UNIVERSIDAD POLITÉCNICA DE VALENCIA

Proyecto Final de Master

Librería multiplataforma para el desarrollo de aplicaciones 3D sobre OpenSceneGraph

Autor: Jorge Izquierdo Ciges
Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital
Dirigido por: Javier Lluch Crespo

Instituto de Automática e Informática Industrial (AI2)
Universidad Politécnica de Valencia
Camino de Vera, s/n
46020 Valencia, Spain

10 de septiembre de 2013

Palabras Clave

Gráficos por computador, Android, dispositivos embebidos, smartphones, OpenSceneGraph, OSG, grafo de escena, representación de escenas tridimensionales, representación de terrenos, GIS, 3D.

Resumen

El desarrollo de aplicaciones y programas multiplataforma, tanto de caracter comercial como para estudios científico es un problema complicado desde el punto de vista del coste económico y temporal que supone ajustar cada aplicación o cada test a la miríada de plataformas existentes en la actualidad.

En este trabajo se presenta el framework mWorld, un framework multiplataforma orientado a la creación de aplicaciones gráficas en Android, Windows, Linux e iOS. La filosofía del framework consiste en la programación basada en aplicaciones, módulos y plugins independientes, simplificando el desarrollo al realizar el código de la manera más genérica posible, y trasladando la complejidad del sistema operativo a pequeñas cajas cerradas que se cambian dependiendo del sistema operativo . Esto permite la simplificación de las complejidades del desarrollo multiplataforma, a la vez que permite reutilizar diferentes librerías open-source en los sistemas operativos donde estén disponibles, y la implementación de código especifico solo en aquellos sistemas donde no haya ninguna alternativa disponible. El renderizado de objetos tridimensionales de este framework se realiza con un plugin basado en la librería OpenSceneGraph.

Durante este trabajo se presenta el esquema de funcionamiento del framework, las pruebas de funcionamiento realizadas y, como ejemplo práctico, el uso del framework mWorld para la generación de una aplicación multiplataforma que se comunica con servicios WMS y representa capas de terreno geolocalizado conviviendo con el ecosistema de cada una de las plataformas y dentro de los límites de memoria de los dispositivos móviles. Finalmente se analizarán los resultados y se presentarán las conclusiones del trabajo.

Índice general

Índice de figuras

Índice de tablas

1

Introducción

En la última década se ha producido un salto cualitativo en las capacidades del hardware móvil. Los avances en el desarrollo del hardware, su capacidad gráfica, y el acceso a internet los ha situado en un lugar prominente de la sociedad. Teléfonos inteligentes, tabletas, mini portátiles o híbridos que combinan características de estos dispositivos; conviven, con mayor o menor éxito, con el hardware de sobremesa.

La gran acogida de todos estos dispositivos "inteligentes" por la sociedad 2.0, la sociedad de la información y del acceso a redes sociales, ha conseguido que, cada vez más, el desarrollo de aplicaciones para estos dispositivos forme parte de los proyectos de software.

Para abarcar la mayor cantidad posible de plataformas sin encarecer el coste de los proyectos se ha optado por una serie de soluciones: Aplicaciones en la nube, aplicaciones en forma de página web en el propio dispositivo o los frameworks multiplataforma. Cada una de estas alternativas tiene una serie de ventajas e inconvenientes.

Las aplicaciones en la nube requieren de una infraestructura importante para manejar las peticiones de múltiples clientes, ya que a medida que los requisitos de la aplicación aumentan, los costos de computación y transmisión de datos lo hacen proporcionalmente. Finalmente, dado que el acceso a estos servicios se realiza mediante un navegador, al igual que las aplicaciones web locales, se ha de limitar la complejidad visual y lógica de la representación para evitar problemas en la ejecución, ya que es

normal encontrar pérdidas de rendimiento al ser código que se interpreta en tiempo de visualización.

Finalmente los frameworks multiplataforma tienden a tener un mayor coste para programar aplicaciones. Sin embargo tienden a obtener mejores resultados para aplicaciones intensivas, ya que se benefician de implementaciones a bajo y medio nivel en la propia plataforma sin emplear tecnologías intermedias que provoquen retrasos.

En este trabajo se presenta el framework mWorld, un framework multiplataforma orientado a la creación de aplicaciones gráficas en Android, Windows, Linux e OSX. Para ello se emplea, como dependencia, el grafo de escena OpenSceneGraph.

El origen de este proyecto se encuentra en los trabajos realizados en el Instituto de Automática e Informática Industrial (ai2) en el grupo de desarrollo de la extensión 3D de gvSIG, donde se realizaron los primeros trabajos de desarrollo multiplataforma y se realizó la planificación y los primeros diseños de este framework. Posteriormente, por falta de continuidad presupuestaria, este trabajo se ha seguido realizando en colaboración con la empresa Mirage Technologies.

Este proyecto realizado durante el último año y medio da continuidad al proyecto final de carrera: Representación interactiva de escenas tridimensionales con OpenSceneGraph en Android [Cig11]. Lo que ha permitido el mantenimiento de la librería OpenSceneGraph para Android y el desarrollo de nuevos procesos para su uso y aprovechamiento en diferentes tipos de proyectos comerciales y científicos.

Finalmente, en base a este trabajo, se ha realizado una aplicación para estudiar el rendimiento del framework. Esta aplicación realiza la carga en tiempo de ejecución de datos no procesados previamente desde servidores WMS.

La presente memoria consta de cuatro partes: En primer lugar, se expone una visión sobre el estado del arte de los diferentes elementos empleados a lo largo del trabajo, así como ciertas metodologías que se han empleado a lo largo del proyecto.

La segunda parte comprende un análisis de la problemática que supone la creación del framework mWorld, así como los objetivos propuestos para su desarrollo.

Seguidamente, se realiza una exposición del trabajo realizado, resaltando las partes más importantes que se han desarrollado del framework.

Posteriormente se presentan los resultados obtenidos durante el desarrollo del proyecto, y para finalizar se presentan las conclusiones y las líneas de desarrollo futuro a partir de este proyecto.

2

Antecedentes

Esta sección cubre el estado actual del arte sobre el que se apoya el trabajo. Primero se hablará de la evolución de dos elementos muy importantes en este trabajo, OpenGL y Android. Seguidamente hablaremos de uno de los pilares troncales de este trabajo, OpenSceneGraph (OSG) y una serie de herramientas y conceptos que se han empleado en este trabajo. Hablaremos de algunas alternativas a OSG y, a continuación se tratará la evolución histórica del renderizado en dispositivos embebebidos. Finalmente se tratará el protocolo de servicio de mapas en la web, WMS y una serie de librerías que se han empleado en el proyecto: GDAL y cURL.

2.1. OpenGL

OpenGL [Khr92] es una API diseñada por el Kronos Group, una asociación sin ánimo de lucro que se dedica a la generación de diferentes estándares multiplataforma. En la actualidad, el mantenimiento y la creación de nuevas versiones de la API OpenGL está a cargo de la OpenGL ARB que está integrado por una serie de empresas de CAD, software y hardware entre las que se encuentran: Nvidia, Amd, Apple, Google, id Software, entre otras.

Desde su creación se han realizado diversas modificaciones del estándar para adaptarse a la tecnología presente en cada momento, así como una serie de sub-especificaciones

para adaptarse a dispositivos embebebidos (OpenGL ES) y sistemas críticos (OpenGL SC). Aunque las características de cada versión han variado en mayor o menor medida, la API ha conservado dos características muy importantes a lo largo de su recorrido:

- La interacción con la tarjeta se realiza usando un modelo cliente-servidor.
- Un sistema de extensiones

El programador gráfico no ejecuta su programa directamente sobre la tarjeta gráfica, sino que el código binario del programa establece una comunicación cliente-servidor con el sistema gráfico correspondiente. Esta comunicación, se realiza independientemente de si el renderizado se efectúa en la propia máquina o en una externa. Esta concepción ha permitido que la API se pudiera emplear en numerosos tipos de arquitecturas que no tuvieran nada en común con la típica de un pc de sobremesa. También hay que señalar que, al comunicarse a bajo nivel con las tarjetas gráficas u otros sistemas de renderización, siempre ha ofrecido un rendimiento superior a otros sistemas como DirectX [Mic92] que requieren de diversos pasos y llamadas a través del kernel del sistema operativo.

Por otro lado, el sistema de extensiones permite emplear funcionalidades específicas del hardware. OpenGL es un estándar y, como tal, solo puede ofrecer una base común. Para mantener una compatibilidad entre todos los hardwares que lo utilizan. Sin embargo, el hardware suele avanzar más rápido que la generación de estándares. Aquí es donde entra el sistema de extensiones; permitiendo al programador consultar en tiempo de ejecución la existencia de nuevas funciones, o de funciones exclusivas para ese hardware que sean de utilidad para el programador. Todo esto sin necesidad de modificar la API cada vez que se añadían extensiones, y a la vez manteniendo la compatibilidad con el hardware ya existente.

Desde su nacimiento, OpenGL no fue concebido para un lenguaje específico. La implementación oficial, está pensada para un lenguaje de tipo imperativo. Y es sobre la que está basada la mayor parte de documentación, que está realizada en ANSI C99. Existen versiones para lenguajes no imperativos que se limitan a enmascarar la imperatividad intrínseca en la comunicación de elementos a representar que realiza OpenGL con el sistema de renderizado. Por otro lado, no se dispone de una versión de la implementación que siga el paradigma de la orientación a objetos. No existe una versión especifica para C++ o bien se emplea la implementación en C o se emplea una librería en C++ que encapsule la funcionalidad. Este es el mismo caso de los bindings que funcionan, en la actualidad, para el lenguaje Java.

El proceso por el cual se genera una representación a partir de datos geométricos. Se suele llamar "tubería". Este nombre se debe al diseño en forma de cadena de producción que tiene la API. Esta cadena de producción recibe en un extremo una serie de datos que se procesan en diversas etapas o fases, en las cuales el programador no puede intervenir físicamente. Para cada fase del proceso, el programador puede ajustar, previamente, una serie de parámetros definidos en el estándar. Estos, modifican el funcionamiento de cada fase de procesado dentro de la tubería.

Este diseño proviene de las estaciones de renderizado que se empleaban en los orígenes de la representación gráfica en computadores. Ese carácter de fijo e inamovible es el que bautiza a este proceso como "Tubería de procesado fija". La figura: 2.1 muestra un resumen de los procesos que se realizaba sobre los datos desde su introducción hasta su representación.

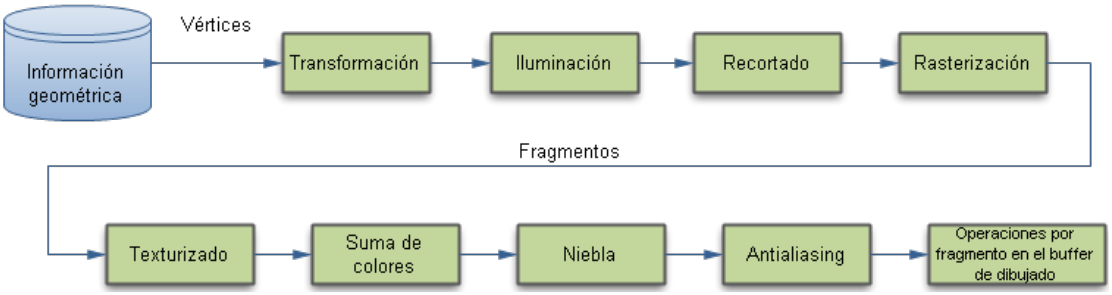

Figura 2.1: Diagrama de los procesos de la tubería de procesado fija en OpenGL.

Durante los años noventa, OpenGL adquirió una posición de ventaja competitiva en detrimento de alternativas como DirectX. Esta posición se debe, sobre todo, a la constante evolución que permitía el mecanismo de extensiones en OpenGL, que permitía ofrecer a los desarrolladores las características que todavía no habían sido integradas de forma fija en el estándar.

En 2008, se lanzó la versión 3.0 del estándar que rompía con la filosofía de compatibilidad de las aplicaciones, ofreciendo dos versiones del estándar a la vez como "perfiles". Un perfil "core" y uno "compatibility". Esta dualidad también sigue existiendo en la versión 4.X del estándar.

El perfil núcleo, se queda con un subconjunto de comandos y estados prescindiendo de los métodos más ineficientes. Los métodos eliminados en este perfil, se pueden seguir empleando pero aparecen marcados como deprecados, lo cual obliga a emplear

el perfil de compatibilidad. Los métodos marcados como deprecados están considerados como métodos que se pueden eliminar en un futuro y como tales no deben usarse si se quiere garantizar el uso futuro del código de un programa o librería. Todo programa basado en el perfil núcleo debe cumplir obligatoriamente con el uso exclusivo de la tubería programable2.1, desapareciendo el uso intermedio que permitía la versión 2.1 de OpenGL.

La tubería programable es una manera diferente de procesar la información geométrica del usuario. La idea, detrás del uso de esta tubería, es que el programador pueda controlar el procesado de sus datos en determinadas partes de la secuencia de forma completa, no solo sobre unos valores prefijados. Para ello, la API expone una serie de procesos en los cuales, el programador, puede cargar código que se ejecutará en el renderizado.

Estos programas, denominados shaders, se ejecutan en paralelo en las unidades de procesamiento y aceptan una gran variedad de tipos de datos de entrada. Actualmente, existen tres tipos de shaders:

- Vertex Shader - Afectan el procesado por cada vértice.
- Geometry Shader - Afectan a nivel de primitiva geométrica de dibujado.
- Fragment Shader Afectan a nivel de píxeles visibles.

Recientemente se ha incorporado un tipo de shader genérico para el procesado de datos general. Se les denomina “Compute Shaders”. Este tipo de shaders se ha introducido con el objetivo de acercar la programación paralela sobre tarjetas que ya se realiza en OpenCV. Así se pretende el uso de procesos de cálculos paralelos sin necesitar de salir de la API de OpenGL.

El uso de los programas shaders ha supuesto un aumento en las capacidades de decisión para los programadores gráficos permitiendo la implementación de nuevas técnicas y efectos que no se podían representar con las limitaciones de la tubería fija.

2.1.1. OpenGL ES 1.X

OpenGl Embebed Systems es la respuesta de la Kronos ARB a las necesidades de algunos miembros del consorcio como PowerVR [Ima92] para llegar a una API estándar en dispositivos embebidos o de recursos limitados. Esta versión busca permanecer lo más cerca posible del estándar de sobremesa ajustándose a las limitaciones

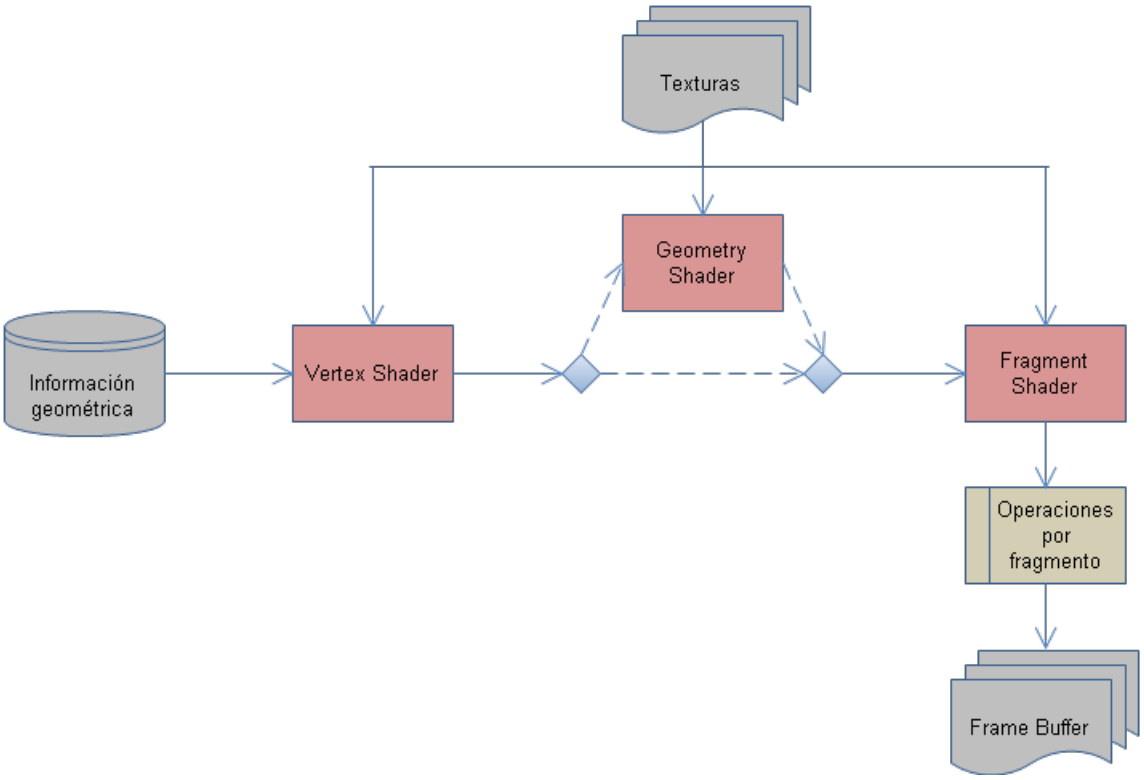

Figura 2.2: Diagrama de los procesos de la tubería de procesado programable.

de memoria o potencia. Por ello en realidad OpenGL ES 1.X se creó como un subconjunto interoperable con la definición para sobremesa. A lo largo de las diferentes versiones de OpenGL se han ido introduciendo una serie de métodos que replicaban funciones ya existentes pero con una mayor eficiencia. Para mantener la compatibilidad entre diferentes versiones, hasta la aparición de los perfiles en OpenGL 3.0, no se eliminaba ninguna funcionalidad. La sobrecarga de métodos similares supone que los drivers necesitan soportar más elementos y, por lo tanto, tienden a consumir muchos más recursos de memoria.

Por esta razón, en OpenGL ES, encontramos una simplificación que elimina las versiones más primitivas y lentas de envío de datos geométricos. Entre otros métodos, se prescinde de las instrucciones por vértices y de las listas de comandos, conservando únicamente los Vertex Array, que ofrecen un mayor rendimiento de cara al envío de datos y su procesado. También se prescinde de algunas instrucciones de consulta al driver gráfico, como serían las relativas a las matrices. Este tipo de instrucciones generan un gran cuello de botella para su escasa utilidad final. En general, desaparecen

los comandos que tienden a ser más lentos y que realizan funciones de apoyo cuyo coste computacional no compense su uso, así como algunas que se han considerado "funciones en deshuso" por la mayoría de la comunidad.

En resumen, el programador encuentra una API compatible con su hermana de sobremesa, OpenGL 1.4 que siguiendo sus líneas, emplea la tubería fija para realizar el tratamiento de los datos geométricos. De igual forma, el estándar de dispositivos embebidos mantiene un sistema de extensiones para incluir funcionalidades, que no estuvieran disponibles en todos los dispositivos, como es el caso de las "Ambient Box", sin necesidad de cambiar la API.

Hay que destacar que la versión 1.0 del estándar impone una serie de restricciones sobre las texturas. Se obliga a que tengan que ser cuadradas y potencias de dos. Esta restricción fue relajada en la versión 1.1. e incluso antes, dado que existían algunas extensiones que permiten usar texturas que no cumplan la restricción de tamaño y forma.

Por último, hay que comentar una limitación que existía en OpenGL ES 1.0 ya que aunque se podía emplear aritmética de coma flotante para los gráficos, su uso ralentizaba mucho el renderizado de resultados. Se recomendaba, como buena praxis, el uso de números decimales en coma fija.

2.1.2. OpenGL ES 2.0

La versión 2.0 de la especificación rompe totalmente la compatibilidad con la versión 1.X. Y, aunque existen muchas similitudes con la versión 2.1 y 3.0 del estándar de sobremesa, existen algunas diferencias para reutilizar el código en sobremesa sin modificaciones. Esta versión continua la idea de mantener la especificación lo más contenida y reducida posible, por ello, en vez de optar en sistemas de perfiles o de esquemas obliga al programador a ajustarse totalmente a la nueva especificación.

Esta versión introduce la tubería de procesado programable de una manera similar a la que se presenta en el perfil núcleo de la versión 3.0 de sobremesa. Si bien, aquí el estándar elimina totalmente el uso de la tubería fija y se emplea el esquema de la figura: 2.1.2.

