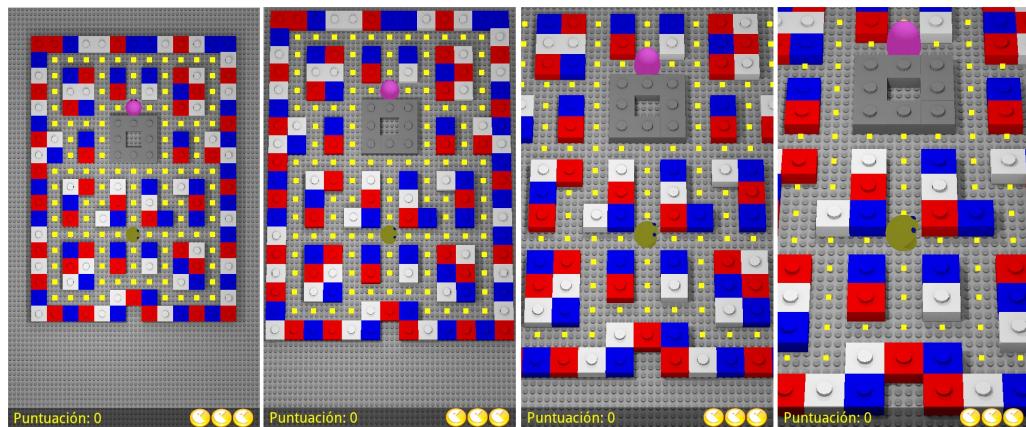


Figura 7.7: Diagrama de secuencia para procesar el fichero level.xml (Parte 2)

Cámaras

La clase Scene contiene una única cámara, a través de la cual se renderiza la escena. La clase Level contiene un listado de cámaras con distintos comportamientos. Según lo requiera el estado del videojuego se selecciona cuál de estas cámaras es la adecuada para renderizar la escena. En éste proyecto se han definido tres tipos de cámaras:

- **Cámara 1:** al comenzar la pantalla se realiza un zoom, acercándose al Pacman y a su vez inclinándose para percibir las tres dimensiones del escenario.



- **Cámara 2:** cuando el Pacman ha sido capturado se realiza un giro de 360 grados mientras se aleja.



- **Cámara 3:** es el habitual en la escena donde la cámara persigue al Pacman por el escenario desde una distancia fija.

Pacman

Dentro del directorio meshes se encuentra el directorio pacman, que contiene la malla del elemento Pacman. La característica a resaltar de este elemento radica en que es una animación, compuesta por dos mallas, una con la boca abierta y otra cerrada.



Capítulo 7. Diseño de Pacmania

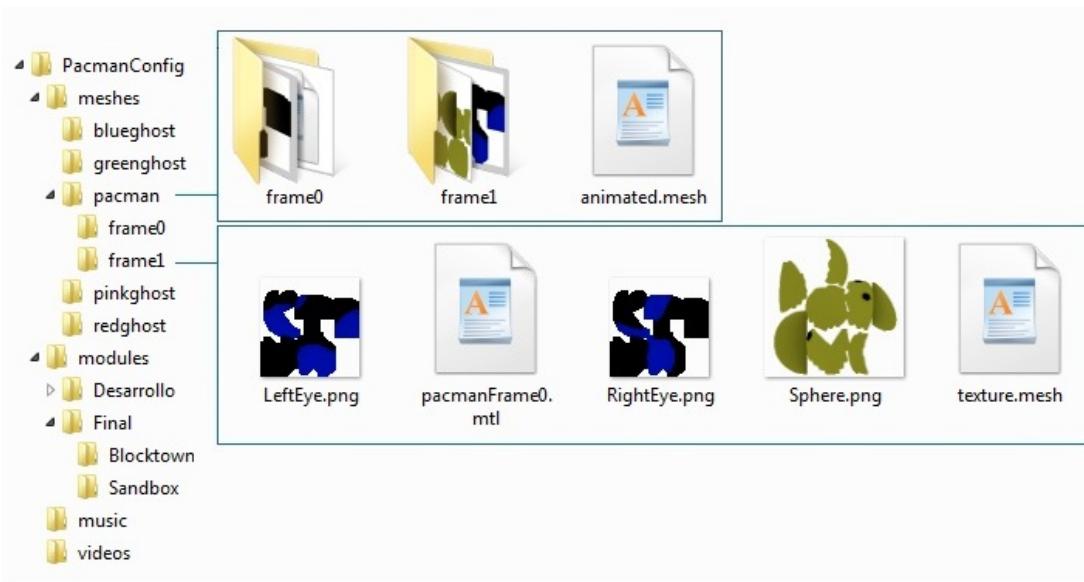


Figura 7.8: Estructura del directorio del elemento Pacman

El fichero **animated.mesh** indica la secuencia de mallas que componen la animación. En cada línea se expresa el directorio que contiene la malla junto con el tiempo que se tiene que mostrar en milisegundos.

Fichero animasted.mesh del elemento Pacman

```
1 # Secuencia de mallas
2   frame0 /500
3   frame1 /500
```

Como se puede apreciar en el contenido del fichero, ambos frames se muestran 500 milisegundos y los nombres de frame0 y frame1 se corresponden con los directorios que contienen la información de las mallas. Al igual que la malla de BlueGhost, estos directorios contienen el fichero texture.mesh junto con los ficheros de materiales e imágenes descritos en él.

Generación de mallas con Blender

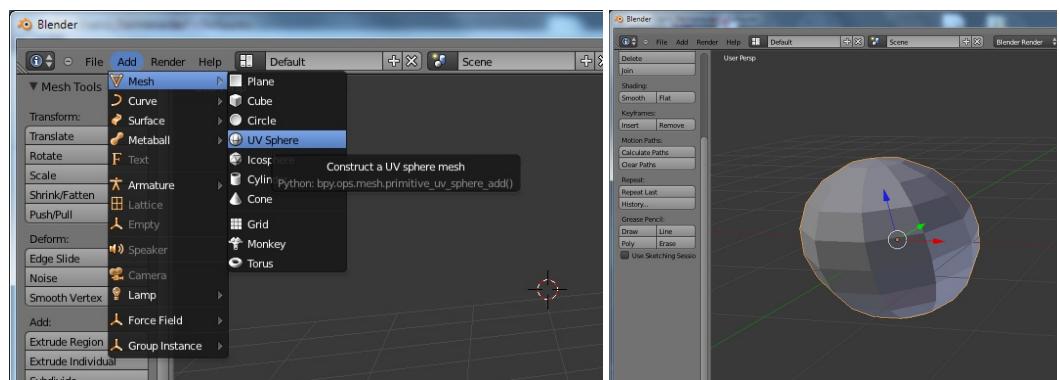
Entre todos los ficheros que se encuentran en el directorio PacmanConfig, resaltan debido a su complejidad los ficheros asociados con las mallas, por ese motivo han sido creados con la herramienta Blender y posteriormente dichos ficheros se leerán mediante la librería desarrollada.

7.2. PacmanConfig

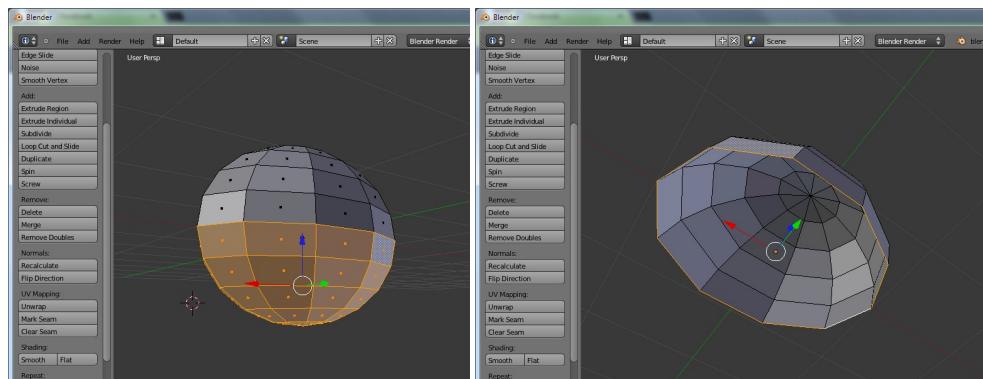
Blender es una herramienta para generación de animaciones 3D tan compleja que existe un mercado laboral en torno a ella. Hay profesionales específicos para ciertas funcionalidades de la aplicación, por ejemplo, algunos se dedican a crear las mallas y otros se especializan en cómo aplicarles las texturas. Obviamente, en el ámbito del proyecto tan sólo ha sido posible utilizar de forma somera, el potencial de la herramienta.

En el siguiente ejemplo se va a explicar cuál es el proceso seguido para generar la malla correspondiente a un fantasma del juego. Este proceso ha sido definido tras múltiples pruebas. En cada una de ellas se observaba el fichero final resultante de las acciones realizadas en el programa.

1. El primer paso, tras arrancar la aplicación, es crear una esfera tal y como se muestra en el gráfico.



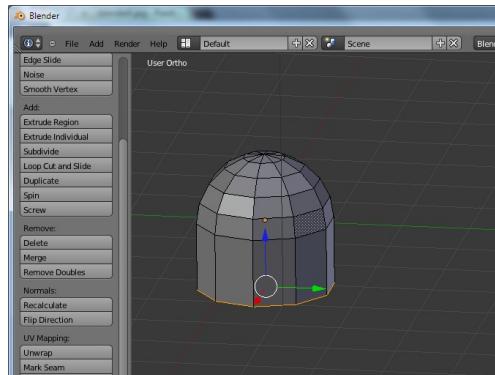
2. Seleccionaremos sólo la mitad de la esfera. El resto de los vértices los borraremos, obteniendo media esfera.



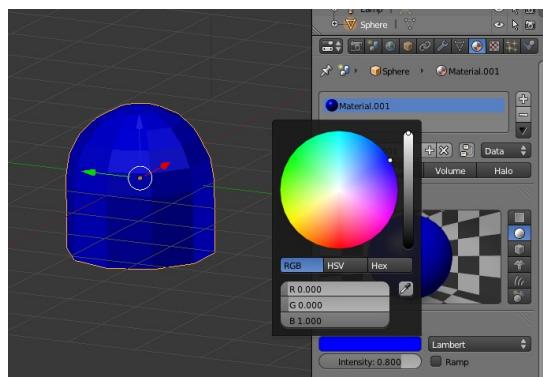
3. El siguiente paso consiste en seleccionar las aristas de la base de la media esfera y moverlas para crear la figura final del fantasma.



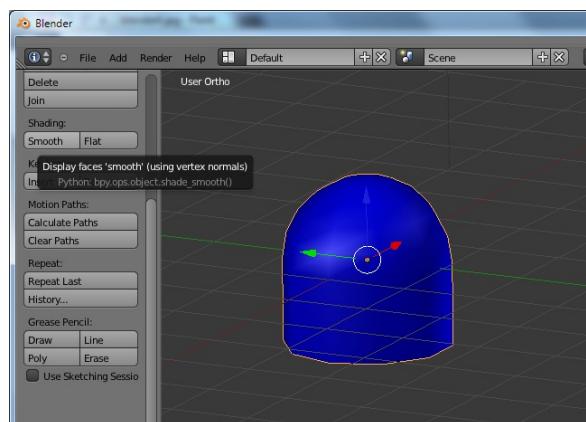
Capítulo 7. Diseño de Pacmania



4. Aplicamos un material sobre la figura creada indicando el color de especulación, difusión y ambiental.



5. En la imagen aún se notan las cambios entre las distintas caras de la figura. Esto es debido a que el tipo de sombreado es Flat. Para que el salto sea homogéneo vamos a modificar el tipo de sombreado a Pong¹.

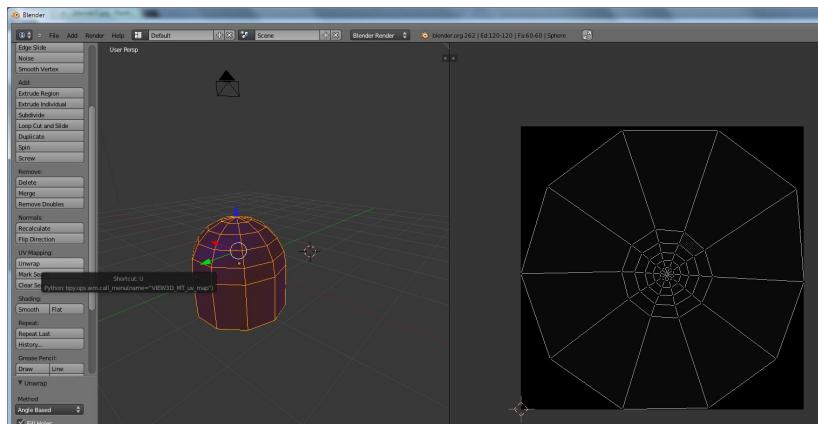


6. El siguiente paso es generar las texturas. Unwrap es el comando encargado de trasladar cada uno de los vértices de la figura sobre una imagen de color negro.

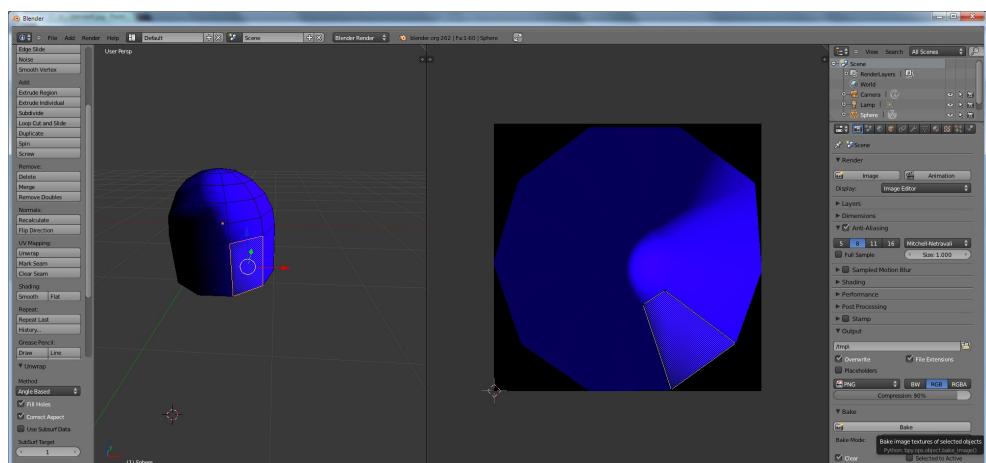
¹El sombreado Phong es conocido por Smooth en las aplicaciones de diseño gráfico 3D