Entre otras características notables, se incluye la desaparición de los límites de tamaño en las texturas y el uso, únicamente, de dos tipos de shaders: (1)Vertex Shaders y (2)Fragment Shaders. El lenguaje de especificación de los Shaders sigue la misma especificación que en sobremesa, GLSL, si bien existen algunas diferencias y limitaciones

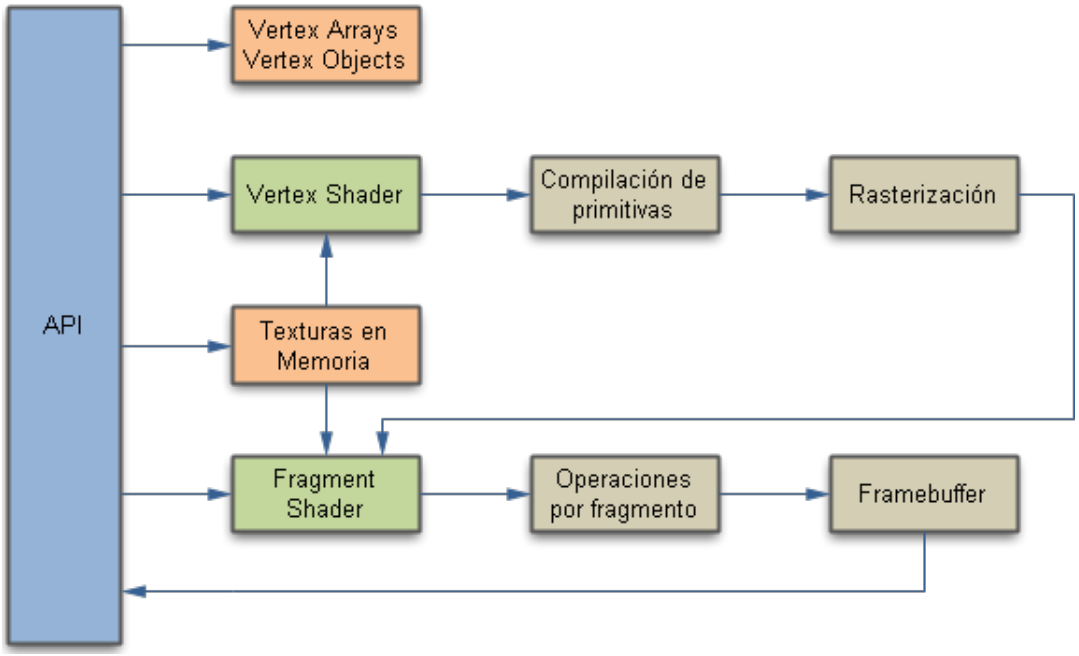

Figura 2.3: Diagrama de la tubería de proceso en OpenGL ES 2.0.

como la necesidad de incluir la especificación de precisión de cada variable utilizada en un programa shader.

2.1.3. OpenGL ES 3.0

Recientemente se ha introducido una nueva especificación estándar, la versión 3.0, que mantiene compatibilidad con la versión 2.0 y además introduce una serie de capacidades que están presentes en las últimas versiones de la versión de sobremesa. Como es posibilidad de realizar múltiples renderizados en texturas a la vez, el uso de instanciación múltiple, etc.

2.1.4. WebGL

WebGL es la especificación diseñada por la Kronos ARB para el desarrollo de programas gráficos en navegadores web. Su objetivo es la utilización de la aceleración hardware para la visualización y representación de páginas web con contenido tridimensional. WebGL define un enlace entre JavaScript y la API OpenGL ES 2.0, la cual debe estar implementada en el navegador. De esta manera, un programa WebGL es capaz de presentar aplicaciones tridimensionales que empleen la tubería de procesado

programable, siendo capaces de enviar código ejecutable a la tarjeta gráfica.

En la actualidad, está soportado por los navegadores: Firefox, Safari, Chrome y Opera, también a través de la librería webKit, lo cual permite su empleo en dispositivos convirtiéndose en una alternativa a la propia versión embebida de OpenGL. A pesar de la aceptación de gran parte de la comunidad como una forma de realizar programas con representación gráfica independiente del sistema operativo, ha sido rechazado por algunos sectores de la industria. En Junio de 2011 Microsoft Security Research & Defense (MSRC) publicó que consideraban la API como "dañina" basándose en los informes de la empresa Context Information Security [For11] [FSJ11], dichos informes encuentran una serie de debilidades que se podrían explotar de forma maliciosa por parte de programadores. Sobretodo, se señalan los siguientes tres problemas:

- El soporte de WebGL expone de forma directa el hardware sin necesidad de permisos.
- WebGL confía toda la responsabilidad de seguridad en las implementaciones independientes. Se pueden presentar agujeros de seguridad dependiendo de la implementación independientemente de la API.
- Se presenta una nueva problemática con ataques de denegación de servicio (DoS) mediante programas gráficos.

La infraestructura de hardware gráfico no está preparada para defenderse ante ataques debido a programas de ataque que empleen código gráfico para ello. Esto es debido a que históricamente los ataques en el lado del cliente no generaban grandes riesgos de seguridad, pero al tratarse de código que proviene de una web, es posible que una web provoque un ataque DoS a todos los clientes que la visitan. Incluso con estos problemas, WebGL se encuentra implementado en varios navegadores compatibles con HTML5 como Mozilla o Chrome. Los dispositivos Apple, por su parte, no lo soportan de forma primaria en su navegador y emplean un navegador que solo deja acceder a sitios específicos para evitar los problemas de seguridad.

2.2. Android

Android es un sistema operativo libre basándose en el Kernel de Linux. Está diseñado para ser empleado con dispositivos embebidos. Sus orígenes comienzan en la empresa Android,Inc que posteriormente fue comprada por Google. Posteriormente,

este sistema operativo es apadrinado por la OpenHandsetAlliance(OHA). La OHA es un consorcio de diversas empresas que se han unido para crear estándares libres para normalizar y garantizar una uniformidad y compatibilidad en el desarrollo de dispositivos embebidos, móviles inteligentes, etc.

En la década de los noventa, las mejoras en la capacidad de procesamiento, invitó a crear dispositivos móviles con mayores capacidades. Debido al rápido desarrollo de estos, las compañías lanzaban un gran número de modelos con grandes diferencias de hardware y sistemas operativos. Esto provocó una enorme fragmentación en el mercado complicando, enormemente, el desarrollo de aplicaciones en este tipo de hardware. Sin ningún tipo de estándar o sistema operativo de referencia cada aplicación tenía que ser diseñada ex profeso para el dispositivo. Con el paso del tiempo y forzados por los costes, las compañías comenzaron a incluir una serie de estándares de facto en el mercado. OpenGL, por ejemplo, se convirtió en el estándar de facto después de un largo periodo en el que convivió con una serie de estándares menores.

Con Android se intentaba ofrecer una posibilidad de crear un sistema operativo totalmente libre que se convirtiese en un estándar. La ventaja de este concepto es su utilidad tanto para desarrolladores como para las empresas que fabrican el hardware. Por un lado, los desarrolladores pueden abarcar una mayor cuota de mercado, en tanto que muchos dispositivos diferentes comparten un mismo sistema operativo, y en el lado de los fabricantes de hardware, consiguen un ahorro y una competitividad mayor. Al tratarse de un sistema operativo libre creado con un diseño de capas, los desarrolladores únicamente tienen que adaptar los drivers para sus componentes, esto supone un gran ahorro de costos en el desarrollo y mantenimiento en un sistema operativo propio. Por otro lado, sus dispositivos poseen una competitividad igual o mayor de cara al consumidor, ya que las aplicaciones desarrolladas en Android no están ligadas a un modelo específico, así cualquier móvil dispone de un gran número de aplicaciones para satisfacer las necesidades del consumidor.

Una de las primera compañías en adoptar el sistema operativo fue HTC. El primer modelo que Google presentó para desarrollo fue el HTC Dream, con el paso del tiempo, más compañías se han unido a la iniciativa. Actualmente compañías como Motorola, LG, Samsung, Archos, Toshiba, Asus, Acer o Sony han incluido Android en sus dispositivos. En la actualidad, Android ha sobrepasado el medio millón de dispositivos nuevos activados diariamente y se estima una tasa de crecimiento de un 4.4 % semanal.

Android ha evolucionado con las necesidades que ha ido presentando el mercado, pasando de ser un sistema para teléfonos móviles, a estar integrado en dispositi-

vos multifunción (relojes, agendas, cámaras), consolas o tabletas. Aunque en la última versión (4.X) el sistema operativo está totalmente unificado para los diferentes tipos de dispositivos, Android ha llegado a tener una versión exclusiva para las tabletas: Gingerbread (3.X) que adaptaba el sistema operativo a un manejo diferente al de los smartphones al disponer de una mayor pantalla.

La última versión del sistema operativo, 4.3 (Jelly Bean), que no presenta grandes novedades en su núcleo, ha sido anunciada en la conferencia I/O de 2013. Las principales novedades son mejoras de seguridad, estéticas de funcionamiento en la plataforma, así como la posibilidad de emplear nuevos estándares de comunicación, posibilitando la creación de aplicaciones que utilicen la API 3.0 de OpenGL ES.

La programación en Android, gira alrededor de la máquina virtual Java Dalvik. Esta máquina virtual, mueve todas las aplicaciones y procesos de un dispositivo en diferentes zonas acotadas e independientes con el objetivo de obtener la mayor compatibilidad y seguridad posible. Aunque el lenguaje de programación nativo de la plataforma es Java/Dalvik, sin embargo, Google permite desde la versión 1.5 del sistema la creación de programas que accedan a la capa Nativa del S.O. mediante los lenguajes C y C++.

El desarrollo del presente trabajo emplea, en mayor medida la programación nativa que se ofrece en Android. En la conferencia [Gal11] se explica el uso, ventajas e inconvenientes del uso de la programación nativa. Aunque se han tenido que realizar partes que emplean la ejecución virtual de Dalvik. Para optimizar convenientemente los programas, se precisa controlar el uso de la memoria, la búsqueda de pérdidas de memoria, así como las condiciones de liberación del recolector de basura de Android. Todos estos temas se encuentran tratadas más ampliamente en [Dub11].

Las principales fuentes de documentación sobre la plataforma Android en las que se ha basado este proyecto son tres:

- La documentación propia de Google. [goo11b] [goo11a]
- Las conferencias de Google IO 2010 y 2011 [goo10] [goo11c]
- Foros y grupos oficiales del SDK y NDK de Android.

2.2.1. Fundamentos de la plataforma

En las bases de la OHA se declaran las diferentes ideas que sirven como base a Android.

- El sistema operativo debe ser abierto, los desarrolladores deben poder ser capaces de crear aplicaciones que empleen todas las características del dispositivo sin ninguna limitación.
- Todas las aplicaciones deben ser consideradas iguales. Es decir, que todas las aplicaciones puedan competir por el acceso a los recursos del dispositivo de forma ecuánime.
- Compartir información, que todas las aplicaciones sean capaces de intercambiar información y recursos.
- Desarrollo de aplicaciones simple y rápido.

Cumplir dichos requisitos no es una tarea simple. La primera condición, requiere de la existencia de permisos por parte del usuario, la segunda condición requiere de una planificación de procesos y de la memoria compartida, la tercera obliga que, además de los permisos de uso, exista un permiso que comunique los datos entre aplicaciones. El último requisito se debe entender desde el punto de vista que la tecnología de este mercado, como se ha comentado previamente, requiere de ciclos de desarrollos cortos, esto también debe de aplicarse al propio sistema operativo.

En contra de crear un Kernel interno propio, Android emplea el Kernel Linux. A día de hoy, desde la versión 3.0 de Android, la versión empleada es la: 2.6.36. El Kernel de Linux ha tenido un desarrollo a nivel de código que le ha dado una gran portabilidad. Existen versiones para una gran variedad de arquitecturas:

- DEC Alpha
- ARM
- AVR32
- Blackfin
- ETRAX CRIS
- FR-V
- H8
- A64
- M32R

- m68k
- MicroBlaze
- MIPS
- MN10300
- PA-RISC
- PowerPC
- System/390
- SuperH
- SPARC
- x86
- x86 64
- Xtensa.

El uso del Kernel Linux resulta de una gran ventaja, ya que no se necesita adaptar los mecanismos internos del sistema, únicamente se adaptan los controladores apropiados para los componentes de cada hardware. Actualmente muchos de los sistemas y chips que se emplean en los dispositivos embebidos también se encuentran en los ordenadores de sobremesa o compartidos por muchas compañías diferentes. Al final, las empresas que incluyen Android en sus arquitecturas únicamente se han de preocupar por configurar apropiadamente los drivers que, a su vez, preparan las empresas que han diseñado cada bloque de hardware. Todo esto, supone una reducción de costes que repercute en la competitividad.

Uno de los mayores aciertos del sistema operativo Android es conseguir una plataforma única que tenga una auténtica compatibilidad entre todas las empresas que lo acojan. Si bien la compatibilidad debería ser perfecta, ya que dado el estado de código abierto, hay compañías que pueden realizar modificaciones sobre el código de forma privada. Siempre existe una posibilidad de que una compañía realice cambios que separen e incompatibilicen su dispositivo con el resto.

La solución que aporta Android para asegurar la compatibilidad, es la implementación, como una parte esencial del sistema operativo, de una máquina virtual, Dalvik

VM. Esta es una máquina basada en registros y que ha sido optimizada para dispositivos con poca memoria, para ello dispone de una serie de características que la diferencian de otras máquinas virtuales:

- La tabla de constantes ha sido modificada para usar únicamente enteros de 32 bits.
- El juego de instrucciones emplea un formato de 16 bits que funciona directamente con las variables locales mediante un registro virtual de 4 bits. Estas mejoras están pensadas para reducir el coste de memoria por instrucción y su cantidad.

Otra característica notable de la máquina virtual, es que incorpora un recolector de basura para liberar la memoria que no se usa; sin embargo el concepto que sigue para la gestión de memoria es singular, Android intenta ocupar el máximo de memoria posible. La memoria que no se usa es memoria desperdiciada, por ello el recolector no intenta limpiar la memoria lo más rápido posible, únicamente lo hace cuando se necesita memoria para aplicaciones nuevas o con el mayor grado de importancia. La razón para este comportamiento es porque aunque una aplicación sea cerrada por el usuario, esta no desaparece de memoria ni es finalizada, esta permanece en segundo plano hasta que el propio sistema necesita los recursos que esa aplicación está ocupando. Este comportamiento se definió así porque en los dispositivos como los smartphones existe una serie de aplicaciones que son lanzadas una y otra vez por parte del usuario de forma muy habitual, por ejemplo, la agenda del teléfono es una aplicación que se ejecuta de forma habitual repetidamente. De esta forma cuando el usuario intenta volver a usarla tras una primera ejecución, si su móvil no ha empleado demasiados recursos en otras aplicaciones, la agenda seguirá en memoria con el ahorro que supone tener el código y los elementos de la aplicación ya precargados en la memoria listos para reanudarse.

Así pues las aplicaciones en Android tienen un ciclo de vida y una idea diferente a la aplicación de sobremesa. Una aplicación en Android tiene una serie de elementos ejecutables llamados actividades. Las actividades son partes de una aplicación con su propia interfaz gráfica, salvo si se especifica de forma diferente, las actividades son módulos pseudo independientes que encapsulan la funcionalidad de diferentes partes de la aplicación. Una actividad incluye, además del funcionamiento, la interfaz gráfica. Cada actividad es capaz de llamar a otras actividades, de esta manera, se puede pasar a un diseño de actividades donde cada una de ellas, de forma independiente, realiza una parte funcional de una aplicación. También se permite en el sistema Android llamar a actividades que ya se encuentran en el dispositivo para realizar funciones que

el programa no tiene incluidas. Un ejemplo sería el uso de las llamadas o de las agendas dentro de otros programas, esto se hace mediante invocaciones a la actividad del dispositivo para llamar, enviar un mensaje, etc. Este tipo de comunicaciones únicamente se pueden emplear en la aplicación cuando el usuario así lo determina durante la instalación.

Todo el concepto de actividades y su interejecución no podría existir sin el framework de soporte que da la propia máquina virtual. Cada vez que una actividad es ejecutada, esta entra como una instancia independiente y separada. Dalvik crea una instancia que sirve de caja de arena a cada actividad. De esta manera, toda actividad incluyendo las llamadas telefónicas, mensajes, agendas, etc dispone de su propio entorno sin interferir con el resto de actividades. Así es como Android cumple el requisito de competencia igualitaria entre las aplicaciones por los recursos, a la vez que este encapsulamiento garantiza una mínima tolerancia a fallos, evitando su propagación fuera de la actividad que lo provocó. Si una actividad incurre en un error grave, este no afecta a las otras actividades. En un dispositivo, cuyo propósito principal es poder utilizarlo para recibir y enviar llamadas, no es agradable tener que reiniciar tu teléfono para poder hacer una llamada porque se ha quedado congelado.

Lo explicado hasta este punto, abarca las ideas básicas de funcionamiento que tiene Android desde sus primeras versiones. Sin embargo, debido al desarrollo y a las nuevas tendencias se han tenido que añadir otras ideas. El número de dispositivos y la variedad de configuraciones ha crecido de forma significativa durante los últimos tres años. Actualmente existe soporte para cinco tipos de pantalla diferentes, así como múltiples diferencias de densidad de píxeles según dispositivos y resoluciones diferentes. El problema de la compatibilidad de la aplicación ya no es debido a su funcionamiento, ahora se debe a la gran variabilidad de configuraciones de pantalla, componentes hardware, etc. No todas las aplicaciones son capaces para estar preparadas para todas las variabilidades existentes.

La respuesta de Android, es la inclusión de requisitos mínimos en las aplicaciones. Una aplicación puede pedir unos requisitos mínimos al dispositivo en el que es instalado. Para comenzar, un dispositivo que no sea compatible será incapaz de encontrar en el repositorio de aplicaciones una que tenga un requisito mínimo que no cumpla, si aun así el usuario intenta instalarla manualmente, el instalador revisa las características mínimas e impide al usuario su instalación.

2.2.2. Desarrollo de las aplicaciones

Las aplicaciones de Android no se presentan en un ejecutable o binario. Cuando se genera una aplicación de Android, esta se empaqueta siguiendo la arquitectura de los APK.

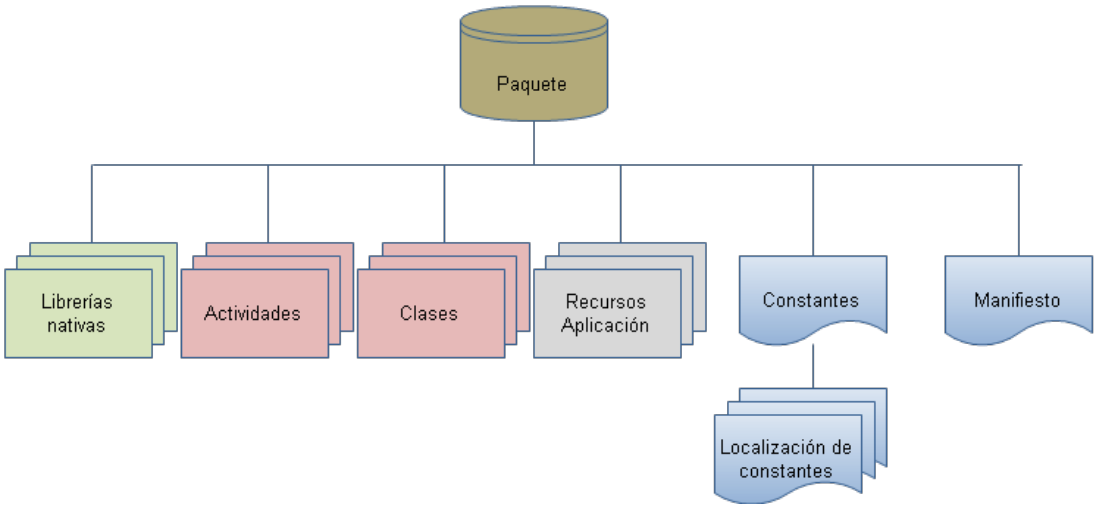

Figura 2.4: Diagrama de los contenidos de un paquete APK.

Los paquetes APK son archivos comprimidos que contienen toda la estructura de una aplicación. En el directorio raíz, las aplicaciones contienen un archivo de manifiesto en formato XML.

Un archivo de manifiesto tiene varias partes:

- Definición de la aplicación
- Definición de las actividades
- Restricciones

La definición de la aplicación representa el nombre, el nombre del paquete Java/-Dalvik que contiene el código y los números de versión de la aplicación. La definición de las actividades es una lista de las diferentes clases que son derivadas de la clase Ac-

tivity y que serán ejecutadas en la aplicación. No se puede realizar una "intent"[1] sobre una actividad de nuestro paquete si esta no se encuentra definida en el manifiesto.

Finalmente las restricciones que se pueden aplicar en un paquete son de diverso tipo: restricciones por tamaños de pantalla, densidades, compatibilidad con librerías, la presencia de determinados componentes hardware, formatos de compresión de texturas, etc.

El directorio principal tiene las diversas clases compiladas en formato Java Dalvik (.dex) y el directorio de recursos. Los recursos de una aplicación se encuentran en esta sección. Y pueden estar comprimidos, o no, dependiendo del tipo de archivo. Esta parte del empaquetado se ha ido cambiando a lo largo de las versiones. Por norma general, se busca que los paquetes tengan el menor tamaño posible, por ello, aunque se permita incluir archivos en el mismo paquete, hoy en día la plataforma soporta el uso de paquetes secundarios de tamaño indeterminado para contener los recursos y que se descargan posteriormente de forma automática.

De esta manera el desarrollador despliega la aplicación en dos paquetes, uno con la parte mínima para la ejecución del programa y otro con todos los recursos gráficos. De cara al usuario final este tipo de distribución es muy importante ya que evita la descarga de la mayor parte de datos cada vez que existe una actualización en la aplicación que no modifique los assets.

La plataforma Android emplea de forma repetida los ficheros de tipo XML. Se emplean tanto para ficheros de configuración como para guardar definiciones, valores por defecto y traducciones de la aplicación. Esta es una práctica aceptada y consolidada dentro de la API que evita mezclar el código de la aplicación con el texto ofreciendo al desarrollador una forma automática de gestionar la traducción de aplicaciones.

Finalmente, si una aplicación es nativa o requiere de alguna librería adicional, el paquete incluye las librerías nativas con las que se debe enlazar.

El mecanismo de paquete es usado actualmente para instalar únicamente las partes que el dispositivo destino pueda emplear. Como se comenta en el siguiente punto, al desarrollar en nativo, se compila para diferentes arquitecturas. Sin embargo cuando se instala un paquete, únicamente se copian las librerías que corresponden a la arquitectura del dispositivo. Esta forma de proceder ha sido extendida en la actualidad en el mercado de Google. Se ha incluido soporte para que una única aplicación tenga diferentes paquetes para tratar casos específicos como el uso, o no, de compresión de

[1] Intent es el término empleado en Android para llamar y ejecutar una aplicación de tu paquete o del sistema operativo desde cualquier aplicación

texturas.

Una aplicación, debe estar formada por la estructura expuesta con los diversos archivos de configuración además del código propio de la aplicación, ya que Android requiere de dichos archivos para configurar apropiadamente el entorno de ejecución de la aplicación, por lo tanto esto ha de cuidarse especialmente en las aplicaciones que se desarrollarán en este trabajo.

Centrándonos más específicamente en el código ejecutable, las aplicaciones de Android no son monolíticas, se encuentran fraccionadas en actividades. Citando la definición de Google.

"***Una aplicación Android se compone de una serie de actividades vagamente relacionadas***"

Una actividad es un proceso del programa que encapsula, a la vez, la interfaz gráfica del proceso y su funcionalidad, y está intrínsecamente relacionada con la interfaz del usuario. Son procesos que deben ser visibles y ofrecer respuesta (o permitir la comunicación) a los eventos generados por el usuario. A diferencia de un proceso común de los sistemas de sobremesa como Unix o Windows, la visibilidad es uno de los factores más importantes para decidir el estado en el que se debe encontrar una actividad.

Tenemos que tener en cuenta que las aplicaciones en Android están pensadas para coexistir con el uso normal de un teléfono móvil. Un usuario normal espera poder emplear su agenda diaria en el teléfono y si recibe una llamada, contestar a esta sin que la aplicación le interrumpa o sea visible. Sin embargo, atender una llamada no significa que el usuario quiera cerrar lo que estaba haciendo. Normalmente, el usuario no se preocupará de salvar sus cambios antes de coger la llamada; por lo tanto el comportamiento que espera y desea el usuario es que la aplicación permanezca a la espera para, posteriormente, reanudar la ejecución desde el punto en el que la dejó.

Hasta ahora se ha hablado únicamente de las actividades, Dado que existe una noción de estado en la propia aplicación y únicamente se emplea el concepto de que un programa esté ejecutando la actividad X en un punto determinado, esto implica que el ciclo de vida no lo es a nivel de aplicación sino de la "actividad". Este ciclo se puede ver en la ilustración: 2.2.2

Una actividad creada por el usuario se crea extendiendo la clase Activity de Android, la cual, tiene una serie de métodos que se llaman automáticamente cuando el sistema reconoce un cambio de estado para la actividad. Dichos métodos pueden ser reimplementados para cada vez que una actividad realice los ajustes necesarios para su comportamiento.

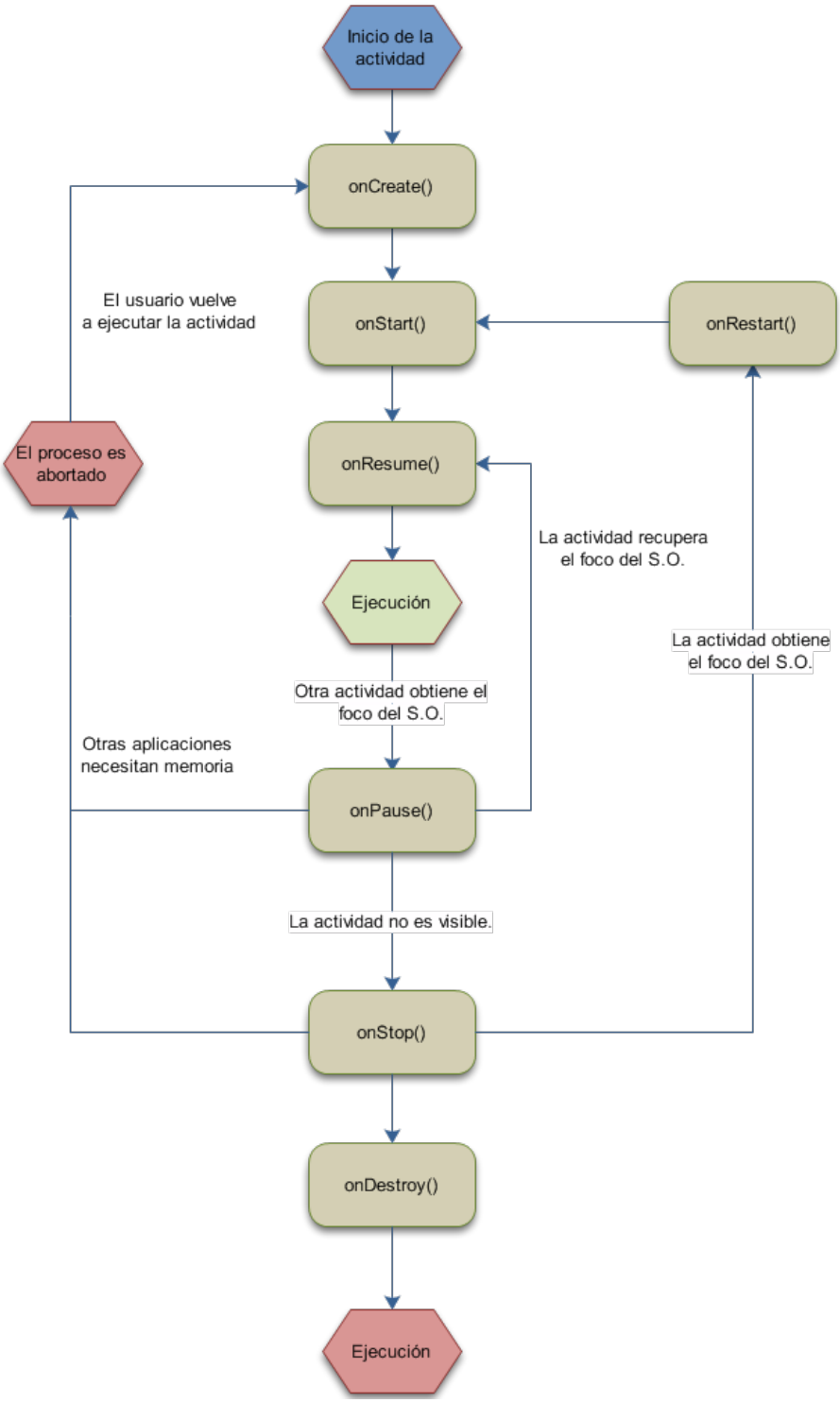

Figura 2.5: Diagrama del ciclo de vida de una actividad en Android.

La vida de una actividad transcurre entre su creación y su destrucción. Estas se producen con las llamadas a "onCreate" y "onDestroy". Durante la creación, la actividad creará todos los elementos que necesite para su estado interno de ejecución, servicios, hilos, etc. En su destrucción, liberará todos los recursos que haya ocupado.

El tiempo de vida que el usuario percibe es aquel que comienza en "onStart" hasta que llega a "onStop". En este período, la actividad es visible por el usuario, sin embargo no es necesario que tenga el foco visual, ya que esta puede estar paralizada en un segundo plano.

Finalmente el tiempo de vida que la actividad realmente es controlada por el usuario abarca desde "onResume" hasta "onPause". La pausa y la reanudación son algunos de los momentos más habituales de una programa. Es común que una actividad se quede a la espera de una respuesta de otra actividad o se suspenda el dispositivo.

Hasta ahora se ha hablado únicamente de las actividades. Como se ha dicho, una actividad necesita tener una representación visual y responder ante los eventos. Esto significa que en ningún momento el hijo principal de ejecución de la actividad puede bloquear la entrada de eventos. Cuando la actividad no es capaz de responder al sistema durante un tiempo, la cantidad varía dependiendo de la implementación, Dalvik aborta la actividad por un error ANR (Activity Not Responding).

Existen determinadas aplicaciones que, debido a sus requisitos, cálculos complejos, carga de archivos, transmisión de datos, tienden a ocasionar una espera demasiado larga provocando el error ANR. Por ello en Android existe el soporte de hilos (Threads) desde el framework Android y adicionalmente, incluye los servicios, que a diferencia de los Threads, funcionan desde el mismo hilo de ejecución que la actividad que los ha invocado, esto implica que no pueden resolver el error ANR, ya que su propósito es crear tareas en el sistema operativo. Estas se ejecutarán en segundo plano sin que el usuario tenga constancia de ellas. Una de sus grandes utilidades es que diferentes actividades pueden hacer uso de dichos servicios si se encuentran presentes.

Es muy importante tener en cuenta el ciclo de las actividades así como el error ANR ya que son conceptos básicos de la plataforma. Si no se cumplen los requisitos de cambios de estado, guardado de estado actual, etc las aplicaciones que se buscan desarrollar en este trabajo serán incapaces de convivir correctamente con el resto de aplicaciones en la plataforma.

Después de la inclusión de la Native Activity en la versión 2.3 de Android, las aplicaciones de Android pueden generarse de dos formas diferentes:

- Aplicaciones Java dentro de la máquina virtual Dalvik
- Aplicaciones con componentes nativos.

Una aplicación de Android, actualmente, está controlada en mayor o menor medida por la máquina Dalvik. No se puede crear una aplicación completamente nativa dentro del ecosistema Android.

Toda aplicación, incluso aquellas que utilizan las "NativeActivity", comienzan su ejecución en la parte controlada por Dalvik con las mismas secuencias de eventos. Una aplicación que emplea métodos nativos, utiliza además JNI para establecer un puente para realizar llamadas sobre código nativo no manejado por la máquina virtual. El framework de Android, nos ofrece una clase especial llamada "NativeActivity" realiza la carga de la biblioteca nativa y nos permite crear nuestra actividad en C/C++, sin embargo la actividad no es totalmente nativa, ya que en realidad todos los eventos que llegan a nuestra aplicación han pasado por el puente JNI.

La sobrecarga que introduce JNI puede parecer un factor negativo; sin embargo siempre y cuando no se realice de forma intensiva el rendimiento no caerá significativamente. El código nativo nos da dos importantes ventajas:

- Manejo de la memoria
- Acceso directo al driver nativo OpenGL ES.

Al trabajar con código nativo, no estamos limitados ni controlados por el "garbage collector" de Dalvik. Esto supone una mayor responsabilidad en el lado del programador, ya que pierde la posibilidad de que el sistema operativo controle los fallos de pérdidas de memoria. Por otro lado esto nos permite utilizar la mayor parte de memoria del dispositivo, siempre dentro de unos límites. Si la máquina virtual detecta que se queda sin memoria para ejecutar los procesos del sistema operativo, matará las aplicaciones por orden de consumo de memoria, y aunque la memoria usada nativamente no esté manejada por recolector de basura de Dalvik, el sistema operativo si que conoce la cantidad de memoria empleada por el proceso de la aplicación.

Trabajar desde código nativo, es una situación ideal para hacer aplicaciones gráficas. Desde el código nativo podemos realizar llamadas directas a la API OpenGL ES del dispositivo sin la sobrecarga que nos da el binding Java de OpenGL y las llamadas JNI que realiza para transmitir nuestras órdenes. Además se evita que el proceso de recolección de basura ralentice de forma significativa nuestro proceso de renderizado.

Las figuras 2.6 y 2.7 representan una modelo básico de estructura de aplicación para Android, concretamente dos modelos de aplicación que emplean la librería OpenSceneGraph (OSG) en Android.

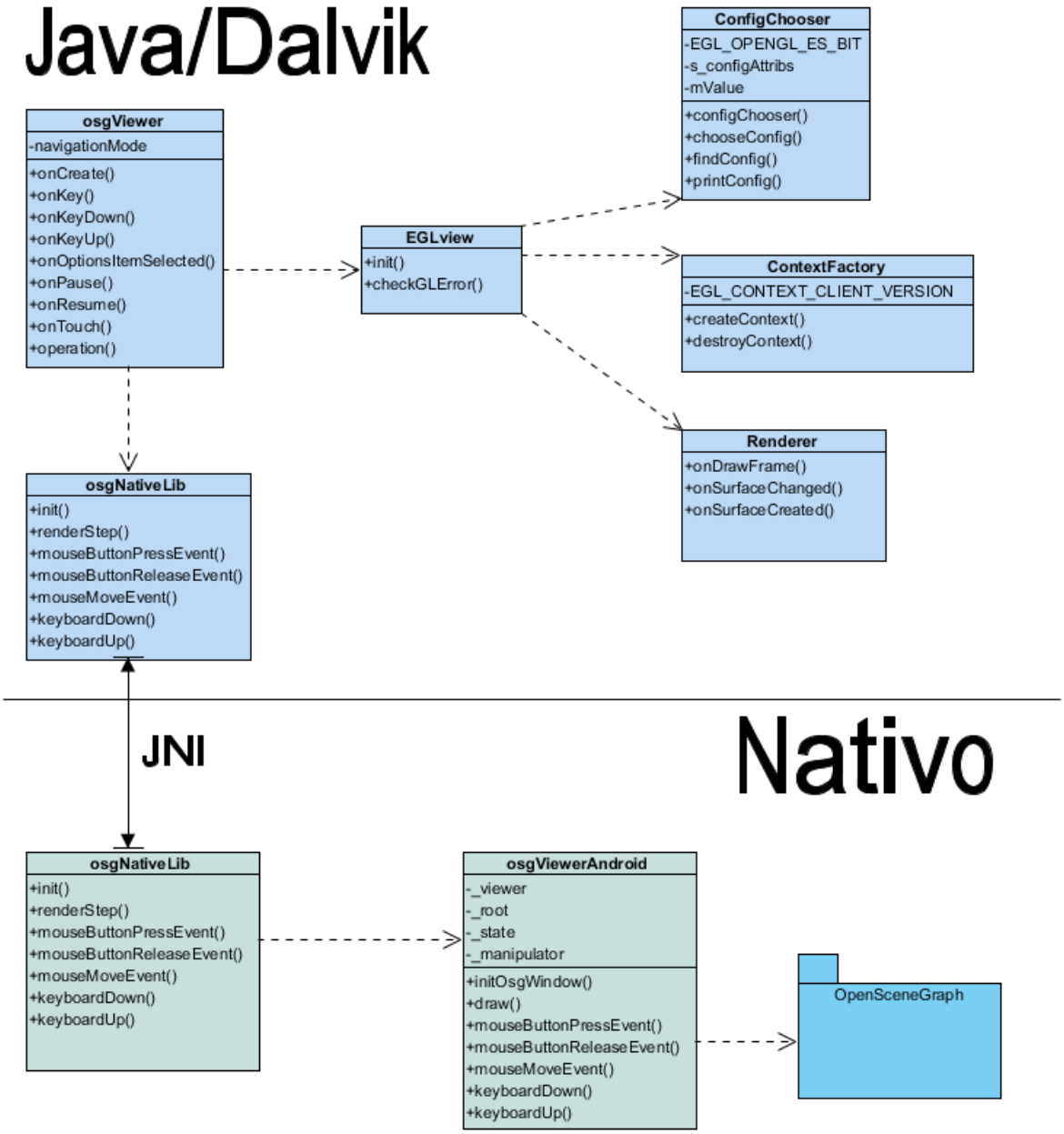

Figura 2.6: Diagrama de clases de la estructura de una aplicación OSG en Android.

Como se puede ver en la figura: 2.6 la actividad principal se genera como código Java. Esta actividad, se comunica con una clase derivada de GLSurfaceView (EGLview) que se encarga de realizar la configuración del contexto gráfico y del registro de la función de renderizado. Para realizar las llamadas a la parte nativa se utiliza la clase osgNativeLib, esta clase contiene los métodos que la capa nativa de la aplicación expone a la capa Java. Una vez en el nivel nativo, el código puede realizar llamadas a otras librerías, bien propias o del sistema operativo. También es posible realizar llamadas en el sentido inverso, es decir, realizar llamadas a la capa Java desde la parte nativa de la aplicación.

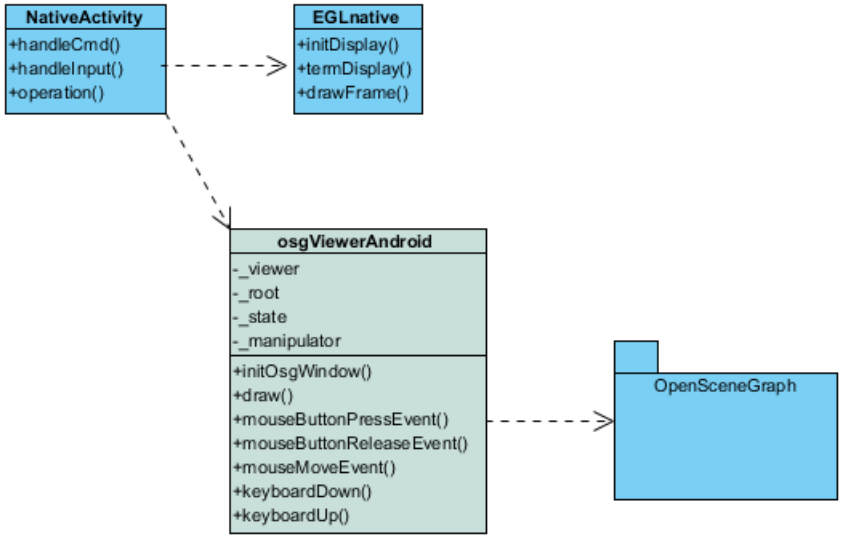

Figura 2.7: Diagrama de clases de la estructura de una aplicación OSG en Android usando NativeActivity.

En contraposición, la versión puramente nativa representada en: 2.7 tiene una clase general que registra los métodos para comunicarse con la "NativeActivity". Estos a su vez, pueden acceder a la clase osgViewerAndroid o realizar cualquier llamada que estimen a librerías del sistema, de la aplicación e incluso al framework Android a través de la capa Java.

2.3. OpenSceneGraph

OSG [OSG12] es una librería de alto rendimiento para trabajar con gráficos tridimensionales, está publicada como código libre y ha sido integrada en en aplicaciones libres y comerciales de todo tipo. Es capaz de manejar entornos tridimensionales complejos y representar gráficos de última generación sin ningún tipo de limitación. A diferencia de otras alternativas, OSG únicamente ofrece las funciones de un grafo de escena donde se pueden incluir nodos propios para extender sus funcionalidades básicas de representación. Debido a la no especialización de la librería, puede ser empleada para la creación de todo tipo de aplicaciones científicas o comerciales que necesiten representar información gráfica. Ha sido utilizada para crear aplicaciones de visualización, realidad aumentada, simuladores de vuelo o juegos.