7.2. PacmanConfig

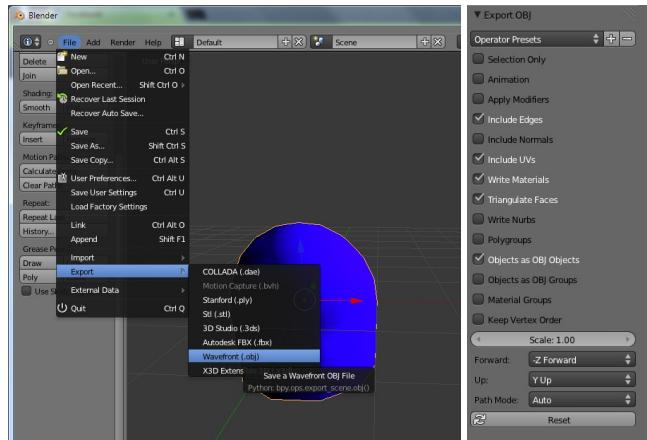
En función de la complejidad de la malla que queramos mapear, es posible que encontremos situaciones donde la aplicación realiza el mapeo directamente u otras en las que hay que indicárselo manualmente, vértice a vértice.



7. La imagen actual es negra y no refleja la realidad representada en la aplicación, por tanto, hemos de 'cocinar la escena'. Este concepto es conocido como **Texture bake** y consiste en aplicar un modelo de iluminación sobre los elementos de la escena y determinar su color final, atendiendo a los materiales y luces que la componen. En la captura de pantalla se puede ver el resultado de la operación y cómo se corresponde la cara seleccionada con la imagen. Esta correspondencia se trasladará al fichero obj mediante los vértices de la textura.



8. El último paso, consiste en exportar el trabajo realizado en Blender, al formato Wavefront. Es indispensable indicar las opciones de exportación correctas, de otro modo, la librería no sería capaz de renderizar correctamente la malla. Se ha de incluir las opciones de mapeo de texturas (Include UVs) y la exportación de las caras mediante triángulos (Triangule Faces).



9. El resultado final es un fichero .obj que contendrá la información de la malla y hará referencia a un fichero .mtl con los materiales. En este caso, sólo existirá un material el cual hará referencia a la imagen cocinada.



7.3. Actividad de arranque

Una vez comprendida la estructura de directorio PacmanConfig, nos centramos en la actividad de arranque, implementada por la clase Boot. Esta actividad muestra las trazas de las comprobaciones realizadas al arrancar la aplicación. El primer mensaje indica que la aplicación se corresponde con un proyecto de fin de carrera y se muestra el nombre del alumno, el tutor, la universidad y facultad en la que se presenta.

El segundo mensaje es el resultado de comprobar si existe el directorio donde está toda la configuración de la aplicación. Esta ubicación se establece en la propiedad PacmanConfigPath, cuyo valor por defecto es /sdcard/PacmanConfig. Por motivos de compatibilidad con otros dispositivos, es posible cambiar esta ruta en la actividad de preferencias del videojuego

El siguiente paso es iniciar el gestor de música y cargar los sonidos que se van a utilizar durante el juego. En concreto los sonidos a cargar son chomp y deatch, los cuales se reproducen cuando existe una colisión entre el pacman-pastilla o pacman-fantasma respectivamente. Al cargar cada uno de los sonidos se mostrará una traza informativa.

7.4. Actividad principal

A continuación se buscan las mallas comunes a todos los escenarios del juego, es decir, los correspondiente a los fantasmas y pacman. Todos ellos están contenidos en el subdirectorio meshes del PacmanConfig. Cada fichero que contiene una malla se procesará y cargará en la memoria utilizando la clase MeshFileReader, desarrollada en la librería del proyecto. Por cada fichero Wavefront que se vaya a procesar, ya sea obj o mtl, se mostrará un mensaje informativo.



Figura 7.9: Actividad de arranque (Boot)

En relación al módulo, la propiedad module de la aplicación indica cuál es el módulo que ha de ser cargado entre los disponibles en el directorio PacmanConfig. Mediante la clase **ModuleHandler** se va a procesar el fichero module.xml y por cada uno de las referencias a un nivel, se utilizará la clase LevelReader para procesarlo.

Una vez completadas todas estas acciones, el contexto del videojuego estará cargado, se finalizará la actividad de arranque y se cargará la actividad principal.

7.4. Actividad principal

La actividad, implementada con la clase Pacman, está compuesta únicamente por el menú principal de la aplicación que permite realizar las siguientes acciones:

- **Comenzar juego:** inicia una partida para lo cual carga la actividad StartGame.
- **Configuración:** muestra el menú con las opciones del videojuego contenido en la actividad Preferences.
- **Marcadores:** inicia la actividad Score para ver las diez mejores puntuaciones obtenidas por los jugadores.



Capítulo 7. Diseño de Pacmania

- **Acerca de:** muestra una ventana de información indicando que la aplicación se trata de un proyecto y su información relacionada (autor, tutor, año...).
- **Salir:** finaliza la actividad.

De fondo, al iniciar la actividad, se inicia la melodía ubicada en la ruta PacmanConfig/music/Intro.mp3.



(a) Actividad Pacman



(b) Acerda de:

7.5. Preferencias

Desde la actividad principal, tras pulsar la opción de Preferencias, accedemos a la actividad **Preferences** que extiende de la clase nativa de Android **PreferenceActivity** e implementa la interfaz **OnSharedPreferenceChangeListener**.

La clase PreferenceActivity permite crear pantallas de configuración de propiedades fácilmente, cargando a partir de un fichero xml las propiedades que se desean configurar. Además estas propiedades serán almacenadas automáticamente y de forma persistente.

La interfaz OnSharedPreferenceChangeListener permite detectar cualquier cambio en las propiedades de la aplicación y actuar en consecuencia. El ejemplo más claro se

produce al deshabilitar el sonido, provocando una invocación a la clase SoundEngine para que deje de reproducir cualquier tipo de sonido.

Las opciones a configurar en la aplicación son:

- **Sonido:** indicando si el sonido debe estar o no activo en el juego.
- **PacmanConfig Path:** ubicación en la tarjeta SD donde se encuentra el directorio PacmanConfig.
- **Directorio del módulo:** donde se indica cuál será el módulo a cargar en el videojuego de los existentes en el directorio PacmanConfig. Los módulos disponibles no se pueden cargar en un xml ya que se encuentran definidos en la estructura de directorios, por lo que dicho menú se crea de forma dinámica.

Fichero xml de configuración setting.xml

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <PreferenceScreen
4
5      xmlns:android="http://schemas.android.com">
6
7      <CheckBoxPreference
8          android:defaultValue="true"
9          android:key="music"
10         android:summary="@string/sound_summary"
11         android:title="@string/sound_title"/>
12
13     <EditTextPreference
14         android:defaultValue="/sdcard/PacmanConfig/"
15         android:key="PacmanConfig_Path"
16         android:summary="@string/configurationpath_summary"
17         android:title="@string/configurationpath_title"/>
18
19 </PreferenceScreen >
```

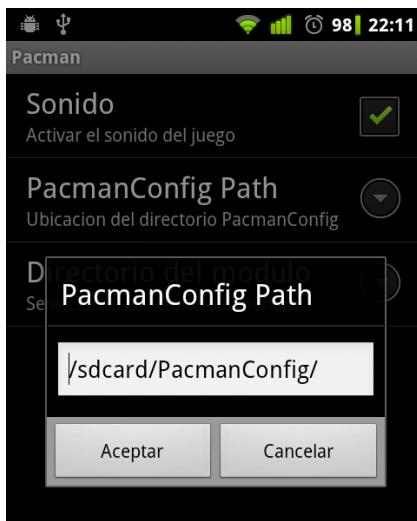
En el fichero xml de configuración, podemos observar que no se muestra la opción del directorio del módulo. Esto es debido a que esta propiedad no es estática, depende de la información contenida en el directorio PacmanConfig, por ese motivo es cargada en tiempo de ejecución.



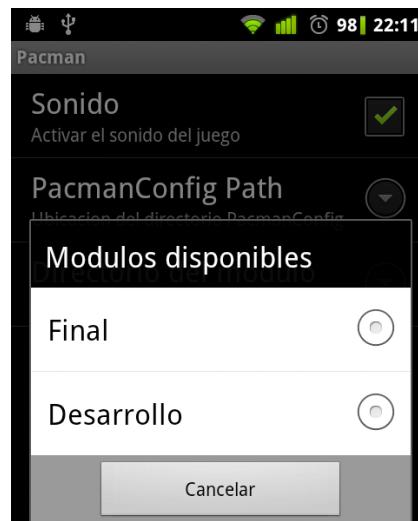
Capítulo 7. Diseño de Pacmania



(a) Preferencias



(b) PacmanConfigPath



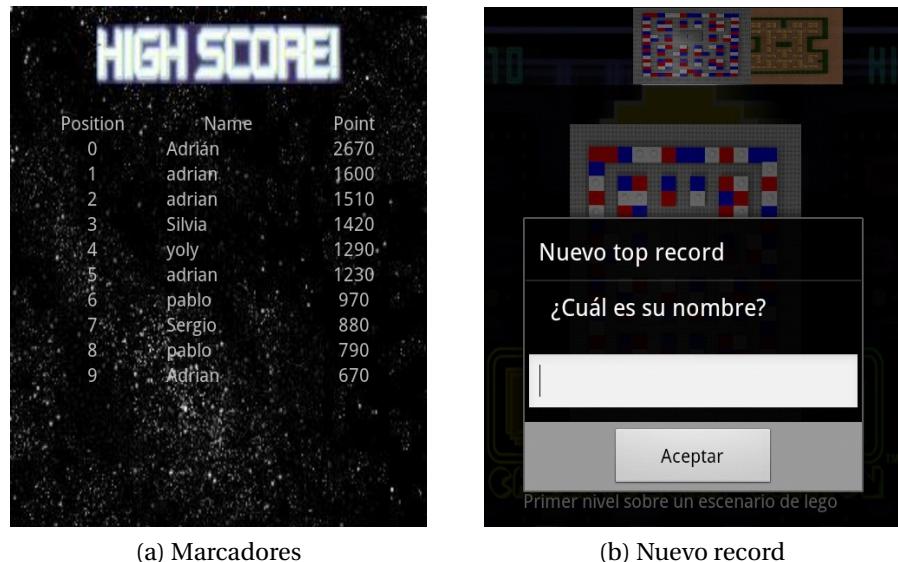
(c) Módulos

7.6. Marcadores

Al pulsar el botón Marcadores de la actividad Pacman, se abre la actividad **Scores** que contiene un listado con las diez mejores puntuaciones. Por cada registro se indica, además de la puntuación, el nombre de la personas que lo consiguió, como se puede apreciar en la siguiente imagen.

Estas puntuaciones son almacenadas en una base de datos SQLite, sobre las cuales existe un soporte nativo en Android. Sus principales ventajas son la rapidez y bajo consumo de recursos. Por contra, no es posible realizar accesos de escritura de forma

concurrente, debido a que almacena toda la información en un único fichero, que se bloquea al ser modificado.



La clase **GenericDAO** extiende a la clase **SQLiteOpenHelper** y contiene el método **onCreate**. En este método se crea la base de datos, en el caso de que no exista, en el espacio de disco reservada para la aplicación.

La clase **ScoreDAO** tiene la lógica requerida sobre la tabla Score. Permite el acceso de lectura para obtener las diez mejores puntuaciones y guardar puntuaciones nuevas. Estas funcionalidades son accedidas por la actividad Score y StartGame.