El núcleo de OSG, se basa en el empleo de una metodología de grafos de escena. Estos son estructuras de tipo grafo acíclico dirigido que forman una representación jerárquica de la escena. Esta ordenación permite realizar optimizaciones espaciales y de representación para mejorar el rendimiento de la visualización en escenas complejas. Esto se realiza mediante inspecciones del grafo que permite visualizar, o no, ramas enteras dependiendo de nuestro criterio de visibilidad y, a partir de la selección generar un orden de visualización óptimo para la renderización de los elementos. Esto permite al programador abstraerse del uso de los comandos de bajo nivel de las API gráficas y concentrarse a nivel de objetos de escena sin preocuparse del código necesario para generar la visualización

Una de las características de OSG es su arquitectura modular que permite añadir elementos y generar nuevos módulos y plugins para ser empleados en el grafo de escena permitiendo al programador extender la librería según sus necesidades. Si estudiamos su estructura, OSG está formada por una serie de pequeñas librerías y plugins. Ambos extienden la funcionalidad, bien añadiendo efectos y patrones gráficos que se pueden incluir directamente en el grafo de escena, o bien añadiendo la posibilidad de trabajar con tipos de archivos. Generalmente estos plugins requieren de librerías externas independientes de OSG que se enlazan dinámicamente con el programa en

ejecución.

OSG además se ha diseñado para cargar las librerías y plugins bajo demanda. Esto supone que un programa que emplee OSG, únicamente cargará las librerías básicas y, dependiendo de las necesidades, el resto de librerías y plugins serán enlazadas dinámicamente cuando el usuario lo requiera. Esta característica permite ahorrar memoria durante la ejecución de un programa al no tener que cargar los módulos que no se utilicen.

Esta metodología funciona bien en plataformas de sobremesa. Por el contrario en dispositivos embebidos o smartphones resulta, muchas veces, imposible de usar, por ello, OSG incluye la posibilidad de una compilación estática de todas las librerías y plugins. Esta compilación requiere que el usuario registre una serie de macros realizan una serie de definiciones internas en la base de datos de OSG que permitan emplear los plugins y librerías sin necesidad de enlazarlos dinámicamente.

La librería OSG únicamente tiene una dependencia directa, la librería OpenThreads(OT). Para simplificar la cantidad de código dependiente de sistemas operativos, OSG encapsula todas las operaciones de hilos en la librería OT. Esta librería ha sido publicada como un proyecto independiente de OSG, aunque se encuentra incorporada en la estructura de la distribución de la librería. De forma opcional, OSG depende en una larga cantidad de librerías para el uso de diferentes formatos de archivos.

Por otro lado, OSG tiene implementadas diversas estrategias para la reducción de complejidad de una escena tridimensional, carga de elementos bajo demanda y permite realizar inspecciones completas de los grafos de escenas. Con estas posibilidades, se pueden crear una técnicas de optimización más complejas como el uso de quadtrees, octrees u otros modos de ordenación del espacio geométrico tridimensional.

Desde el año 2010, y en base al trabajo realizado en el proyecto final de carrera: Representación interactiva de escenas tridimensionales con OpenSceneGraph en Android [Cig11] la librería OSG funciona en el sistema operativo Android.

2.4. VirtualPlanetBuilder

VPB es una librería basada en OSG. El propósito de la librería es la creación de bases de datos de geometría tridimensional de terrenos. A diferencia de otras bases de datos geográficos, las generadas por el programa están formadas por los diferentes nodos que conforman la representación del grafo de escena. Si desgranamos los archivos generados, vemos perfectamente la ordenación jerárquica que se carga en la

escena de nodos y sus tipos. Al estar basada en OSG, permite emplear todos los tipos de archivo de modelos y texturas que soporta OSG para generar nuestras bases de datos geográficas.

VPB permite generar bases de terrenos planos o esféricos. Opcionalmente, si se poseen datos geométricos, se pueden emplear para que se forme el terreno con mallas geométricas que representen las alturas. Entre las opciones más destacadas, hay que señalar que también permite emplear la librería Nvidia Texture Tools (NvTT) [Nvi] para emplear texturas con compresión, el uso de texturas comprimidas cuyo uso está muy extendido en la actualidad. Para ello se usan una serie de formatos de compresión fija cuya descompresión se ejecuta a muy bajo coste computacional en las tarjetas. Algunos de los formatos de compresión que admite son: DXT $1/3/5$ [Cas07].

2.5. CMake

CMake es una aplicación multiplataforma diseñada para ofrecer un sistema de compilado independiente de la plataforma. Mantener una aplicación multiplataforma es una labor compleja. Ya que suelen existir diferencias y parches que dependen de la plataforma objetivo, además, es necesario tener los scripts que sirvan para compilar la librería en cada uno de los sistemas para los que ha sido desarrollada. El objetivo de CMake es evitar el mantenimiento de un gran número de scripts por plataforma unificándolos en un único tipo de fichero. El programador únicamente tiene que escribir una serie de ficheros con el lenguaje de scripting de CMake, en ellos define los paquetes, archivos a compilar, opciones, etc. Finalmente el usuario que desea compilar el proyecto ejecuta CMake, a partir de dichos scripts, CMake generará los ficheros apropiados para ejecutar la compilación de acuerdo a la plataforma objetivo que emplea el usuario.

Al ser un lenguaje de scripting, CMake puede ser empleado para extenderse a si mismo. Es posible emplearlo para generar scripts de compilación diferentes a los que ya tiene programados internamente.

2.6. Compilación cruzada

La compilación cruzada, es un término que emplearemos varias veces en el trabajo y que conviene clarificar. La compilación cruzada (Cross compiling en inglés), sirve para denominar aquellas compilaciones que se generan en una arquitectura diferente

de la que se va a emplear para ejecutar el código compilado. Es la forma normal de compilación para dispositivos embebidos, consolas, o en general todo sistema operativo donde no se tiene la posibilidad de emplear un compilador. En la actualidad existen varios ejemplos de compiladores libre y comerciales que permiten esta técnica: GCC, GUB o Intel C++ Compiler entre otros.

2.7. Renderizado de geometría tridimensional en dispositivos embebidos

La representación gráfica tridimensional en dispositivos embebidos es un problema que ha tenido un largo recorrido. Los primeros dispositivos móviles carecían de un criterio común o estándar para tratar la renderización de gráficos. La aparición de diferentes API gráficas propietarias sin compatibilidad complicaban mucho el desarrollo de este tipo de aplicaciones. Conforme las necesidades gráficas aumentaron se fue generando la necesidad de un estándar gráfico mínimo. Siguiendo a su homólogo de sobremesa, la Kronos ARB creó un estándar OpenGL para dispositivos embebidos [Khr04], aunque en la actualidad sea un estándar para la mayor parte de dispositivos, a lo largo de la evolución de estos dispositivos han aparecido otros estándares que han entrado en competencia. Por citar algunos, habría que hablar de PocketGL [Ler04] desarrollado por Pierre Leroy o la reciente implementación de DirectX en el sistema móvil de Windows.

En los últimos años, los avances en la potencia de cálculo y consumo de los chipsets gráficos ha permitido un avance de calidad en los programas gráficos móviles. En la actualidad, la API GLES 1.X ha sido sustituida, en la mayoría de dispositivos, por su sucesor: GLES 2.0. Se ha eliminado el uso de la tubería fija de procesado pasando al uso de una tubería programable. Incorporando parte de las ideas presentes en las especificaciones 2.0 y 3.0 de la API de sobremesa OpenGL. Para una información más detallada de la nueva API, se puede leer el artículo de Ken Catterall [Cat10] donde se abordan las nuevas posibilidades que ofrece esta API, la eficiencia que presenta en los dispositivos y proporciona una serie de programas shader con la implementación de diferentes técnicas. También recomendamos el artículo de Gustavsson et al. [GBO09] que presenta una comparación entre las dos APIs gráficas así como las partes importantes a tener en cuenta para optimizar el código con ambas versiones. Recientemente se ha introducido la versión 3.0 del estándar pero esta no ha sido tratada en el curso del trabajo.

La redenderización interactiva en dispositivos móviles está restringida por las limitaciones inherentes de estos dispositivos. Cálculo computacional, capacidad de memoria y latencia en la memoria física. Estos temas se han estudiado previamente con diferentes acercamientos: (1) Renderizado basado en imagen [CG02, BPT05, SP09, SZL02]. (2) Renderizado con puntos [GD98, HL07, DD04] (3) Renderización geométrica con un sistema de simplificación externa [LGCV05, LGEC06, HMK02].

El renderizado basado en imágenes emplea una simplificación de elementos geométricos utilizando elementos bidimensionales. Este es el caso del trabajo de Chang et al. [CG02] o el de Bouatouch et al. [BPT05]. Ambos estudios sugieren el uso de una arquitectura cliente-servidor que renderiza la escena en el servidor y manda, como resultado partes de la escena, o la escena completa, como una imagen al dispositivo. Para realizar la simplificación de la escena, Sanna et al. [SZL02] emplea una estructura jerárquica de cara a optimizar el acceso a diferentes servidores y dividir las zonas que se han de simplificar. En el trabajo de Sterk et al. [SP09] aplican esta técnica para el renderizado de globos terráqueos.

El método de renderizado basado en puntos, consiste en recubrir la forma de un objeto con una nube de puntos. Se descarta la malla original y únicamente se representa una parte de ese conjunto de cara al usuario. Este sistema de renderizado se encuentra ámpliamente analizado en el trabajo de Grossman et al. [GD98]. Basándonos en Zhiying et al. [HL07], esta técnica se puede emplear para generar representaciones multiresolution que entren dentro de los límites de memoria de un dispositivo. También encontramos en el trabajo de Duguet et al. [DD04] el uso de un modelo jerárquico para optimizar la generación de la vista.

La renderización geométrica con un sistema de simplificación externa ha sido empleada en los trabajos de Lluch et al. [LGCV05, LGEC06]. Esta técnica está basada en un modelo cliente-servidor: El cliente, dispositivo móvil, realiza una petición de representación de la escena visual. El servidor por su lado obtiene los datos de la vista que tiene el dispositivo y genera una vista simplificada de la escena. Esto se consigue reduciendo la calidad de los modelos, texturas, eliminando elementos no visibles de la escena, etc. Una vez la escena se ha simplificado, el servidor devuelve la información al dispositivo que, finalmente, renderiza la escena. Estos trabajos se basan en el uso de un grafo de escena (OSG) para realizar las optimizaciones necesarias para que la escena entre en los límites de memoria del dispositivo. Esta misma técnica se emplea en el trabajo de Hekmatzada et al. [HMK02] pero centrándose en la representación de modelos no fotorrealistas.

Como he comentado existen diferentes ejemplos que destacan el uso de la meto-

dología basada en representaciones jerárquicas de la información. Este es el caso de los grafos de escena. Se emplean para simplificar y optimizar la representación de una escena visual. Durante este trabajo vamos a emplear OSG, una librería open-source que puede ser utilizada en aplicaciones comerciales y no comerciales. En la actualidad es uno de los principales grafos de escena basados en OpenGL y se emplea de forma significa en el uso de aplicaciones científicas.

Algunas de las librerías que se emplean actualmente en Android son: jMonkey, jPCT-AE o Simple DirectMedia Layer(SDL).

2.7.1. Unity

Unity es un kit de desarrollo de aplicaciones gráficas especializado en videojuegos, por esta razón su empleo se suele limitar a juegos, aplicaciones comerciales que rendericen escenas o hagan simulaciones gráficas. En la actualidad, Unity se puede utilizar para desarrollar aplicaciones multiplataforma en PC, iOS, Android y algunas consolas. La gran ventaja de este kit de desarrollo es que está orientado para que se pueda emplear sin conocimientos avanzados de programación, pero si se buscan comportamientos complejos, una persona con conocimientos de programación puede crear sus propios elementos lógicos y scripts dentro del framework que ofrece Unity al desarrollador.

2.7.2. jMonkey

Es una librería especializada para la creación de videojuegos cuyo lenguaje primario es Java. Actualmente se usa en algunos proyectos libres. La librería implementa un grafo de escena para la renderización de elementos, los cuales son extensibles debido a su diseño modular.

2.7.3. jPCT-AE

Es una librería especializada para la creación de videojuegos cuyo lenguaje primario es Java, permite la implementación de programas con simulación de físicas y capacidades en red. Emplea optimizaciones jerárquicas basadas en en la geometría de la escena.

2.7.4. Simple DirectMedia Layer

SDL es una librería libre con un largo desarrollo en los computadores de sobremesa. Su lenguaje primario es C aunque existen una serie de enlaces para usarse con otros. Está especializada en ofrecer acceso a bajo nivel del hardware de audio, vídeo y controles.

Ninguna de las alternativas actuales ofrece la libertad de uso y especialización propia de la librería OSG, además, el renderizado, en la mayor parte de ellas, se realiza desde el nivel de la máquina virtual mediante Java sin acceso nativo. En este trabajo, se ha decidido abordar la compatibilización de la librería OSG para poder tener una librería que no esté especializada para hacer representaciones gráficas de un único tipo de programa y que realice el renderizado desde un nivel nativo.

De esta manera, este trabajo pretende prescindir de cualquier tipo de arquitectura de refuerzo externa o simplificación no geométrica. Se entiende que la evolución actual de los dispositivos tras el recorrido presentado en esta sección, permitirá emplear un grafo de escena desde el propio dispositivo y que permita optimizar las escenas para renderizar, con él, terrenos tridimensionales.

2.8. Renderización de terrenos

La renderización de terrenos tridimensionales, es la unión de varias capas de información expresada en una geometría. Para generar la geometría de un terreno se necesita la información visual del terreno, la ortofoto, y, al menos, una capa de información de puntos geodésicos de altura. A partir de estos datos se puede generar una representación visual en tres dimensiones. Este proceso puede ser realizado de tres formas distintas: Durante la creación de la base de datos, durante la carga de los datos y en el dibujado.

La conversión geométrica de los datos bidimensionales a un espacio tridimensional es un proceso idéntico en los tres casos, las únicas diferencias entre ellos radican en el número de veces que se realizan, cuando y qué herramientas están al alcance para realizar el proceso. Sin entrar de forma detallada, se genera una gratícula o malla cuyas alturas son alteradas para ajustarse a los diferentes parámetros de altura. La ortofoto se emplea para dar el color a los polígonos de dicha malla. Es el mismo proceso que se realiza en las técnicas de mapas de alturas [AMHH08].

Cuando se genera la geometría en una base de datos, el resultado supone que la

base de datos incremente su tamaño. Para evitar esto, se suelen emplear algoritmos de compresión de datos sobre la información. Esta solución es la que está implementada sobre el programa Google Earth. Es una solución que obtiene los mejores costes computacionales en tiempo de dibujado ya que no se ha de realizar ningún proceso, con la desventaja de aumentar el peso de los datos a transmitir e inflexibilizar la representación. Debido a su bajo coste, esta solución se emplea durante este proyecto para realizar dos de los programas de prueba de representación de terrenos.

Si la geometría se genera durante la carga de datos, los datos que se transmiten al programa no aumentan de tamaño al ser los mismos datos originales sin incluir geometría. La desventaja de este método es que requiere de un procesado con un coste temporal durante la carga, esto generará una latencia que se ha de suplir mediante el procesado en un hilo no bloqueante y cachés para mejorar la experiencia. La mayor ventaja de este método es que la imagen representable no es única y puede ser cambiada en tiempo de ejecución modificando los datos de entrada. Esta es la solución que está presente en programas como GvSig.

Finalmente, realizar el proceso durante el tiempo de dibujado, aporta la posibilidad de evitar la generación de geometría que no es visible además de evitar el tiempo de preproceso. En este caso el proceso de conversión se realiza en la propia tarjeta gráfica en cada pase de dibujado. La ventaja de este método es que se realiza la conversión de las zonas visibles y dicha conversión puede ser realizada con el nivel de detalle que se ajuste mejor al rendimiento y al punto de visión actual. En este artículo, se muestra el uso de esta forma de representar terreno para representar terrenos con mallas de detalle variable y su impacto en el rendimiento.

Para la creación del programa de pruebas de terrenos tridimensionales creados en tiempo de dibujado, este trabajo se basa en la técnica de desplazamiento de vértices. El artículo [SkU06] sirve de resumen del estado actual del arte de dicha técnica. Otro trabajo importante a mencionar es [Kry05] con su implementación de la técnica para el renderizado de agua. A diferencia del trabajo [Don05] solo realizaremos la técnica a nivel de vértices en vez de realizarla por píxel.

Debido a que la técnica realiza el cálculo de la geometría en tiempo de renderizado, es necesario implementar el cálculo de normales, o su lectura si ya están pregeneradas. Para el cálculo de las normales en tarjeta, se ha empleado el trabajo [Mik10], debido a las limitaciones actuales por el coste de los cálculos en coma flotante, empleamos una simplificación de la técnica de cálculo de Bump Mapping para el cálculo de las normales.

En la plataforma Android, ya se han desarrollado una serie de trabajos sobre la representación de terreno. En [SP09] se compara la eficiencia de emplear un renderizador local en contra de uno remoto, mientras que en [HW09] se plantea un posible diseño para la representación de modelos planetarios.

2.9. Protocolo Web Map Service

Web Map Service [OGC00] (WMS) es un protocolo definido por el Open Geospatial Consortium (OGC). OGC es un consorcio internacional de compañías, agencias gubernamentales y universidades. Su objetivo principal es la de servir como un forum de colaboración entre desarrolladores para crear estándares internacionales para garantizar la interoperabilidad de datos y servicios geoespaciales.

Los estándares desarrollados por OGC buscan resolver problemas relacionados con la creación, comunicación y uso de datos espaciales, bien sean datos cartográficos, modelos 3D urbanos u otro tipo de datos geolocalizados. En la actualidad sus estándares se usan ampliamente en sectores como: aeronáutica, busines intelligence, defensa, emergencias y manejo de catástrofes, estudios medioambientales o planificación urbanística.

El protocolo WMS sirve para realizar peticiones de mapas basados en información geográfica referenciados espacialmente. Según la definición de WMS, un "mapa" es la representación, en un archivo de imagen digital, de una información geográfica para representarse en una pantalla. No se considera que el mapa en sí sean los propios datos. Los mapas generados por un servicio WMS son, por norma general, archivos gráficos renderizados de formatos como PNG, GIF o JPEG. También es posible que se generen mapas basados en elementos gráficos vectoriales en formatos como Scalable Vector Graphics (SVG) o Web Computer Graphics Metafile (WebCGM).

El estándar de WMS define tres operaciones:

- GetCapabilities
- GetMap
- GetFeatureInfo

La primera devuelve un archivo de metadatos (xml) con la información del servicio. La segunda sirve para realizar peticiones sobre los mapas ofrecidos por el servidor.

La tercera operación, que es opcional; y por lo tanto no se encuentra implementada en todos los servicios WMS, devuelve información sobre algunas características definidas que se muestran en el mapa.

Las operaciones del servicio WMS pueden ser invocadas usando un navegador web estándar utilizando Uniform Resource Locators (URLs). La composición de los parametros cambiará dependiendo de la operación solicitada. Por ejemplo, la operación de obtener mapas requiere de información sobre la capa que se pide la información, las coordenadas solicitadas o el formato de datos que deseamos recibir.

La especificación WMS tiene la ventaja de ser totalmente transparente de cara al usuario final del servicio. El usuario del servicio, únicamente conoce las diferentes capas que puede solicitar, así como las limitaciones en formatos de imagen soportadas por la capa o sistemas de referencia que se pueden emplear para realizar peticiones.

Un servicio WMS clasifica los diferentes tipos de datos accesibles a través de él en forma de "Layers" (capas). Ademaś es capaz de ofrecer al usuario una serie de "Styles" (estilos) de cara a la representación para el usuario. La información de las capas se estructura de forma jerárquica. Una capa define su extensión, los sistemas de coordenadas que soporta para realizar peticiones, los formatos admitidos para la generación de datos en una petición y una serie de metadatos. Una capa ademaś puede opcionalmente tener un nombre o identificador.

Una capa WMS puede incorporar otras capas WMS como hijas. Conceptualmente, se entiende que cada una de ellas es un subconjunto de las capas antecesoras. Una capa hereda la información geográfica paterna y sobrescribe (especializa) la información para sus necesidades. Esta sobreescritura incluye la inclusión de nuevos formatos de imagen, nuevos sistemas de referencia para las peticiones o, más comúnmente, la sobrescritura de los límites geográficos para realizar peticiones.

Así, una capa contenida dentro de otra, puede dar una información más concreta que sus antecesoras. También puede ser una división o fragmento de la superior.

Las únicas peticiones de mapas que un usuario puede realizar contra el servidor WMS es contra capas que tengan nombre. Así pues, no todas las capas tienen por qué ser accesibles para el usuario. Pueden ser utilizables como una representación jerárquica interna del servidor.