```

Método OnCreate del GenericDAO
1  @Override
2  public void onCreate(SQLiteDatabase db) {
3
4      db.execSQL(
5          "CREATE TABLE " +
6          ScoreDAO.TABLE_TITLE + "(" + ScoreDAO.TABLE_ID +
7          " INTEGER PRIMARY KEY AUTOINCREMENT , " +
8          ScoreDAO.TABLE_NAME + " TEXT NOT NULL , " +
9          ScoreDAO.TABLE_MODULE + " TEXT NOT NULL , " +
10         ScoreDAO.TABLE_LEVEL + " TEXT NOT NULL , " +
11         ScoreDAO.TABLE_POINTS + " INTEGER , " +
12         ScoreDAO.TABLE_DATE + " TEXT NOT NULL );"
13     );
14 }
```



Capítulo 7. Diseño de Pacmania



Figura 7.10: Diagrama de clases sobre SQLite

7.7. Actividad StarGame

El objetivo inicial de esta actividad es seleccionar el escenario en el cual jugar, de los contenidos en el módulo cargado. Visualmente está compuesta en su parte superior con el componente **Gallery** de Android. Este componente gráfico permite mostrar un listado de imágenes, las cuales se corresponden con un escenario del videojuego.

7.7. Actividad StarGame

La parte inferior está compuesta por tres elementos gráficos que representan el nombre del escenario, una descripción del mismo y una imagen con mayor detalle del escenario.

Al pulsar sobre cada imagen de la galería, la parte inferior cambia, mostrando los datos correspondientes al escenario seleccionado. También cabe destacar un cambio de música, el cual se corresponde con el nivel seleccionado.

La información relativa a cada nivel se encuentra almacenada en el contexto de la aplicación y ha sido cargada previamente en la actividad Boot.

Volviendo sobre el ejemplo del escenario SandBox, el nombre y la descripción del escenario estaría contenida en los atributos del fichero level.xml, las imágenes mostradas se corresponden con Image1.png e Image2.png y la música con el fichero music.mp3.

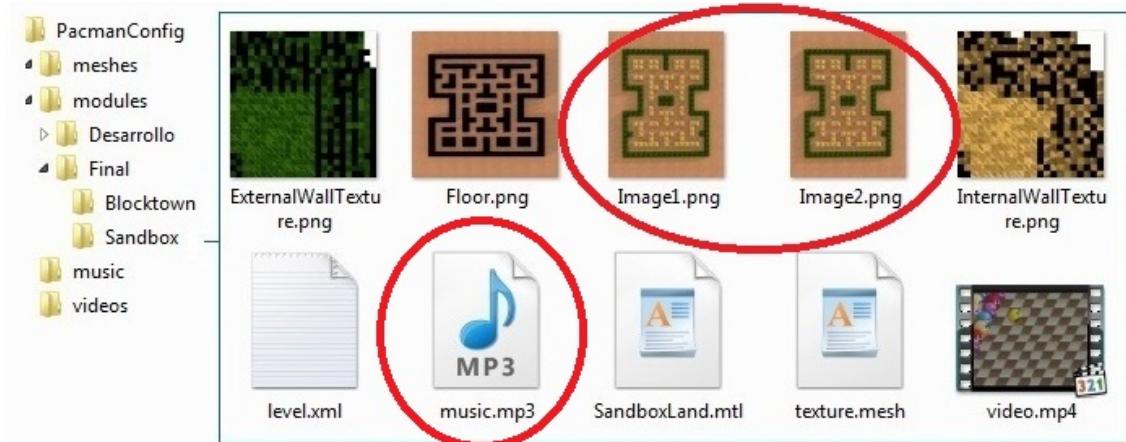


Figura 7.11: Configuración escenario SandBox

Una vez seleccionado un escenario, si pulsamos sobre la imagen central del escenario en el que queremos jugar, comienza el objetivo principal de esta actividad, que consiste en coordinar la sucesión de escenarios y vídeos.

Tras haber seleccionado el escenario en el cual queremos jugar, se mostrará el vídeo asociado a dicha pantalla, que está almacenado con el nombre video.mp4 en el directorio raíz del mismo. Una vez concluido el vídeo, se creará una nueva actividad de tipo **StartLevel** que cargará el escenario seleccionado para jugar en él.

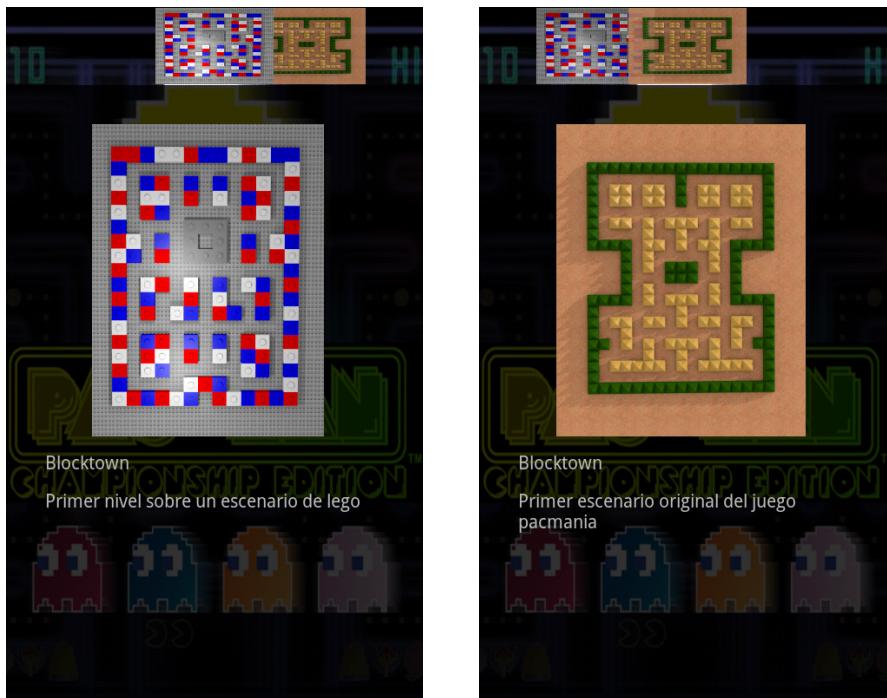


Figura 7.12: Captura de pantalla de la actividad StartGame

La actividad StartLevel puede concluir de dos formas, habiendo finalizado el escenario o cuando el usuario pierde todas las vidas. La actividad StartGame ha de identificar cuál es el motivo de cierre de la actividad StartLevel y actuar en consecuencia ya que:

- Si el usuario ha perdido todas las vidas de las que dispone, se ha de mostrar el vídeo GameOver.mp4 ubicado en la carpeta videos del directorio principal.
- Si el usuario ha finalizado con éxito la pantalla, en caso de no ser la última se ha de proceder a cargar el vídeo del siguiente escenario. Si no existen mas pantallas se ha de mostrar el vídeo GameComplete.mp4 ubicado en el mismo lugar que GameOver.mp3.
- Tras mostrarse el vídeo de un escenario, ha de crearse una actividad StartLevel correspondiente al vídeo mostrado en pantalla.
- Tras finalizar el vídeo de GameComplete.mp4 o GameOver.mp4 se comprobará si la puntuación obtenida esta entre las diez mejores del juego. Si esto ocurre se solicitará al usuario introducir su nombre para grabarlo en el histórico.

7.8. Actividad StartLevel

A nivel visual, la actividad **StartLevel** está compuesta por dos componentes gráficos:

- El primero de ellos ocupa toda la pantalla y se corresponde con el componente de Android **GLSurfaceView**. Es responsable de capturar los eventos producidos sobre la pantalla y contiene en su interior un render de OpenGL. El render a utilizar es el **PacmanRenderEngine**, el cuál extiende el desarrollado en la librería del proyecto.
- El segundo es un layout situado en la parte inferior de la pantalla. Contiene el número de vidas del jugador y la puntuación actual.

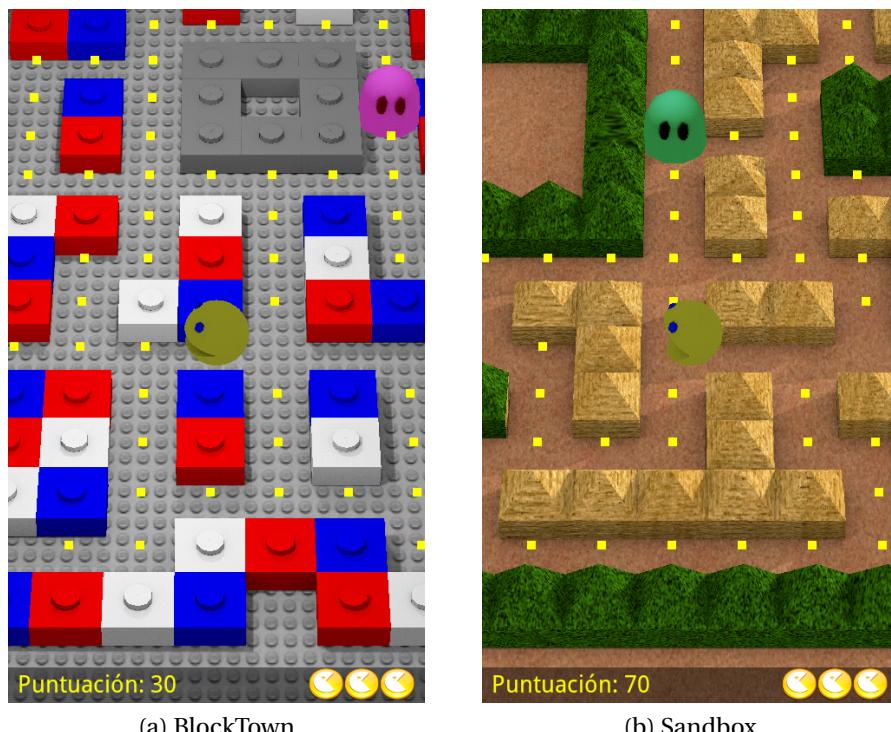


Figura 7.13: Captura de pantalla de los escenarios desarrollados

La forma de interactuar con el componente **GLSurfaceView** se basa en la detección de movimientos del dedo sobre la pantalla. El método **onTouchEvent** es invocado al detectar un movimiento sobre la pantalla táctil, ofreciéndonos información sobre la posición que se está presionando y el tipo de evento entre los cuales están:

- ActionDown: ocurrido cuando el usuario presiona con el dedo la pantalla.
- ActionUp: ocurrido cuando el usuario despega el dedo de la pantalla.



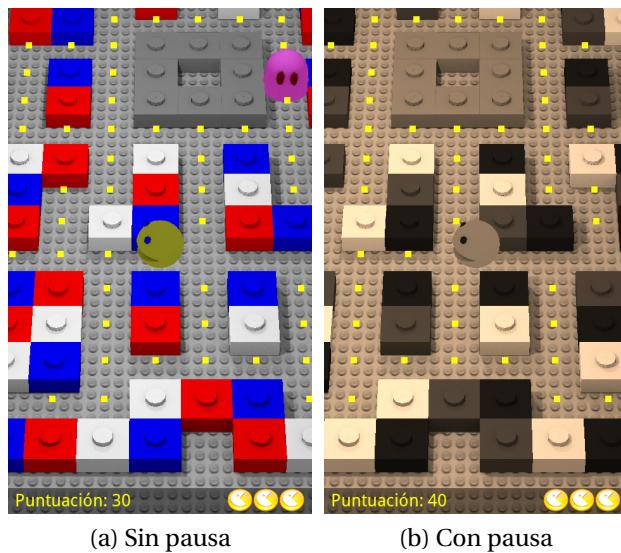
Capítulo 7. Diseño de Pacmania

Al analizar la secuencia de eventos provocados por la pantalla táctil, se realizarán las siguientes acciones:

- Al desplazar el dedo de izquierda a derecha, estará indicando que el Pacman ha de desplazarse hacia la derecha, en cuanto sea posible.
- Al desplazar el dedo de derecha a izquierda, estará indicando que el Pacman ha de desplazarse hacia la izquierda, en cuanto sea posible.
- Al desplazar el dedo de arriba a abajo, estará indicando que el Pacman ha de desplazarse hacia abajo, en cuanto sea posible.
- Al desplazar el dedo de abajo a arriba, estará indicando que el Pacman ha de desplazarse hacia arriba, en cuanto sea posible.
- Si se presiona la pantalla con el dedo y seguidamente, sin desplazarlo, se levanta, el Pacman saltará.

El elemento Pacman, cuyo comportamiento se basa en la clase PacmanBehaviour, intentará moverse en la dirección indicada, cuando la posición actual del Pacman sobre el escenario lo permita.

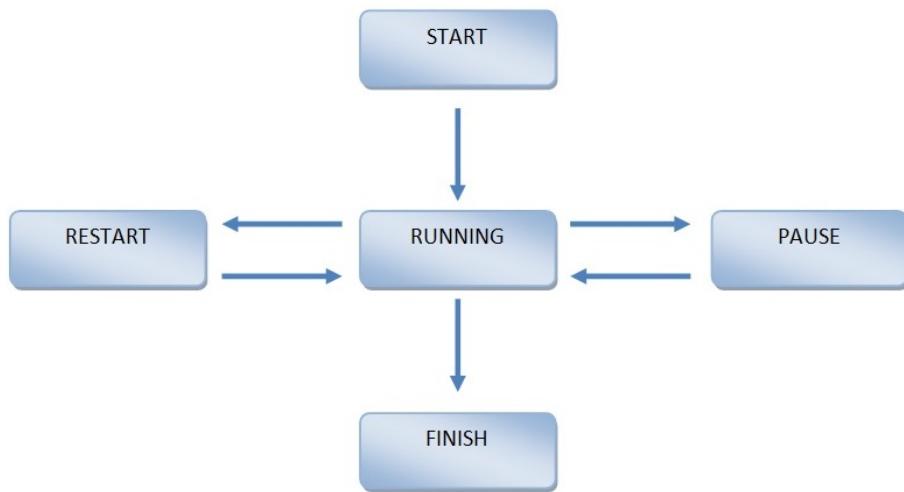
Es posible usar el trackball para mover al Pacman, siempre y cuando el smartphone disponga de uno. Además es posible pausar el videojuego mediante el botón menú del móvil. Cuando el videojuego esté en pausa, cambiará la tonalidad a tonos sepia como se puede ver en la siguiente imagen.



PacmanRenderEngine

Ahora que comprendemos cómo es posible dirigir el comportamiento del Pacman, hemos de centrarnos en la clase PacmanRenderEngine. Esta clase se adapta a las necesidades concretas del videojuego, extendiendo el GameEngine e implementando las operaciones de preRender mencionada en el capítulo anterior.

Antes de cada pasada del bucle principal del juego, se ejecuta la operación `preRender`, sobre la cual se ha implementando la siguiente máquina de estados:



El estado Start

Este estado transcurre desde que se inicia el nivel hasta que la cámara inicial o cámara 1, ha completado su recorrido. Realiza un acercamiento hasta la posición del Pacman, junto con una ligera inclinación, pudiendo apreciar las tres dimensiones del juego. Una vez completado el movimiento de la cámara se cambia al estado Running.

El estado Running

En este estado los elementos del juego empiezan a moverse sobre el escenario, la cámara ha sido sustituida por la cámara 3, que persigue al Pacman. Tras cada iteración se comprueba si existen colisiones con los distintos elementos del juego actuando en consecuencia.

Si la colisión se produce sobre uno de los fantasmas, se invoca al motor de sonidos para ejecutar el sonido `death`, se quita una vida al jugador y el estado se cambia a `Restart`.



Si la colisión es con una pastilla, implica modificar el marcador, ejecutar el sonido chomp y actualizar el elemento PillsNetElement, para que no renderize la pastilla. Finalmente se ha de verificar si existe alguna otra pastilla sobre el escenario. En el caso de haber sido la última, se procede a cambiar al estado Finish.

El estado Restart

Al iniciar este estado, se modifica la cámara para que realice un efecto de giro de 360 grados. La duración de este estado coincide con la duración de la animación de dicha cámara. Al finalizar la animación, si el usuario aún continúa teniendo vidas, cambia al estado Start, en caso contrario, la partida debe finalizar, yendo al estado Finish.

Antes de poder volver al estado Start será necesario volver a reiniciar las posición tanto del elementos Pacman como de los fantasmas que componen la escena.

El estado Pause

Durante este estado Running es posible que el usuario presione la tecla de pausa, cambiando de forma automática al estado Pause. Al volver a presionar sobre la tecla de pausa, vuelve automáticamente al estado de Running.

Durante este estado, el motor de sonido ha sido parado y se produce un cambio en un parámetro del render, modificando la iluminación de la escena a tonos sepia. Estos cambios son revertidos al salir del estado.

El estado Finish

Este es el último estado del escenario mediante el cual se cierra la actividad, volviendo a la actividad StartGame, la cual atendiendo a la forma en que finalizó el escenario, decidirá la siguiente acción, como ya se explicó previamente.

Comunicación entre actividad y render

Por último, cabe resaltar la problemática debida a que la actividad StartLevel es ejecutada en un thread principal y el renderer de OpenGL en otro. Al actualizar desde el método preRender, la puntuación y el número de vidas de la partida en pantalla, se produce un error de privilegios, provocado por estar accediendo desde el thread de OpenGL. Para evitar esta situación, se envían mensajes entre los distintos thread usando la clase **Message** de Android. Cuando el thread de la actividad recibe dicho mensaje, accede a la clase GameContext para recuperar la puntuación y el número de vidas, para después, actualizarla en pantalla.

7.8. Actividad StartLevel

Para que la actividad detecte el mensaje, ha de contener una propiedad pública de tipo Handler, a partir de la cual, el thread de OpenGL, enviará el evento.

Envió del mensaje desde la clase PacmanRenderEngine