Esto permite que un servicio WMS, implemente de forma distribuida sus servicios distribuyendo la carga de solicitudes del usuario entre diferentes servidores. También es posible que un servicio WMS incorpore la posibilidad de realizar llamadas sobre otros servicios o servidores, de forma interna sin afectar al usuario final.

2.10. GDAL

GDAL [htt] (Geospatial Data Abstraction Library) es una librería para leer, escribir y procesar rasters de formatos geoespaciales. Y está liberada con la licencia MIT. Desarrollada por la Open Source Geospatial Foundation (OSGeo) con el objetivo de promover y dar soporte a diferentes alternativas libres de distribución de datos georeferenciados. Como librería, GDAL emplea modelo único de datos abstracto para contener y tratar diferentes tipos de capas y entradas de datos.

GDAL también incluye la librería OGR que provee unas funcionalidades similares para tipos de datos vectoriales.

2.11. cURL

cURL [htt97] es, a la vez, un programa y una librería open-source realizada en el lenguaje C. Su propósito es la transferencia de archivos e información a través de diversos protocolos: FTP, FTPS, HTTP, HTTPS, TFTP, SCP, SFTP, Telnet, DICT, FILE y LDAP, así como diversos tipos de certificados. En la actualidad es una de las librerías más usadas a bajo nivel para realizar peticiones y transferencias de archivos..

3

Análisis del problema

A continuación se expondrán los objetivos del proyecto y se realizará un análisis de los conceptos y el diseño empleado en mWorld. Finalmente se comentará las problemáticas del desarrollo en Android, así como las decisiones para emplear mWorld en esta plataforma.

3.1. Objetivos del proyecto

Este proyecto tiene como objetivo principal la creación de un framework con las siguientes características:

- Multiplataforma.
- Fácil integración con software ya existente.
- Arquitectura basada en plugins.
- Acceso remoto.
- Acceso a bases de datos.
- Acceso a bases de datos geolocalizadas.

- Capacidad de representación gráfica tridimensional.

- Gestión de datos out-of-core.

A este framework se le ha dado el nombre “mWorld” 3.1

Figura 3.1: Logo del framework mWorld.

3.2. Diseño de un framework multiplataforma

Desde un punto de vista comercial, un programa ha de intentar aumentar su base comercial lo máximo posible sin que eso aumente de forma considerable el coste del desarrollo. En el ámbito de los ordenadores de sobremesa nos encontramos con una gran base de usuarios que, históricamente, han empleado de forma mayoritaria sistemas operativos de la familia Windows.

Al mismo tiempo, en los últimos años, la base de usuarios de equipos Apple; con su sistema operativo OSX, y el gran número de personas que están optando por distribuciones Linux ha crecido de forma significativa. Convirtiendose en un mercado potencial para la venta de un número más amplio de aplicaciones que hace una década.

En principio, podría parecer que; con tres sistemas operativos, somos capaces de alcanzar a la mayor parte de usuarios potenciales. Sin embargo, debido al auge social que han alcanzado los llamados "teléfonos inteligentes", así como su capacidad de cómputo y representación gráfica de escenas tridimensionales; los convierten en un mercado potencial para el desarrollo de aplicaciones.

En el ecosistema de teléfonos móviles nos encontramos con tres potenciales alternativas: Android, iOS y Windows Phone.

En el ámbito de los ordenadores de sobremesa, es común encontrarse con librerías y aplicaciones multiplataforma. En la actualidad no existen grandes diferencias que impidan mantener una compatibilidad a nivel de código. Aún existen algunas áreas que provocan conflictos: manejo de hilos y procesos, interfaces gráficas, etc. Es común encontrarse librerías que ofrecen una abstracción de estos procesos críticos facilitando la creación de aplicaciones y librerías multiplataforma.

En el caso de los diferentes sistemas operativos para móviles, este proceso no es tan sencillo. Cada uno de ellos se comporta como un ecosistema totalmente diferente. Existen determinadas reglas, requerimientos y requisistos que son diferentes para cada uno de ellos.Esto dificulta mucho la reutilización de código entre diferentes plataformas.

mWorld intenta paliar los problemas derivados del desarrollo multiplataforma con una metodología de trabajo fija y única para todas las plataformas. El framework está compuesto por una librería core que da la mayor parte de funcionalidad del sistema, una serie de librerías opcionales y un runtime para cada plataforma soportada. Además el framework permite, mediante un sistema de plugins, extender las posibi-

lidades de acceso a datos, uso de librerías externas, etc.

3.3. Conceptos de mWorld

Como se puede observar en la figura 3.2, el framework mWorld está compuesto por una librería core que provee el núcleo de funcionalidad y gestión que se da a cualquier aplicación mWorld. Una serie de librerías de apoyo al programador y una serie de plugins que se pueden invocar en tiempo de ejecución en una aplicación mWorld.

Figura 3.2: Los componentes de mWorld están distribuidos entre un runtime (mwLauncher) la librería core (mwCore) una serie de librerías de apoyo y una serie de plugins.

Una aplicación o un plugin de mWorld se genera utilizando la librería mwCore. Esta librería ofrece una API pública de clases y servicios que el usuario puede emplear. Más concretamente, mwCore, ofrece al desarrollador dos clases, Application y Plugin que permiten al desarrollador derivar su comportamiento por defecto para implementar su propia aplicación o plugin como proceda.

Esto es así porque el framework mWorld no genera un código binario ejecutable. Se emplea un ejecutable runtime para cargar una aplicación mWorld que haya sido derivada de la clase Application de mwCore. Este lanzador de aplicaciones es mwLauncher. Como se puede ver en la figura 3.3 mwLauncher es una encapsulación de la librería mwCore junto a una parte específica para cada sistema operativo. Esta parte realizará las siguientes operaciones:

- Carga de librerías, aplicaciones y plugins solicitados en tiempo de inicialización
- Inicialización de las aplicaciones y plugins cargados
- Inicialización de las estructuras que encapsulen funcionalidades del sistema operativo.
- Inicializar el bucle de aplicación.

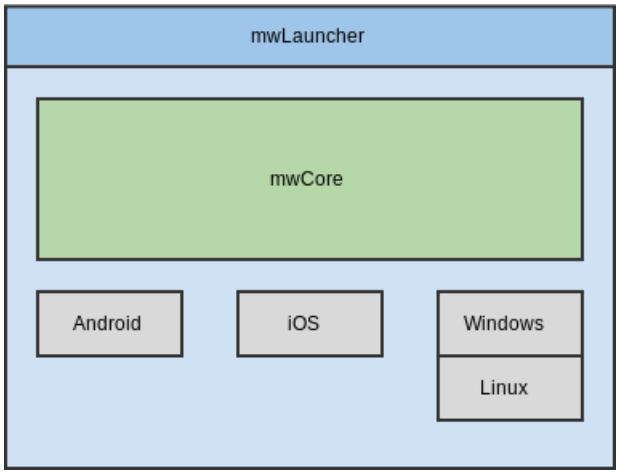

Figura 3.3: mwLauncher es un fichero ejecutable que encapsula la librería core junto a una serie de funciones que adaptan su ejecución para cada plataforma

Esta solución permite ofrecer al desarrollador una gran libertad para emplearla en sus proyectos. Puede generar su aplicación y realizar un deploy con el lanzador mwLauncher como aparece ilustrado en la figura: 3.4 ; o, por el contrario, puede integrar la librería mwCore dentro de su aplicación. De esta manera, su aplicación se convierte en el lanzador en sí mismo integrándose homogeneamente como se puede observar en la figura 3.5. En este caso el desarrollador podría tener dentro de su aplicación una aplicación mWorld o una serie de plugins bien, internos del proyecto, bien externos.

Esto nos permite dar completa libertad a los desarrolladores para integrar el framework con su entorno de trabajo y sus diferentes proyectos.

3.4. Librerías, plugins y drivers

En este punto trataremos los tres tipos de elementos propios del framework: las librerías, los plugins y los drivers.

3.4.1. Librerías

Como se ha expuesto en el punto 3.3 el framework mWorld busca facilitar su empleo permitiendo tanto su uso "stand-alone" mediante una aplicación runtime como mediante la posibilidad de utilizar la librería core como parte de una aplicación. Si bien, en este caso, el programador deberá dar soporte específico para cada plataforma

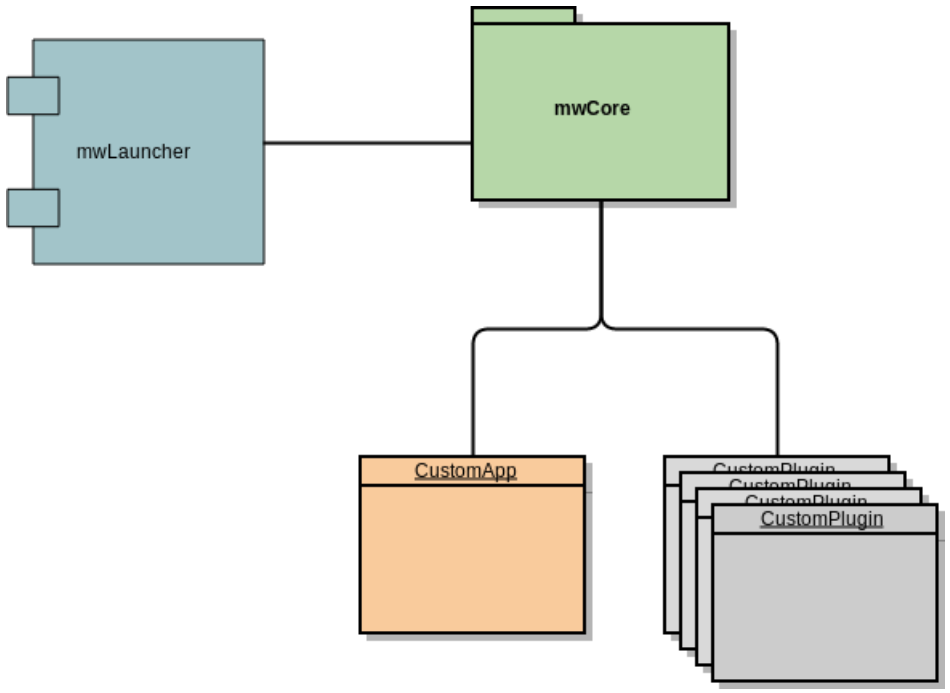

Figura 3.4: Diagrama de funcionamiento del runtime mWorld. El runtime ejecuta aplicaciones y plugins compatibles que utilizan las estructuras proporcionadas por la librería core de mWorld

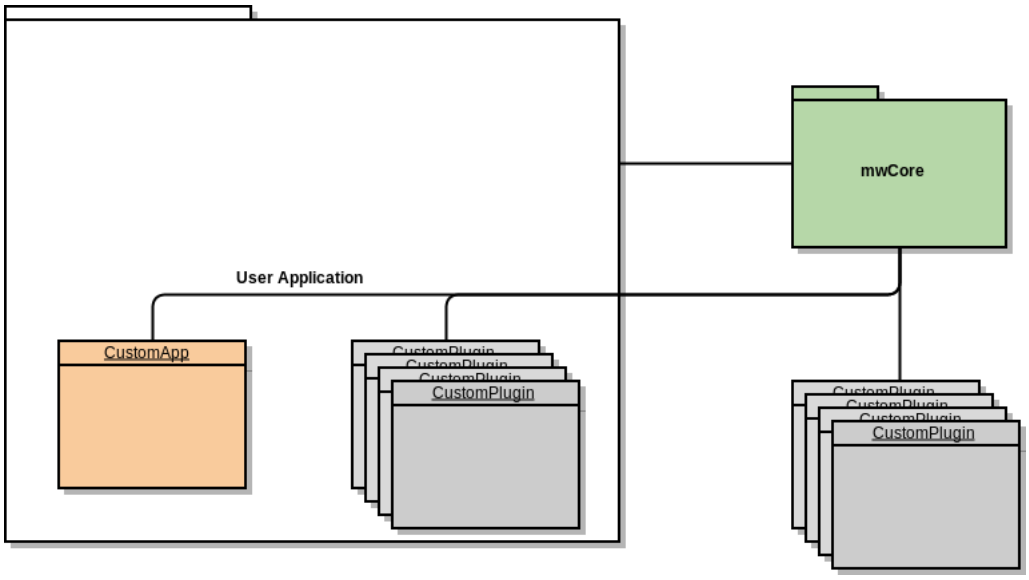

Figura 3.5: Diagrama de funcionamiento de una aplicación que integra el runtime mWorld. Puede realizar la ejecución de aplicaciones o plugins que haya creado o utilizar otros plugins externos

destino de su aplicación final.

Aunque este framework; y más concretamente, las partes tratadas en este trabajo final de master, tengan como foco central la representación de datos GIS, este no es el propósito del framework. mWorld es un framework abierto sin ningún tipo de foco. Precisamente por esto la única librería que hemos señalado en este análisis es la librería core. El objetivo es que mWorld se expanda con diferentes librerías multidiscipinares creadas con el core del framework y que provean con funcionalidades agnósticas a cualquier plataforma.

En el momento de redacción de esta memoria, el framework cuenta con las siguientes librerías:

- mwCache
- mwDataAcces
- mwGLM
- mwGIS
- mwRenderer

Estas librerías sirven de apoyo al desarrollo de aplicaciones y pueden ser utilizadas directamente por las aplicaciones mWorld del desarrollador o a través de plugins.

3.4.2. Plugins

En el diseño de esta librería, hemos querido poner un gran énfasis en la modularidad. Así pues, una aplicación mWorld no necesita implementar o incluir funcionalidades de forma explícita. La aplicación puede pedir la funcionalidad de un plugin en tiempo de ejecución. Si se consigue encontrar el plugin que implementa la funcionalidad, se incluye en el bucle de ejecución. Si no, la aplicación puede seguir funcionando con normalidad. Esto abre una serie de posibilidades muy interesantes al desarrollador. Tanto para desarrollos open-source como para comerciales.

Por un lado el tamaño de las aplicaciones desciende significativamente y solo se cargan las partes que el usuario necesite bajo demanda. Esto es muy importante de cara al uso de una aplicación en un entorno con limitaciones de memoria. No solo eso, este tipo de comportamiento evita el bloqueo del desarrollo de una aplicación al no depender directamente del código del plugin.

También hay que señalar las ventajas, comerciales, que ofrece: El desarrollador puede vender la licencia de una aplicación junto a todos sus plugins, o licenciar un subconjunto de ellos.

3.4.3. Drivers

Mientras que los plugins proveen a los programas de funcionalidad y acceso a librerías de forma indirecta, los drivers sirven para proveer a los programas de un acceso abstracto a datos. Cualquier aplicación o plugin mWorld puede comprobar la existencia de un driver concreto para acceder a un tipo de datos en tiempo real. Si es posible se devuelve el driver capaz de realizar el acceso.

No solo eso, un driver es capaz de realizar la misma comprobación para obtener un driver que trate un tipo de datos determinado. Esto permite generar "tuberías" de procesado de datos que implementen funcionalidades muy complejas con un bajo coste. En la figura 3.6 se puede ver un ejemplo real de su uso en la aplicación de ejemplo y validación que se ha creado para este trabajo. La aplicación utiliza una clase WMS Layer (Que hereda de la clase Raster Layer) y que gestiona las peticiones de tiles a un servidor WMS. Para ello emplea el driver WMS, el cual se encarga de gestionar la comunicación contra un servicio de este tipo y devolver una imagen procesada convenientemente. Para ello, el driver WMS, a su vez, emplea los drivers cURL y GDAL. Por un lado realiza las peticiones de comunicación y trozos de capas utilizando el driver cURL que le permite obtener datos de servidores remotos. Por otro lado, la información de tiles obtenida en la consulta WMS se procesa mediante el driver de GDAL. Este recibe el archivo obtenido y devuelve una extensión de la imagen original como datos RGBA. En el caso que nos ocupa, la extensión que el driver WMS solicita al de GDAL siempre abarga la totalidad de la tile que se ha pedido originalmente.

De esta forma, la tarea de negociación, acceso, petición y procesado de los datos de un servidor WMS se simplifica. Se divide la tarea global en pequeñas partes que hacen uso de elementos que ya nos ofrecen estas funcionalidades. Así, la clase WMS Layer emplea el driver WMS para obtener, en tiempo de ejecución, parches de terreno determinados por la vista de la aplicación. No necesita ocuparse del complejo proceso de negociación de versión con el servidor, ni del acceso a servidores remotos, ni de reinterpretar los datos de imagen como datos RGBA. Únicamente tiene que realizar peticiones contra un driver que le permite comprobar si ha sido posible conectarse, las capas a las que puede acceder y la extensión que puede pedir. A su vez, este driver evita reimplementar la rueda delegando el acceso remoto a un driver que emplea la

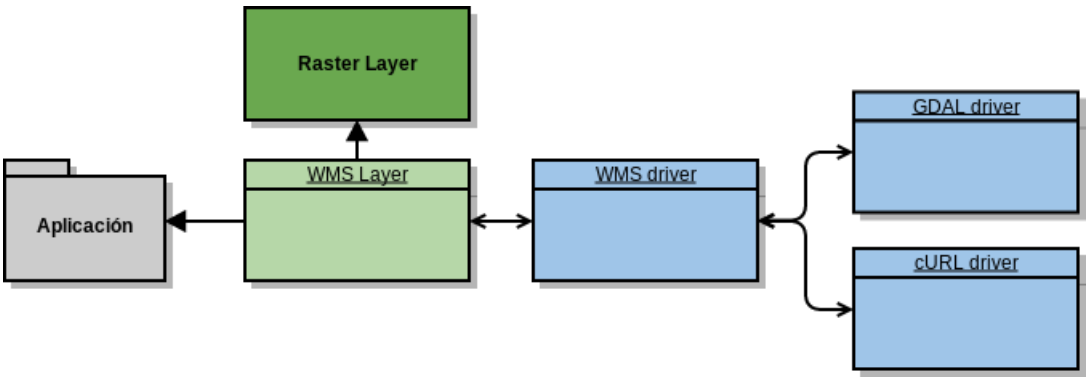

Figura 3.6: Diagrama de funcionamiento de una aplicación que integra el runtime mWorld. Puede realizar la ejecución de aplicaciones o plugins que haya creado o utilizar otros plugins externos

librería cURL y el procesado de imágenes a la librería GDAL.

De esta forma, tres elementos separados, e independientes, se pueden emplear para generar comportamientos muy complejos; con la ventaja de que, al gozar de independencia entre ellos, posteriormente se pueden sustituir estos componentes por otros que cumplan la misma interfaz y función.

3.5. La plataforma Android

Los dispositivos Android tienen una gran variedad de tamaño, forma y propósito. Aunque es posible encontrarse dispositivos anclados permanentemente a la red, o dispositivos cuyo uso está preparado para el salón de casa[1], Los dispositivos Android son, eminéntemente, móviles. Desde relójes inteligentes, pasando por teléfonos, tabletas y portátiles.

A la hora de desarrollar una aplicación en Android, hay que tener muy presente el tipo de dispositivos a los que va dirigida. Son dispositivos que dependen de una batería y, por lo tanto, una aplicación no puede mantenerse de forma indefinida en un segundo plano consumiendo recursos. Así pues, es importante que el grueso de la aplicación, como tal, solo se mantenga activo cuando está visible. Si se necesita algún tipo de servicio secundario para consulta de datos, temporizador, etc, es conveniente mantenerlo como una pieza separada que consuma pocos recursos.

[1]Este es el caso de Ouya, uno de los primeros dispositivos con el propósito de emplearse como una consola de videojuegos y media center en el salón de casa.

Por otro lado, existen limitaciones tanto para el uso de memoria, como para el almacenamiento de datos. Los dispositivos móviles no tienen la misma capacidad de almacenamiento, y en muchos casos, las características como la latencia de acceso a la memoria terminan convirtiéndose en el cuello de botella de una aplicación.

Android es un sistema operativo que ofrece muchas posibilidades al desarrollador. Es posible llegar a crear aplicaciones con un grado de complejidad muy elevado a la par que permite realizar optimizaciones muy específicas para las aplicaciones. El funcionamiento general y la creación de aplicaciones se ha tratado previamente en el punto 2.2.2. Así pues, en los siguientes puntos trataremos las principales problemáticas que hemos tratado durante el desarrollo de mWorld y la adaptación para su uso en Android.