```
1 private void updateGameBar() {  
2  
3     Thread update = new Thread(new Runnable() {  
4  
5         @Override  
6         public void run() {  
7             Message msg = new Message();  
8             activity.handler.sendMessage(msg);  
9         }  
10    });  
11    update.start();  
12 }  
13 }
```

Captura del mensaje en la clase StartLevel

```
1 public Handler handler = new Handler() {  
2  
3     public void handleMessage(Message msg) {  
4         updateLifesAndScore();  
5     }  
6 };  
7  
8 public void updateLifesAndScore() {  
9  
10    scoreView.setText(  
11        "Puntuacion: " + GameContext.getScore());  
12  
13    int lifes = GameContext.getLifes();  
14  
15    life1.setVisibility(  
16        lifes > 0 ? View.VISIBLE : View.GONE);  
17  
18    life2.setVisibility(  
19        lifes > 1 ? View.VISIBLE : View.GONE);  
20  
21    life3.setVisibility(  
22        lifes > 2 ? View.VISIBLE : View.GONE);  
23 }
```


CAPÍTULO

8

Conclusiones y líneas futuras

La realización de este proyecto de fin de carrera no habría sido posible pensando en él como un mero trámite para finalizar la carrera. De haber sido así, podría haber presentado cualquiera de los proyectos en los que he estado involucrado en mi vida profesional, por ejemplo el sistema de gestión de recursos aeroportuarios de la Terminal 4 de Barajas, el sistema de gestión de reclamaciones de una empresa de seguros, sistema de cámaras de vigilancia... Sin embargo, quería hacer algo distinto, fuera del ámbito de las aplicaciones de gestión, un proyecto que satisfaciera mis curiosidades.

Estar trabajando me ha permitido afrontar el proyecto, con tranquilidad, sin una fecha límite, permitiéndome dedicar el tiempo suficiente para asimilar los complejos conocimientos necesarios para realizar un proyecto de estas características. He partido de cero en lo referente al desarrollo de videojuegos, aunque con una base sólida adquirida durante la licenciatura.

Tras finalizar el proyecto se ha logrado adquirir un sólida base en el desarrollo de videojuegos. Esta situación ha permitido codificar la librería *TfcGameEngine*, de ámbito general, para el desarrollo de videojuegos. También se ha desarrollado con éxito una prueba de concepto de la librería. Esta prueba ha consistido en desarrollar el videojuego *Pacmania*, el cual ha sido probado de forma satisfactoria en varios smartphones y tablet. Para alcanzar estos ambiciosos objetivos se ha tenido que llevar a cabo una larga fase de documentación sobre cómo desarrollar videojuegos.

Durante el desarrollo del proyecto han existido innumerables complicaciones, las más destacables, por enumerar algunas son:

- Dificultad para realizar debug sobre la aplicación. El bucle principal del juego se ejecuta varias veces por segundo y un fallo ocurrido en la enesima iteración puede haberse solventado en siguiente iteración, dificultando la detección del fallo.



Capítulo 8. Conclusiones y líneas futuras

- La programación de *shaders* con GLSL es bastante rudimentaria. Cualquier fallo en su codificación puede llevarnos a renderizar imágenes totalmente en negro.
- El modelo de información del videojuego puede ser modificado por el thread de OpenGL y el thread principal de la aplicación. Por lo que ha sido necesario la creación de zonas de exclusión mutua.
- La comunicación con la interfaz gráfica de Android desde el thread de OpenGL no está permitida, provocando una excepción. Para resolver esta situación se ha tenido que recurrir al paso de mensajes.
- El emulador que viene incorporado en el kit de desarrollo de Android no soporta correctamente OpenGL ES 2.0, por lo tanto, ha sido necesario disponer de un smartphone para realizar cualquier tipo de prueba.
- El recolector de basura de la versión Froyo de Android se ejecuta bruscamente, paralizando la ejecución de las aplicaciones durante un tiempo razonable para cualquier aplicación que no sea un videojuego, en el cual se pierde la suavidad del movimiento. La versión Gingerbread, que no estaba disponible de manera oficial para el smartphone destinado al proyecto, resuelve esta problemática. La solución ha sido rootearlo e instalar una versión no oficial llamada Cyanogen.
- Problemas de rendimiento debidas al alto número de invocaciones a la tarjeta gráfica para dibujar las pastillas existentes en el escenario, reduciendo de forma considerable el número de imágenes mostradas por segundo. La solución a esta situación se ha basado en la creación de una única malla sin caras, sólo vértices y cada uno de ellos se correspondía a una pastilla. También ha sido necesaria una estructura de datos adicional en la cual se identificaba qué pastillas eran visibles y cuáles no. Toda esta información es analizada por un shader, que la procesa mostrando por cada vértice visible una partícula del tamaño de la pastilla.

Actualmente el videojuego soporta un número de frames aceptable, aunque no es lo suficiente rápido para videojuegos con elementos de un mayor nivel de detalle. Una línea futura para optimizar el rendimiento sería la implementación de un algoritmo octree, aún así no sería comparable con los framework de desarrollo de videojuegos actuales. Estos framework utilizan NDK, un kit de desarrollo de Android que permite construir librerías compartidas para poder llamar desde Java a código nativo C o C++, lo que nos permite una mayor realizar una mejor gestión de la memoria, evitando el recolector de basura de Dalvik.

La librería desarrollada contiene las funcionalidades básicas para crear un videojuego, por lo que existen múltiples funcionalidades que pueden ser añadidas con mayor o menor dificultad. Las más destacables y que podrían ser en sí un proyecto de fin de carrera son las siguientes:

-