3.5.1. Coexistencia de las aplicaciones en el entorno operativo

Android es un sistema multitarea para su utilización en móviles. En un sistema multitarea normal, una aplicación puede consumir todos los recursos del sistema. Este tipo de comportamiento "greedy" de las aplicaciones, aunque no es deseable, ha sido una constante habitual en los sistemas de sobremesa. Android, al estar diseñado para dispositivos móviles da mayor importancia a la estabilidad y responsibidad del dispositivo. Para ello, incluye algunas ideas típicas de los sistemas operativos orientados a procesos críticos. El sistema operativo Android tiene una serie de límites temporales y de memoria que intentará cumplir.

El control de este sistema operativo para conocer si una aplicación sigue respondiendo al usuario se basa en el control de los eventos. Toda acción del usuario sobre el dispositivo genera un evento que es encolado, a bajo nivel, en una cola de eventos propia para cada aplicación. Si la aplicación no vacía esta cola, en una cantidad de tiempo fijo, la máquina virtual Dalvik entiende que la aplicación no está respondiendo al usuario y lanza una excepción ANR. Así pues, una aplicación debe procesar los eventos en el menor tiempo posible, y a ser posible, de forma no bloqueante.

Por otro lado, la máquina virtual de Android controla el uso de la memoria del sistema. En anteriores versiones del sistema operativo, las aplicaciones tenían un límite fijo de memoria definido por el fabricante. Este límite de memoria solo se aplicaba a nivel de la parte de memoria controlada por el recolector de basura. En la actualidad, es posible solicitar y comprobar en tiempo de ejecución la cantidad exacta de memoria controlada que puede disponer nuestra aplicación, así como solicitar una mayor cantidad.

Si nuestra aplicación necesita una mayor cantidad de memoria, entonces hemos de recurrir a la memoria no manejada por el recolector de basura de Android. Ambas memorias cuentan para el cómputo de memoria del proceso, pero a diferencia de la primera, no existe un límite prefijado. Sin embargo, que no exista un límite, no es un indicativo para pedir toda la memoria del sistema. Aunque esta memoria no está "manejada", el sistema operativo controla todo el uso de memoria, y si en algún momento la cantidad de memoria libre llega a determinados límites, este control cierra las aplicaciones no esenciales.

En la actualidad, las herramientas de desarrollo de la plataforma permiten la creación de aplicaciones que funcionan sobre la máquina virtual y aplicaciones mixtas que usan funciones nativas. En este último caso, además es posible utilizar una api de "actividad nativa" con la cual se puede acceder, nativamente, a una gran parte de los componentes hardware. En este último caso, la aplicación deja de tener acceso a los componentes visuales de la plataforma. Esto se debe a que, al usar el contexto gráfico de forma nativa, no puedes utilizar de forma compartida el contexto gráfico que se emplea para el renderizado de estos elementos.

Así pues, la creación de una aplicación debe de tener en cuenta que convive en un ecosistema cerrado donde compite, en igualdad, con el resto de aplicaciones por unos recursos limitados:

- Contexto gráfico.
- Memoria física.
- Tiempo de cálculo.

Dependiendo de las circunstancias de una aplicación, el uso de los recursos se debe orientar más hacia la programación nativa o totalmente nativa. En el desarrollo de mWorld hemos decidido dar la libertad al desarrollador final, así pues, se implementarán dos versiones del lanzador: Una basada en el uso de la actividad nativa, y otra en la que se emplea una mezcla de programación Java y nativa.

El lanzador basado en la actividad nativa se empleará, principalmente, para la generación de programas que necesitan de acceso directo al hardware gráfico y que buscan utilizar un modo a pantalla completa. Este caso es típico en aplicaciones como videojuegos o reproductores multimedia, donde se busca evitar el intrusismo de la interfaz Android, las notificaciones visuales, etc.

Por otro lado el lanzador JNI, se empleará para generar aplicaciones que hacen uso de las posibilidades de la interfaz del sistema operativo junto a la aplicación, como por ejemplo:

- Aplicación en forma de widget para el sistema operativo.
- Acceso a la barra de notificaciones.
- Empleo de elementos UI Android.

3.5.2. Compatibilidad entre dispositivos móviles

Como se ha comentado previamente, la diversidad de dispositivos es un problema importante para el desarrollador. Si bien existen algunos componentes hardware que se han empleado en múltiples dispositivos, las diferencias en capacidad gráfica, potencia de cálculo o formato de pantalla aparecen durante la fase de testeo. Es muy normal encontrarse con dispositivos muy parecidos, incluso del mismo modelo, que se han implementado de formas muy distintas.

Es muy habitual encontrarse con dispositivos cuya tasa de refresco de pantalla está limitada. Esto se suele realizar para alargar la vida de la batería. Hay casos donde, directamente se ha rebajado el reloj del procesador para este mismo fin. Hay otros casos donde el fabricante busca ahorrar en costes de producción. Cada vez es más habitual que, componentes menores que suelen pasar desapercibidos, sean sustituidos para obtener mayores márgenes de beneficio.

Esto se traduce en que incluso dentro de dispositivos con la misma marca y nombre encontremos diferencias significativas al usar memorias con mayor latencia o módulos ligeramente diferentes. En algunos casos estas diferencias no afectan a una aplicación final, sin embargo son un factor desestabilizante que, por estadística y dada la gran base de usuarios y aplicaciones, termina emergiendo como problemas en el uso de aplicaciones.

En la plataforma Android, como se ha explicado en el punto 2.2.1, existe un archivo de manifiesto que describe la aplicación. Entre las opciones de descripción, se pueden incluir determinadas cláusulas que limitan los dispositivos que pueden instalar el paquete de la aplicación. Como estas reglas de rechazo se encuentran a nivel de paquete, el desarrollador puede resolver y optimizar su aplicación para diferentes dispositivos. Para ello distribuye varias versiones con manifiestos ligeramente distintos que permiten al repositorio de la plataforma, Google Play en la actualidad, determinar la versión

más compatible con el dispositivo del usuario que demanda la aplicación. De la misma manera, el usuario puede ver claramente cuando una aplicación no está soportada para su dispositivo por el desarrollador, aunque esto no significa que sea totalmente incompatible o incapaz de ejecutarla.

Para el desarrollo de mWorld se han fijado los siguientes requerimientos en los dispositivos:

- Android 4.0
- 1Gbyte de memoria ram.
- Procesador Arm V7 ¿1Ghz

3.6. Planificación del trabajo

Las siguientes figuras presentan la planificación del proyecto. El trabajo ha sido dividido en treinta tareas incluyendo la planificación inicial y la escritura de la memoria final del proyecto.