- Usar las funciones físicas y de detección de colisiones que han sido desarrolladas en el proyecto hasta lograr un modelo físico, que permita simular la gravedad, rozamientos entre objetos y uniones entre ellos por algún tipo de junta, como por ejemplo bisagras.
 - Crear una herramienta que procesando la información de la malla, almacenada en el formato *Wavefront*, genere una forma implícita simplificada, permitiendo realizar un cálculo de colisiones óptimo.
 - Generación de animaciones de los elementos mediante armaduras en lugar de secuenciar mallas. El concepto de armadura se describe en la página 40.
 - Desarrollar *shaders* que calculen en tiempo de ejecución la iluminación de la escena, utilizando modelos de iluminación, en lugar de calcularla previamente mediante la técnica de *texture bake*. Los conceptos de modelo de iluminación y *texture bake* son descritos en las páginas 43 y 141 respectivamente.
 - Desarrollo de un modelo de red que permita jugar online varios jugadores.
 - Incluir funciones de mapeo con texturas, como por ejemplo *normal mapping* y *displace mapping* descritas en la página 39.



Bibliografía

- [1] Vladimir Silva. *Pro Android Games*. Apress, 2009.
- [2] Mark Murphy. *Beginning Android*. Apress, 2009.
- [3] Satya Komatineni, Dave MacLean, and Sayed Hashimi. *Pro Android*. Apress, 2009.
- [4] *Hello, Android*. Burnette, 2009.
- [5] Reto Meier. *Professional Android Application Development*. Wrox, 2009.
- [6] Dave Astle and Dave Durnil. *OpenGL ES 10 Game Development*. Premier Press, 2004.
- [7] Aaftab Munshi. *OpenGL ES 2.0 Programming Guide*. Addison Wesley, 2008.
- [8] Dave Shreiner. *OpenGL Programming Guide 7th*. Addison Wesley, 2009.
- [9] David Bourg. *Physics for Game Developers*. Gameinstitute, 2009.
- [10] Dave Shreiner. *OpenGL Shading Language*. Addison Wesley, 2009.
- [11] Ian Millington. *Game Physics Engine Development*. Elsevier, 2010.
- [12] Christer Ericson. *Real time collision engine*. Elsevier, 2004.
- [13] James D. Foley, Andries Van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics - Principles and Practice 2ed in C*. Addison Wesley, 1996.
- [14] *Programming Game AI by Example*. WordWare, 2005.
- [15] Mat Buckland. *Artificial Intelligence for Games*. Elsevier, 2005.
- [16] *Beginning OpenGL Game Programming*. Course Technology PTR, 2009.
- [17] Luke Benstead. *Data Structure For Game Programmers*. Premier Press, 2002.

- [18] David Wells. *Game Mathematics*. Gameinstitute, 2009.
- [19] Angelina Vallejo Fernández, David y Cleto Martín. *Desarrollo de videojuegos: Arquitectura del motor de videojuegos*. Universidad de Castilla-La Mancha, 2013.
- [20] Javier A González Morcillo, Carlos y Albusac Jiménez. *Desarrollo de videojuegos: Programación Gráfica*. Universidad de Castilla-La Mancha, 2013.
- [21] David Villa and Francisco Moya. *Desarrollo de videojuegos: Técnicas avanzadas*. Universidad de Castilla-La Mancha, 2013.
- [22] Francisco Jurado and Javier A Albusac, Jiménez. *Desarrollo de videojuegos: Desarrollo de componentes*. Universidad de Castilla-La Mancha, 2013.
- [23] Wikipedia. <http://www.es.wikipedia.org>.
- [24] Nate robbing. <http://user.xmission.com/~nate/opengl.html>.
- [25] Dossier sobre pacman. <http://home.comcast.net/~jpittman2/pacman/pacmandossier.html>.
- [26] Blender. <http://www.blender.org>.
- [27] Russell Smith. Open dynamic engine. <http://www.ode.org>.
- [28] Torus Knot Software. Ogre3d. <http://www.ogre3d.org>.
- [29] Khronos Group. Especificaciones de GLSLangSpec.Full.1.10.59. <http://www.opengl.org/documentation/glsl/>.
- [30] Wavefront. Especificaciones del formato obj. <http://www.fileformat.info/format/wavefrontobj/egff.htm>.
- [31] Wavefront. Especificaciones del formato mtl. <http://www.fileformat.info/format/material/>.

Parte IV

Apendices

Características del móvil HTC Dream

El HTC Dream, comercializado también como T-Mobile G1, es un dispositivo de telefonía móvil construido por HTC. Fue lanzado al mercado el 22 de octubre de 2008, a un costo estimado de 179 Dolares U.S.A., siendo, según declaraciones de la empresa,¹ el primer dispositivo móvil de comunicación en incorporar el Sistema operativo móvil de Google denominado Android.

Característica	Descripción
Dimensiones	117 x 55.7 x 17.1 milímetros
Peso	158 gramos
Procesador	Qualcomm MSM7201A 528 Mhz
Memoria RAM	192MB
Memoria ROM	256 MB
Pantalla	3.2 pulgadas
Resolución	WVGA (320 x 480 pixels)
Cámara	3.15 MP
Vídeo	No
Batería	1150 mAh
Red	SPA/WCDMA y MHz y GSM
Conectividad	GPRS, EDGE
Autonomía 3G	406 horas en espera y 5 horas 20 minutos
Bluetooth	Sólo para auriculares
GPS	Si
Radio	No
Jack 3.5mm	Sí
Micro USB	No
Acelerómetro	Sí
Brújula digital	Sí
Sensor de proximidad	No
Sensor de luz ambiental	No
Puerto inflarojos	No

Características del móvil HTC Desire

En Enero de 2010 la compañía HTC saca al mercado un nuevo smartphone, en él se ejecuta la versión 2.2 del sistema operativo Android. El desarrollo del proyecto se ha centrado en este móvil debido a que es uno de los primeros móviles con un procesador Snapdragon de 1Ghz, que contiene un chipset gráfico Adreno 200, soportando la programación de la GPU (unidad de procesamiento gráfico) mediante la inclusión de shaders.

Característica	Descripción
Dimensiones	119 x 60 x 11,9 milímetros
Peso	135 gramos
Procesador	Procesador Qualcomm Snapdragon a 1 GHz
Chipset gráfico	Adreno 200 con soporte OpenGL ES 2.0
Memoria RAM	576MB
Memoria ROM	512MB hasta 32GB
Pantalla	AMOLED multitáctil de 3.7 pulgadas
Resolución	WVGA (480 x 800 píxeles)
Cámara	5MB
Vídeo	Alta definición 720p
Batería	1400 mAh
Red	SPA/WCDMA y MHz y GSM
Conectividad	3G, GPRS, EDGE y WiFi (802.11b/g)
Autonomía 3G	360 horas en espera y 6 horas 30 minutos
Bluetooth	Con soporte A2DP
GPS	Con soporte A-GPS
Radio	Sí
Jack 3.5mm	Sí
Micro USB	Sí
Acelerómetro	Sí
Brújula digital	Sí
Sensor de proximidad	Sí
Sensor de luz ambiental	Sí
Puerto inflaros	No



Figura 8.1: Dispositivo móvil HTC Dream



Figura 8.2: Dispositivo móvil HTC Desire