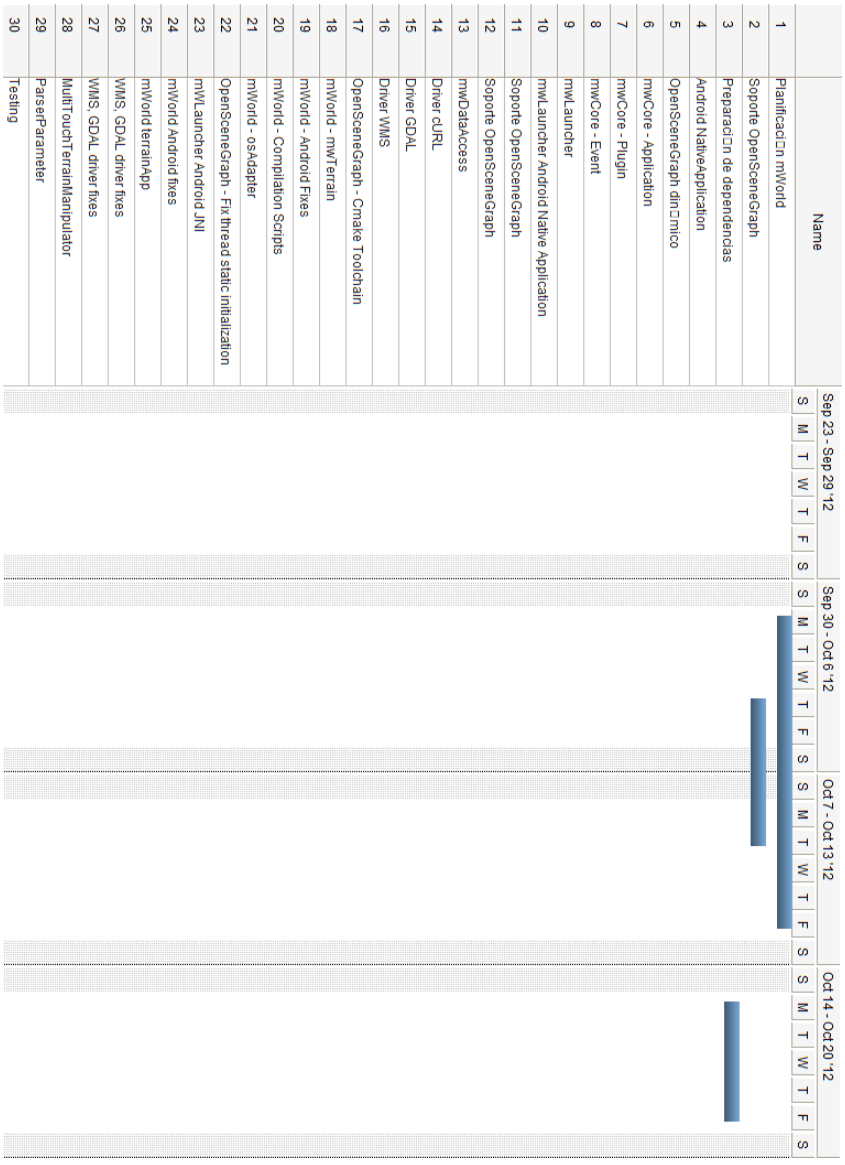

Figura 3.7: Diagrama de Gantt de la planificación del proyecto página: 1/11

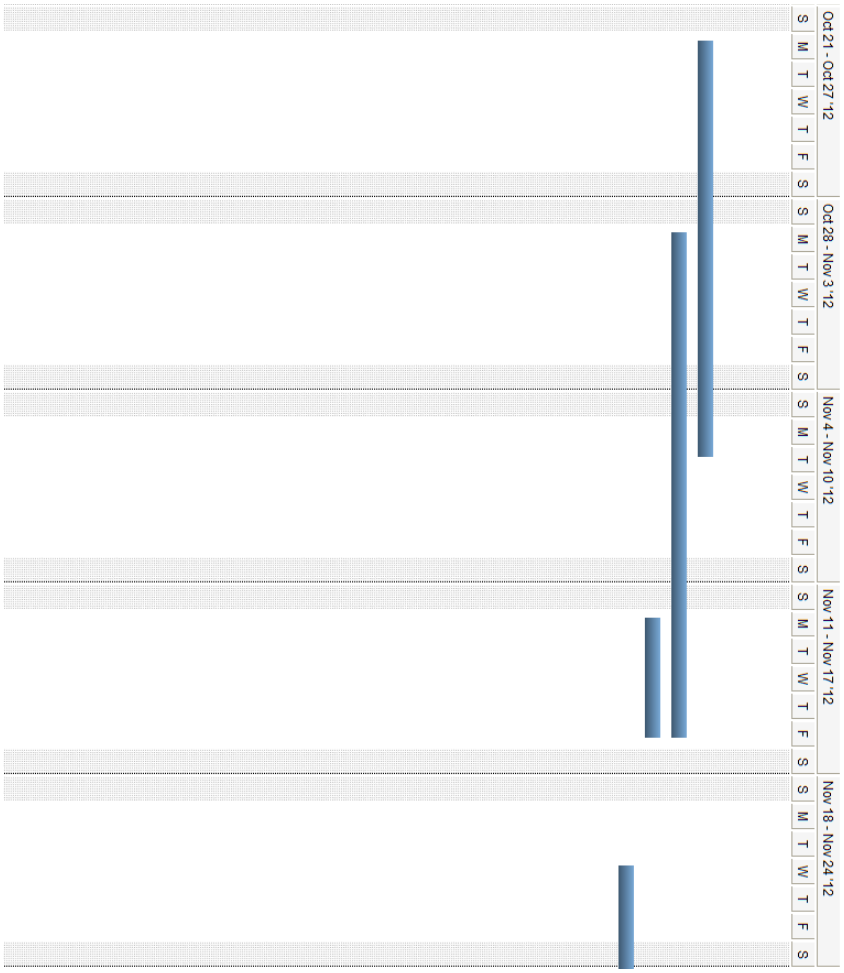

Figura 3.8: Diagrama de Gantt de la planificación del proyecto página: 2/11

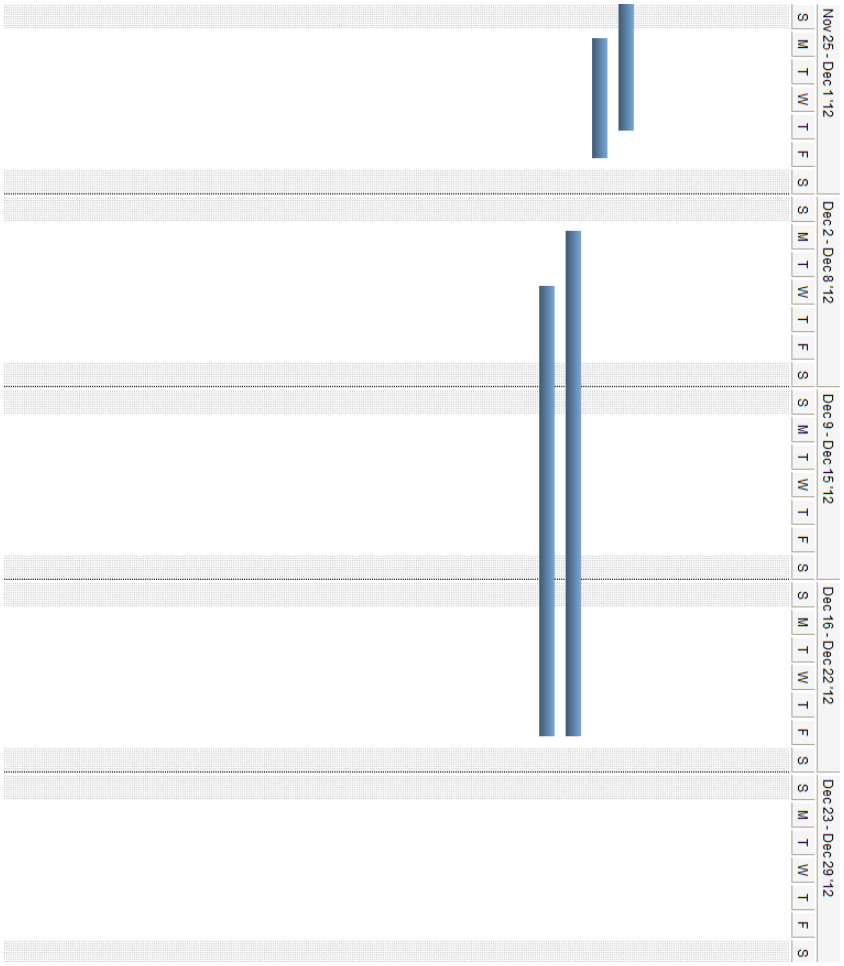

Figura 3.9: Diagrama de Gantt de la planificación del proyecto página: 3/11

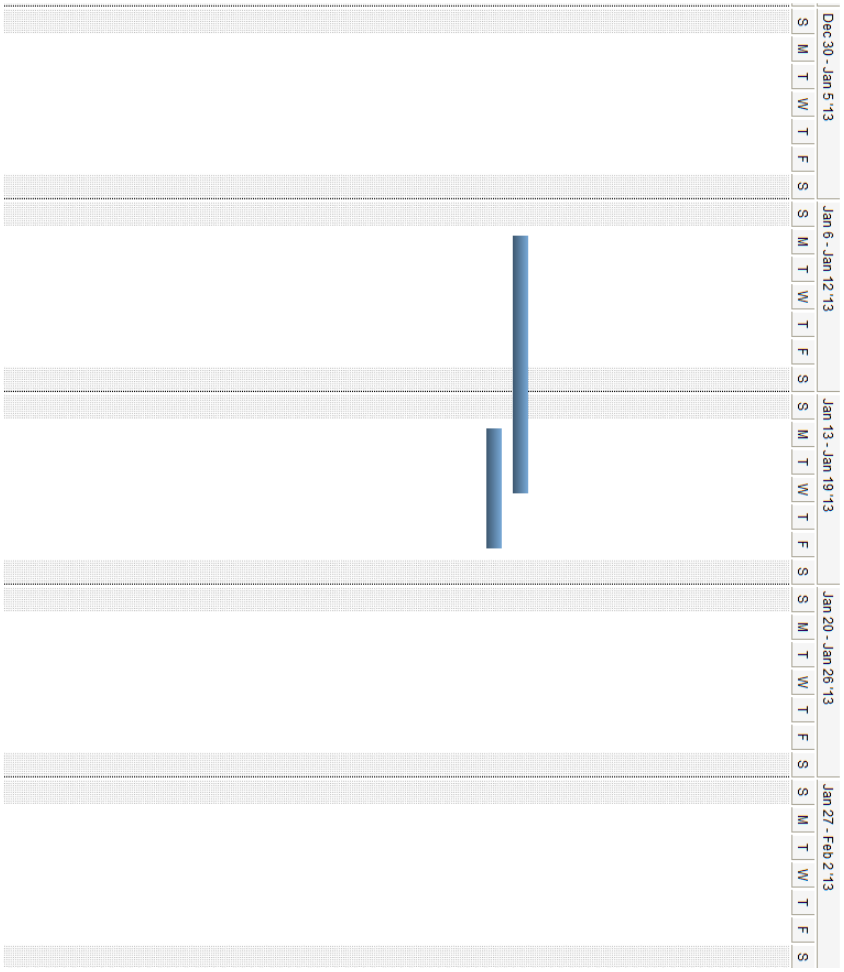

Figura 3.10: Diagrama de Gantt de la planificación del proyecto página: 4/11

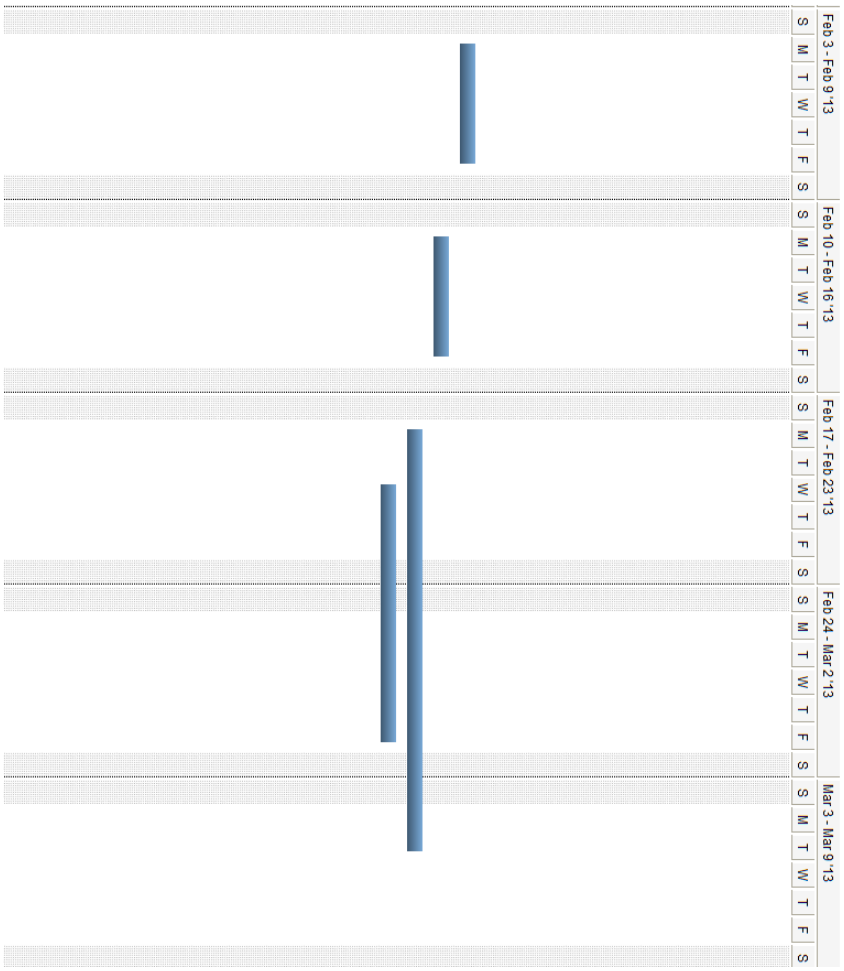

Figura 3.11: Diagrama de Gantt de la planificación del proyecto página: 5/11

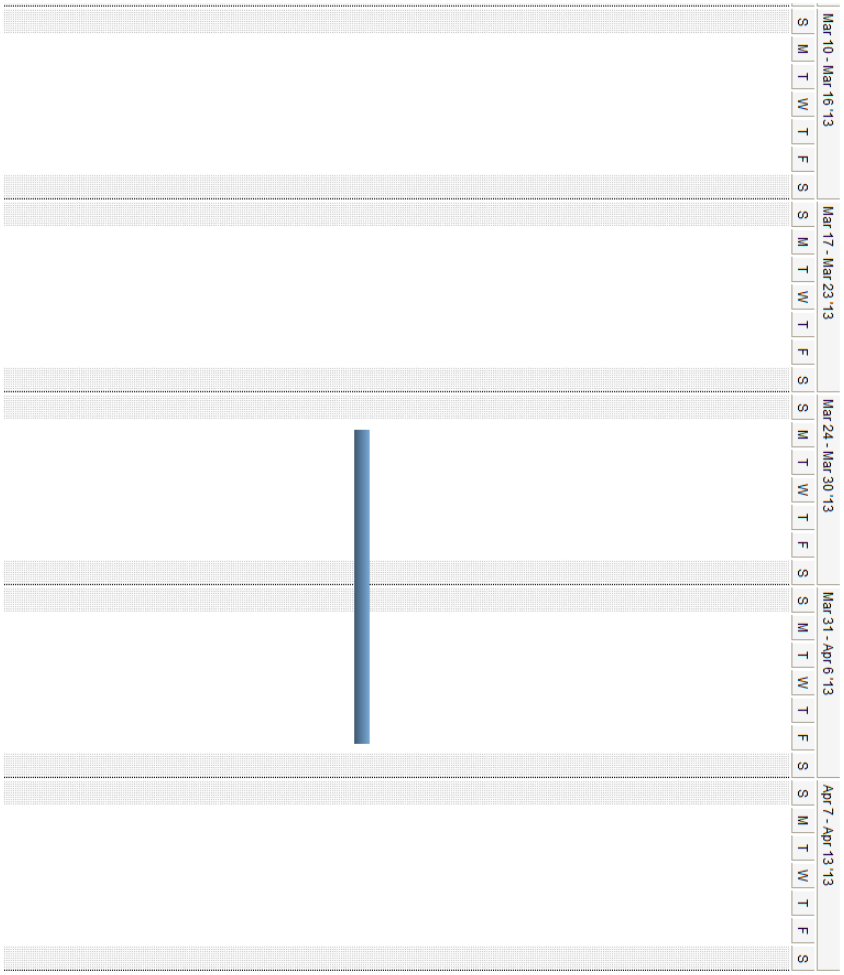

Figura 3.12: Diagrama de Gantt de la planificación del proyecto página: 6/11

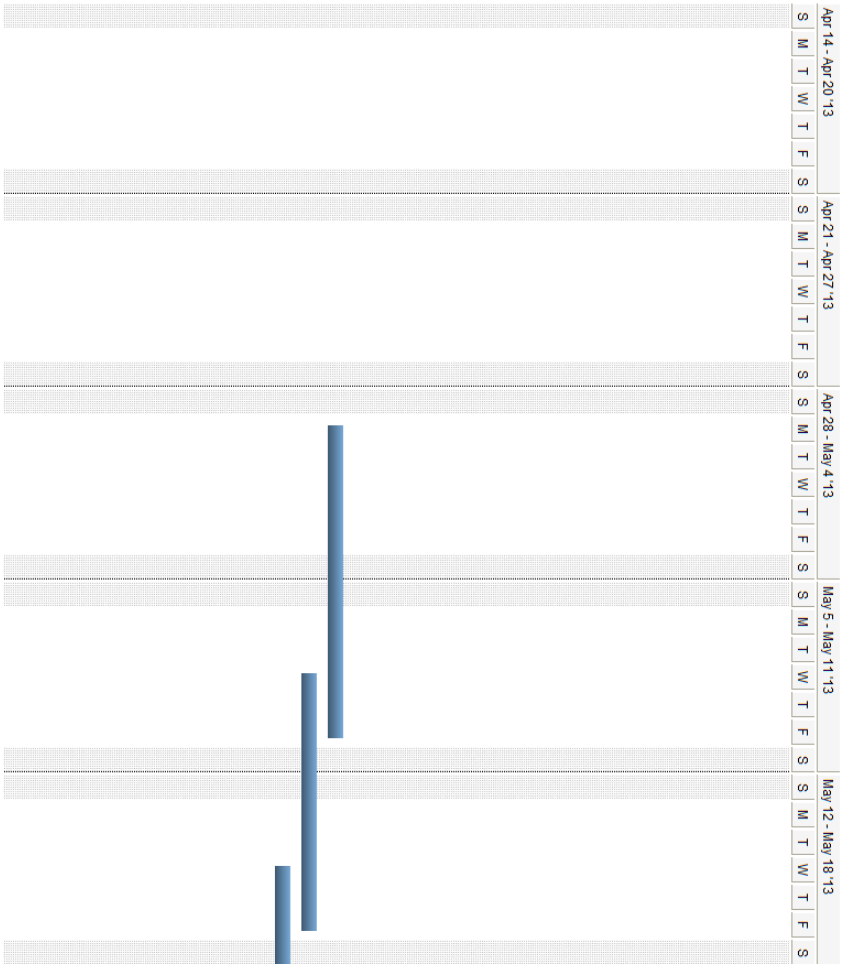

Figura 3.13: Diagrama de Gantt de la planificación del proyecto página: 7/11

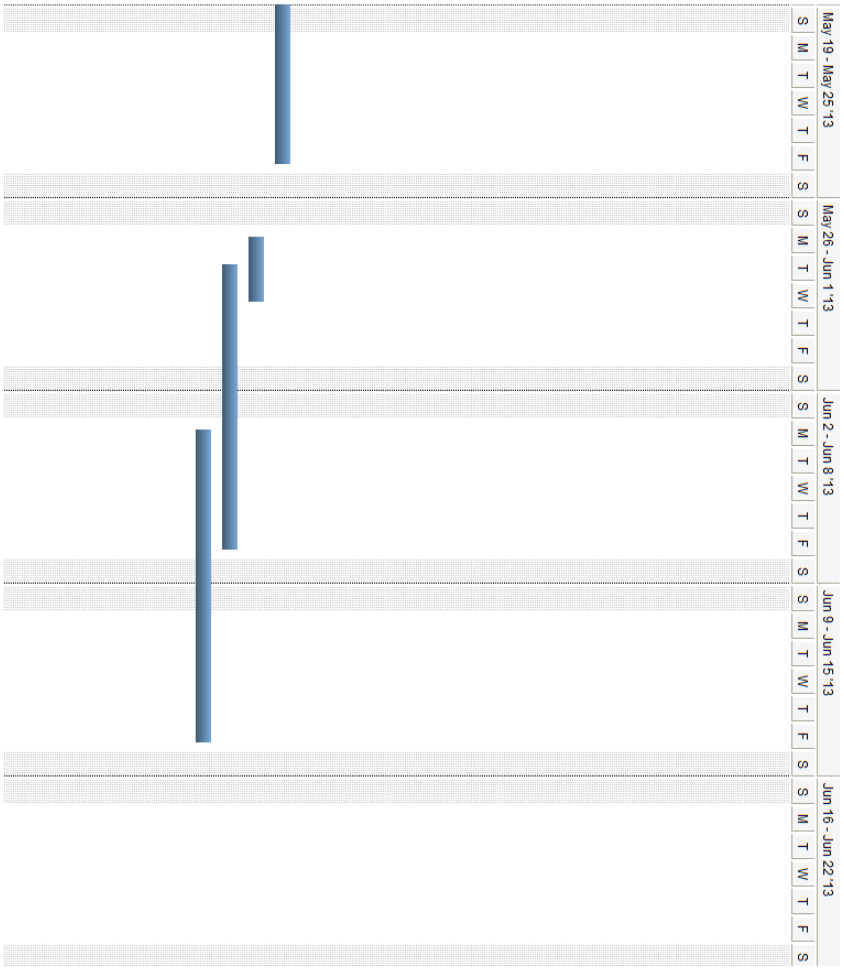

Figura 3.14: Diagrama de Gantt de la planificación del proyecto página: 8/11

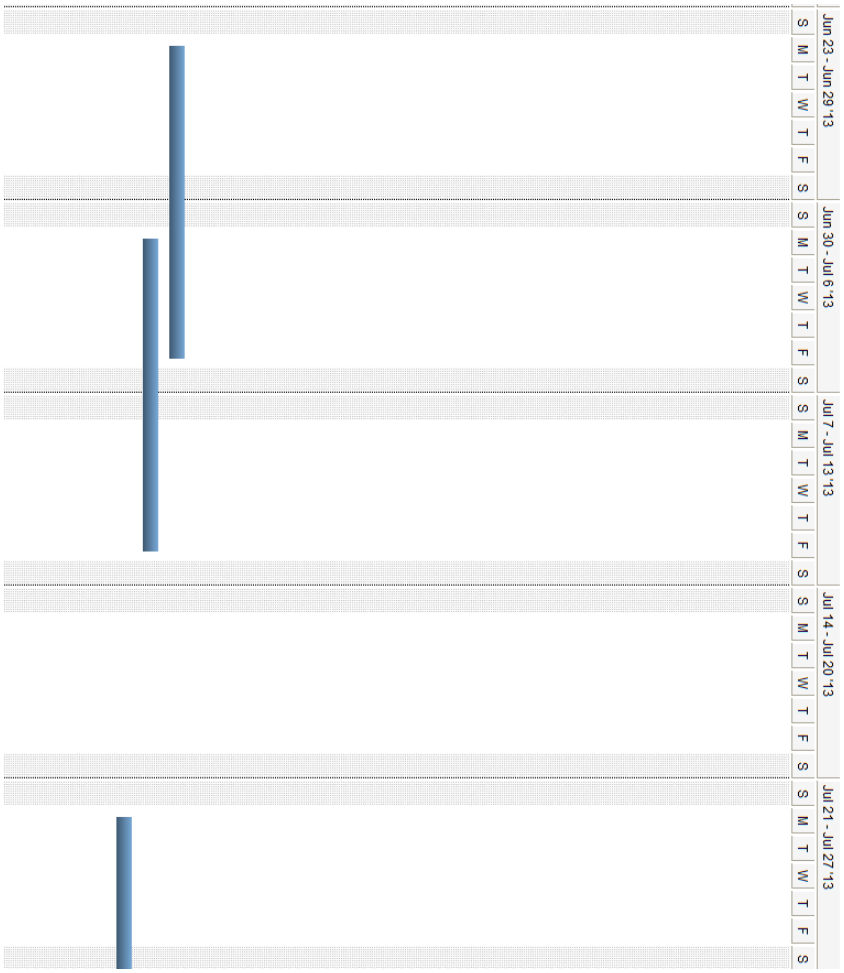

Figura 3.15: Diagrama de Gantt de la planificación del proyecto página: 9/11

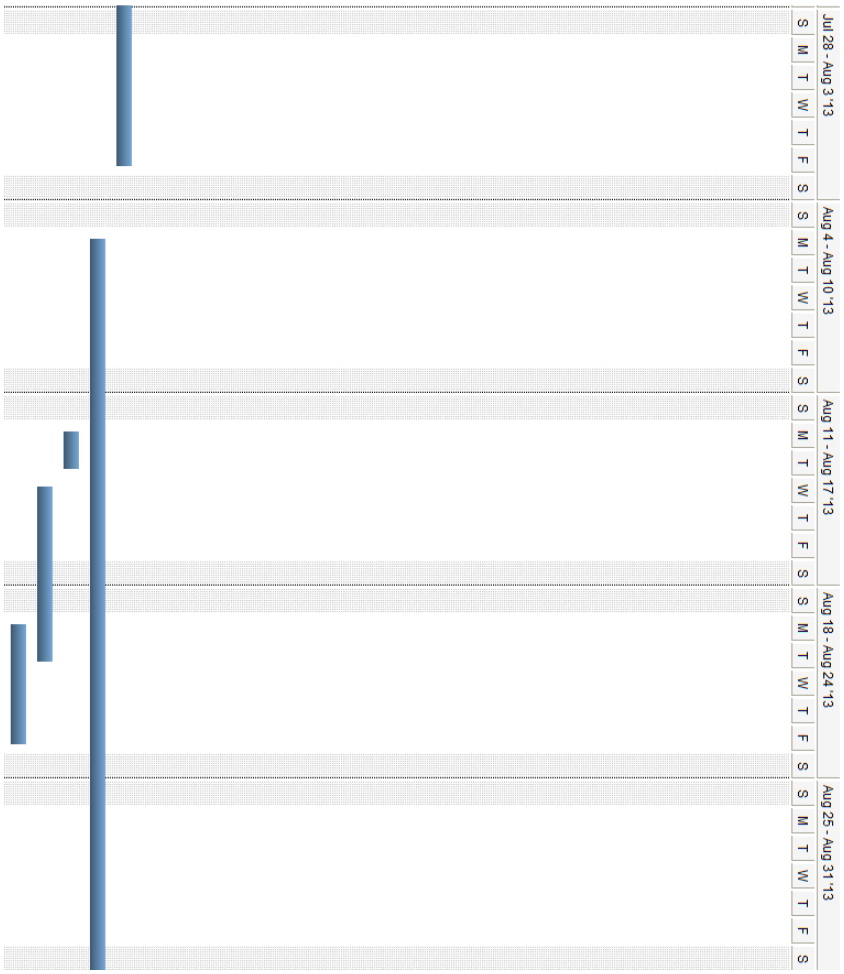

Figura 3.16: Diagrama de Gantt de la planificación del proyecto página: 10/11

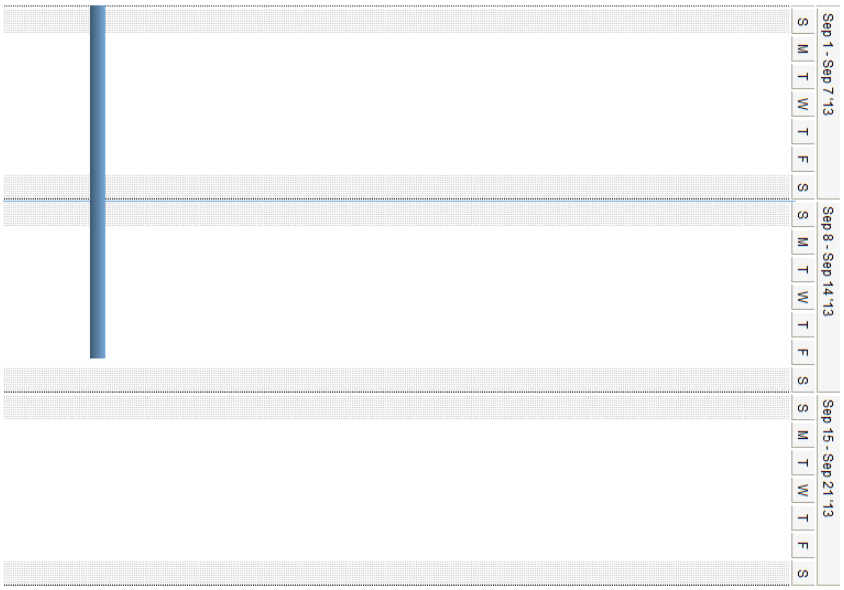

Figura 3.17: Diagrama de Gantt de la planificación del proyecto página: 11/11

4

Desarrollo

A lo largo de esta sección, se hablará de la implementación de los componentes principales del framework mWorld. Se tratarán los componentes más importantes que se han implementado, así como las implementaciones de los lanzadores mWorld tanto para las plataformas de sobremesa como para el sistema operativo Android. Terminaremos este capítulo hablando del desarrollo de la aplicación de prueba mWorld que emplea los diferentes elementos descritos en esta sección para representar un terreno bidimensional cargado en tiempo de ejecución desde un servicio WMS.

4.1. mwCore

La librería mwCore, es la librería central del framework mWorld. Está compuesta por una serie de clases que implementan la funcionalidad del framework, una serie de clases con funcionalidades de apoyo y finalmente una serie de clases templatizadas que se utilizan como base de algunos elementos de la librería.

La mayor parte de clases presentes en la librería sirven de apoyo al desarrollador y se emplean a lo largo del código. Un ejemplo de esto son las clases: RefPtr, Function, Logger, QualifiedPaths o ParameterParser.

RefPtr implementa el patrón conocido como “Puntero de referencia”. En este patrón todo objeto tiene asociado un contador de referencias, el número de veces que una va-

riable apunta al mismo objeto, y cuando dicho contador llega a cero referencias el objeto se borra automáticamente. Esto sirve para prevenir posibles fugas de memoria.

Nuestra implementación de punteros de referencia es una clase templatizada que tiene una serie de políticas de conteo diferentes para ajustar la responsabilidad de conteo de referencias, el tipo de almacenamiento del objeto y la política de comportamiento para trabajar con hilos.

Function, por su lado, es una implementación de functores genéricos. Un functor es un puntero que accede a una función en vez de a un dato. Es una clase que simplifica la notación de functores de C y que, a diferencia de los que ofrece, permite utilizarse con funciones no estáticas de cualquier tipo de clase.

El framework mWorld, utiliza la clase Logger para realizar el log de sucesos durante la ejecución. La clase Logger permite el reporte de sucesos con un modificador de importancia. Los niveles reconocidos son:

- LOG_ERROR
- LOG_WARNING
- LOG_NOTICE
- LOG_DEBUG
- LOG_VERBOSE

Estos niveles sirven para que el usuario pueda filtrar, bien mediante configuración de logger, bien mediante filtrado de la salida, los mensajes relevantes.

Además el Logger presente en la librería mwCore permite incluir diversos mecanismos que traten la entrada de Log. Esto se realiza mediante una serie de LogHandlers que se registran en la clase Logger. La clase LogHandler es una interfaz que el desarrollador puede derivar (O emplear una de las ya existentes) para decidir el destino, sintaxis, y tratamiento de los logs. Además como el Logger permite tener diferentes manejadores de la salida, este puede replicarse en diferentes fuentes de salida. Finalmente, también, hay que señalar que un mensaje de log puede ser enviado directamente a un único handler o a todos.

QualifiedPaths en una clase con utilidades para la búsqueda de ficheros, gestión de rutas de búsqueda de ficheros y carga de librerías.

Finalmente, la clase ParameterParser implementa la funcionalidad principal de parseado de una línea de parámetros compleja. Esta clase puede ser especializada para recopilar información con respecto a una serie de variables que decida el programador. En la figura 4.1 se puede observar la implementación realizada, utilizando esta clase, que sirve para obtener los parámetros notables pasados a la aplicación mwLauncher.

Código 4.1: Implementación de la clase LauncherParameterParser que sirver para parsear los argumentos del lanzador mwLauncher en dispositivos de sobremesa.

```
1  class MW_EXPORT LauncherParameterParser: public ↙
       ↳ mwCore::ParameterParserResultSet
2  {
3  public:
4      LauncherParameterParser()
5      {
6          mwCore::ValueParameter argLibs("l","libs", ↙
               ↳ "Load␣library␣by␣name");
7          mwCore::ValueParameter ↙
               ↳ argLibPaths("L","libPaths", "Load␣↙
               ↳ library␣by␣path");
8          mwCore::ValueParameter argMods("m","mods", ↙
               ↳ "Load␣module␣by␣name");
9          mwCore::ValueParameter ↙
               ↳ argModPaths("M","modPaths", "Load␣↙
               ↳ module␣by␣path");
10         mwCore::ValueParameter argPath("p","path", ↙
               ↳ "Paths␣to␣load␣libraries␣and␣modules");
11         mwCore::ValueParameter ↙
               ↳ argAppArgs("a","args", "String␣of␣↙
               ↳ arguments␣that␣are␣passed␣to␣the␣↙
               ↳ Application␣and␣Plugins");
12         mwCore::FlagParameter ↙
               ↳ argVerbose("v","verbose", "Flag␣to␣↙
               ↳ show␣debug␣mode");
13
```

```
14          addParameter(std::string("argLibs"), argLibs);
15          addParameter(std::string("argLibPaths"), ↙
                ↳ argLibPaths);
16          addParameter(std::string("argMods"), argMods);
17          addParameter(std::string("argModPaths"), ↙
                ↳ argModPaths);
18          addParameter(std::string("argPath"), argPath);
19          addParameter(std::string("argAppArgs"), ↙
                ↳ argAppArgs);
20          addParameter(std::string("argVerbose"), ↙
                ↳ argVerbose);
21      }
22
23      ~LauncherParameterParser()
24      {
25      }
26
27      std::vector<std::string> getLibsName(){return ↙
            ↳ _variables[std::string("argLibs")]._value;}
28      std::vector<std::string> getLibsPath(){return ↙
            ↳ _variables[std::string("argLibPaths")]._value;}
29      std::vector<std::string> getModsName(){return ↙
            ↳ _variables[std::string("argMods")]._value;}
30      std::vector<std::string> getModsPath(){return ↙
            ↳ _variables[std::string("argModPaths")]._value;}
31      std::vector<std::string> getPath(){return ↙
            ↳ _variables[std::string("argPath")]._value;}
32      std::vector<std::string> getAppArgs(){return ↙
            ↳ _variables[std::string("argAppArgs")]._value;}
33      bool                     getVerbose() { return ↙
            ↳ _flags["argVerbose"]._flag; }
34  };
```

A parte de estas clases que dan funcionalidad directa al desarrollador, la librería

también cuenta con una serie de clases templatizadas que permiten añadir funcionalidad a las clases que ya tenga en sus proyectos.

Entre estas clases nos encontramos:

- Singleton
- Properties
- Object factory

Así pues, es posible utilizar el template singleton para dotar, a una clase cualquiera, de la funcionalidad del patrón Singleton. En el fragmento de código 4.2 se puede ver un ejemplo de código que demuestra la sencillez de cara a su uso.

Código 4.2: Empleo del template Singleton para generar en tiempo de compilación una versión con el template singleton a partir de una clase.

```
1
2 typedef Singleton<ClassImpl> ClassSingleton;
3 ClassSingleton::getInstance();
```

Por otro lado, el template de Properties permite añadir a las clases la posibilidad de usar un sistema de propiedades. Este tipo de sistemas permtie al programador consultar, o llamar a métodos de la clase que no estén expuestos en la interfaz. De esta forma el programador no necesita recurrir a la RTTI de C++ para conocer el objeto real detrás de la interfaz.

Este template dota a una clase de funcionalidad para registrar, consultar y ejecutar propiedades. Una propiedad tiene asignado un nombre y una función con cero o un parámetros. De esta forma se puede emplear una propiedad para tener funciones getter/setter, así como para ejecutar algún tipo de funcionalidad de la clase. Todo esto sin que el código compilado necesite conocer la forma concreta de la clase, solo su interfaz.

La Object Factory es un template que nos permite crear factorías de objetos. Esta factoría de objetos nos permite registrar diferentes tipos de clases y luego, pedir que genere una nueva instancia de cualquiera de los tipos registrados.

Finalmente la librería mwCore tiene una serie de clases que dan la funcionalidad importante de cara a la ejecución de las aplicaciones mWorld y sus plugins:

- Registry
- Application
- Plugin
- Driver
- Event
- OSAdapter

4.1.1. Registry

El registry o registro es la clase central de la librería mwCore. Esta librería cumple dos propósitos muy importantes. Por un lado se encarga de establecer el orden de inicialización y destrucción de los elementos comunes. Por otro lado, esta clase sirve como registro para plugins, drivers y filtros. Es decir sirve para que todos los elementos con interfaz que se pueden cargar de forma dinámica se registren. De esta forma cuando el usuario pregunta por uno de estas interfaces con una implementación determinada, se consulta en el registro si se ha cargado, y en el caso de que no se haya cargado se realiza una búsqueda en los paths definidos en QualifiedPaths para encontrar una librería que se ajuste a la funcionalidad. La declaración de la clase Registry se puede ver en el código 4.3

Código 4.3: Definición de la clase Registry. Se pueden observar los diferentes métodos y miembros que permiten el registro y uso de diferentes elementos de código en tiempo de ejecución.

```
1
2 class MW_EXPORT RegistryImpl
3 {
4 private:
5     RegistryImpl(const RegistryImpl &rhs);
```

```
6       RegistryImpl &operator =(const RegistryImpl &);
7
8   public:
9       RegistryImpl();
10      virtual ~RegistryImpl();
11
12      typedef RefPtr<Application> ApplicationPtr;
13      typedef RefPtr<Plugin> PluginPtr;
14      typedef RefPtr<Library> LibraryPtr;
15      typedef std::vector<PluginPtr> PluginList;
16      typedef std::map<std::string, LibraryPtr > ↙
            ↳ LibraryMap;
17
18      void removePlugins();
19      void removeLibraries();
20
21      /** Getters **/
22      Application &getApplication();
23      const PluginList &getPluginList();
24      bool validApplication();
25
26      /** Registers **/
27      void registerApplication(ApplicationPtr appPtr);
28      void registerPlugin(PluginPtr pluginPtr);
29
30      /** File Paths **/
31      _private::FilePathImpl ↙
            ↳ &getResourceFilePathRegistry();
32      _private::FilePathImpl ↙
            ↳ &getLibraryFilePathRegistry();
33
34      /** Load a library and modules**/
35      bool loadLibrary(const std::string& libraryName);
36      bool loadModule(const std::string& libraryName);
```

```
37      bool loadLibraryPath(const std::string& ↙
            ↳ libraryNamePath);
38
39      bool moduleLoaded(const std::string  &moduleName);
40      bool libraryLoaded(const std::string ↙
            ↳ &libraryName);
41
42      /** Drivers **/
43      DriverRegistry &getDriverRegistry();
44
45      /** Filters **/
46      FilterRegistry &getFilterRegistry();
47
48      /** Logger **/
49      Logger &getLogger();
50      MTLogger &getMTLogger();
51
52      /** RefMap **/
53      inline _private::RefMap &getRefMap() { return ↙
            ↳ _refMap; }
54
55  protected:
56      Logger                          _logger;
57      MTLogger                        _mtLogger;
58      _private::RefMap                _refMap;
59      _private::FilePathImpl          ↙
            ↳ _resourceFilePathRegistry;
60      _private::FilePathImpl          ↙
            ↳ _libraryFilePathRegistry;
61      DriverRegistry                  _driverRegistry;
62      FilterRegistry                  _filterRegistry;
63
64      mwCore::RefPtr<Application> _mainApplication;
65      PluginList                      _pluginList;
```

```
66      LibraryMap                      _libraryMap;
67
68      RefPtr<OSAdapter>               _osAdapter;
69
70      bool                            _cleaned;
71
72      /** OS Adapter **/
73      friend OSAdapter*    _private::getOSAdapter();
74      friend void          ↙
            ↳ _private::setOSAdapter(OSAdapter* ptr);
75  };
```

4.1.2. Application

La clase Application (Aplicación) es la clase que todo programa mWorld debe derivar para poder ser utilizado en el runtime de mWorld. El usuario ha de implementar la funcionalidad de los métodos virtuales que se ejecutarán apropaidamente a lo largo del ciclo de vida de la aplicación mWorld. En la pieza de código 4.4 se puede ver la definición de la clase. Si nos fijamos, se puede ver que la aplicación del usuario puede ser inicializa con una cadena de argumentos. Como explico en el punto 4.3.1 el lanzador puede recibir una serie de parámetros de entrada, entre los cuales se encuentra uno (-a) que permite añadir una linea de comandos que se pasará integramente a la aplicacion. De esta manera se puede alimentar el inicio de la aplicación con una serie de parámetros que alteren la funcionalidad del programa.

Código 4.4: Definición de la clase Application. Toda aplicación mWorld debe derivar de esta clase.

```
1
2  class MW_EXPORT Application
3  {
```

```
4  private:
5      typedef std::vector<Event>        VEvents;
6      bool     _done;
7      bool     _initialized;
8      VEvents _vEvents;
9
10 public:
11     typedef std::vector<std::string> Arguments;
12
13     Application();
14     virtual ~Application();
15
16     virtual void init(const Arguments& args);
17
18     virtual void event(const Event& ev);
19     virtual void processEvent(const Event& ev);
20     virtual void resume();
21     virtual void run();
22     virtual void step(double simulationTime);
23
24     bool isDone();
25     void setDone();
26     void stop();
27
28 };
```

4.1.3. Plugins

Los plugins son trozos de programa opcionales que sirven para añadir funcionalidad a una aplicación mWorld. Un plugin no está pensado para recibir eventos de flujo de la aplicación, más allá de los eventos de comunicación y la llamada para hacer un paso de simulación.

Un plugin se registra, en el momento de su carga, en la clase Registry y se permite

una cadena de entrada como parámetros de inicialización. Se puede observar más detalladamente la declaración de la clase plugin en la pieza de ćodigo 4.5.

Código 4.5: Definición de la clase Plugin. Todo plugin desarrollado o distribuido con el framework que se quiera emplear en el ciclo normal de vida de la aplicación debe heredar de esta clase.

```
1
2  class MW_EXPORT Plugin
3  {
4  private:
5      typedef std::vector<Event>          VEvents;
6      VEvents _vEvents;
7
8  public:
9      typedef std::vector<std::string> Arguments;
10
11     Plugin();
12     virtual ~Plugin();
13
14     virtual void init(const Arguments& args);
15     virtual void destroy();
16
17     virtual void event(const Event &ev);
18     virtual void processEvent(const Event& ev);
19     virtual void step(double simulationTime);
20 };
```

4.1.4. Driver

La clase driver es una interfaz abstracta que emplean todos los drivers de acceso a datos. Como se puede ver en la pieza de código 4.6 provee de funciones necesarias para la apertura y cierre de un acceso a datos. Todos los drivers, bien sean interfaces o implementaciones finales, derivan de esta clase y pueden recuperados desde la clase Registry.

Código 4.6: Definición de la clase Driver. Todo driver de acceso a datos debe derivar de esta clase, o de una subclase.

```
1
2  class MW_EXPORT Driver: public virtual PropertySet
3  {
4  public:
5      enum { isSpatial = 0 };
6
7      enum OpenMode
8      {
9          READ,
10         WRITE,
11         APPEND
12     };
13
14     Driver();
15     virtual ~Driver();
16
17     /** Connects to the url/file or whatever
18       * necessary to open the connection
19       * (i.e. the file with the data)
20       **/
21     virtual bool open(const std::string ↙
           ↳ &connectionString, OpenMode mode) = 0;
22     virtual bool isOpen() const = 0;
23
24     /** Close the connection **/
25     virtual bool close() = 0;
26
27     /** Flush all data but without closing **/
28     virtual bool flush() = 0;
29
30     virtual std::string getFileExtension() const { ↙
           ↳ return std::string(); }
```

```
31
32 private:
33     /// Disallow copy constructor and assign operator
34     Driver(const Driver &);
35     Driver &operator=(const Driver &driver);
36 };
```

La interfaz Driver extiende a la clase PropertySet. Esto significa que una clase final derivada de Driver tiene la posibilidad de acceder, mediante consulta de propiedades a los parámetros y funciones de la clase mediante el nombre de la función, parámetro o propiedad. Esto permite acceder a la funcionalidad de la clase final que hay detrás de la interfaz sin necesidad de que el código tenga que compilar contra la otra librería.

4.1.5. Event

La clase Event permite encapsular los eventos de comunicación entre el launcher, la aplicación y los plugins. Esta clase permite incluir un enumerador definido por el programador con el código de evento y se puede incluir una estructura de datos previametne conocida, así como un functor que sirve de "callback" una vez se ha realizado el evento.

4.1.6. OSAdapter

La clase OSAdapter sigue una variación del patrón de diseño Adapter. Esta implementación de adaptador tiene dos partes. Por un lado el lanzador del sistema operativo en cuestión implementa las partes de código del sistema operativo que se quieren exponer, bien sea porque han de provocar una respuesta por parte la aplicación o bien porque la aplicación debe poder llamarlas. Por el otro lado, el adaptador tiene una serie de functores que se pueden registrar de cara a la aplicación o al sistema operativo. De esta manera cuando el sistema operativo ejecuta un evento controlado por mwLauncher, se dispara un trigger que llama a la función que haya registrado previamente la aplicación mWorld. De forma inversa, una aplicación mWorld puede pedir

al adaptador una determinada funcionalidad y este llamará a la función que se haya registrado para tal propósito.

4.1.7. Ciclo de vida de la aplicación mWorld

El bucle principal de una aplicación mWorld se genera derivando la clase de la librería mwCore: Application. Durante la ejecución de la aplicación, el runtime llamará a diferentes funciones de la clase siguiendo un ciclo de vida. En la figura 2.6 se puede ver dicho ciclo. Cuando una aplicacion se carga en el runtime, este ejecuta la función de inicialización.

Seguidamente, el runtime comienza un bucle de ejecución. En primer lugar se consultan los eventos que ha recibido la aplicación. Seguidamente se llama a la función step con un parámetro indicando el tiempo pasado desde la última ejecución de esta función. Una vez finalizada esta función se vuelve a ejecutar la función de procesado de eventos.

Existen eventos que pueden parar la ejecución del programa. Bien porque el usuario desee terminar la ejecución o porque el sistema operativo provoca una pausa. En este caso, se entra en la función stop. Cuando se quiere que el programa vuelva a continuar se llama a la función resume.

4.2. mwDataAccess

La librería mwDataAccess dota a los desarrolladores de herramientas para acceder, de forma abstracta, a diferentes tipos de datos. En concreto, la librería de acceso a datos extiende el driver básico que aparece en la librería core. En la actualidad añade cinco tipos de driver, así como una serie de plugins que facilitan el acceso a determinados tipos de datos empleando los drivers anteriormente mencionados.

4.2.1. Tipos de driver

Actualmente, la librería mwDataAccess ofrece las siguientes definiciones de drivers:

- DatabaseDriver
- IOStreamDriver

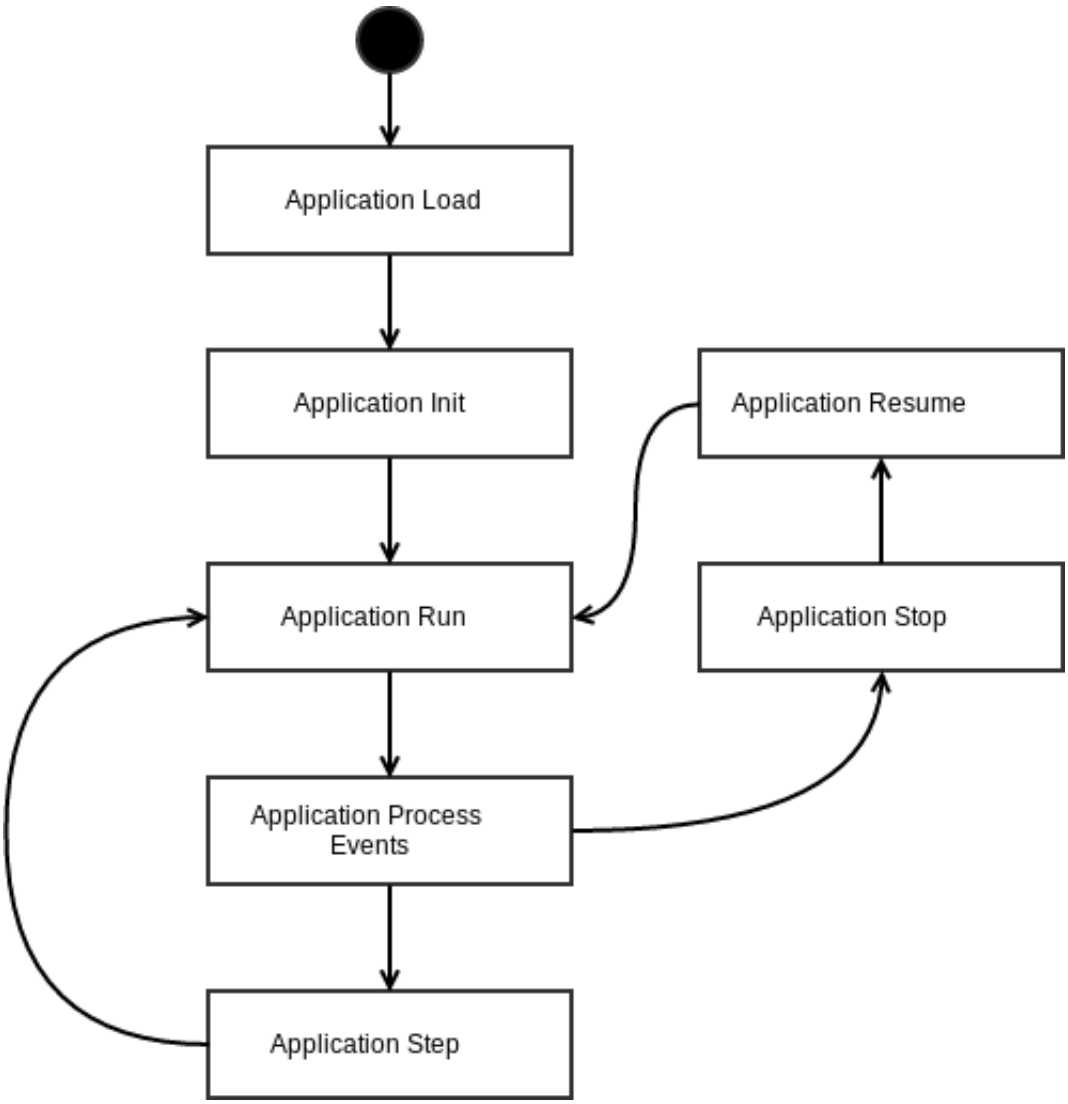

Figura 4.1: Ciclo de vida de una aplicación mWorld.

- BufferDriver
- RemoteDriver
- SpatialDriver

En la pieza de código 4.7 se puede ver la definición del driver de acceso a bases de datos. Este driver permite realizar consultas de dos tipos: (1) Consultas que ejecutan una serie de instrucciones y solo devuelven el estado (2) Consultas que ejecutan una serie de instrucciones y devuelven una estructura de datos indeterminada.

Código 4.7: Definición del driver de acceso a datos DatabaseDriver. Este driver está especializado para realizar consultas sobre bases de datos

```
1
2  class MW_EXPORT DatabaseDriver: public virtual ↙
       ↳ mwCore::Driver
3  {
4  public:
5      DatabaseDriver();
6      virtual ~DatabaseDriver();
7      /**
8        * Executes a query, only for selects,
9        * specific driver commands
10       * that can return a tabular Dataset
11       *
12       * @returns a Dataset to iterate
13       * over the retrieved data
14       */
15     virtual mwCore::RefPtr<Dataset> ↙
          ↳ executeQuery(const std::string &sql) = 0;
16     /**
17       * Executes a query, could be insert,
18       * update, create, etc that not return
19       * any data
20       *
21       * @returns true if execution was correct
22       */
23     virtual bool execute(const std::string &sql) = 0;
24 private:
25     /// Disallow copy constructor and assign operator
26     DatabaseDriver(const DatabaseDriver &);
27     DatabaseDriver &operator=(const DatabaseDriver &);
28
29 };
```

Este driver hace uso de la clase Dataset que está implementada en la librería mwDataAccess. Esta clase provee la funcionalidad de una estructura de datos genérica con forma tabular. Permite consultar los nombres de cada una de las propiedades del conjunto de datos y recuperar cualquier subconjunto de datos de la tabla.

El driver IOStreamDriver tiene como propósito el manejo de flujos de datos. Permite el uso tanto de lectura como de escritura. Estos flujos de datos pueden recibirse de tres formas: (1) Como un flujo de bytes y un tamaño (2) Como un Dataset (3) Como un Value.

Un Value es una clase que permite encapsular cualquier tipo de datos y proporciona automáticamente la serialización del tipo de datos que contenga.

Código 4.8: Definición del driver de acceso a datos IOStreamDriver. Este driver está especializado para la lectura y escritura de flujos de datos

```
1
2  class MW_EXPORT IOStreamDriver: public virtual ↙
       ↳ mwCore::Driver
3  {
4  public:
5      IOStreamDriver();
6      virtual ~IOStreamDriver();
7
8      virtual bool eof()=0;
9
10     virtual bool  seek(uint32_t pos) = 0;
11     virtual mwCore::RefPtr<mwCore::Value> read()=0;
12     virtual size_t read(char *data, size_t size)=0;
13     virtual mwCore::RefPtr<Dataset> readDataset();
14     virtual bool write(const mwCore::Value &value)=0;
15     virtual bool write(const Dataset &dataset)=0;
16     virtual size_t write(char *data, size_t size)=0;
17     virtual size_t size() const=0;
18 private:
19     /// Disallow copy constructor and assign operator
20     IOStreamDriver(const IOStreamDriver &);
```

```
21      IOStreamDriver &operator=(const IOStreamDriver &);
22  };
```

En la pieza de código 4.8 se puede examinar más detenidamente la declaración formal del driver en cuestión.

El BufferDriver, cuya declaración se puede observar en la pieza de ćodigo 4.9 es un driver que permite realizar la lectura y escritura de datos sobre un buffer.

Código 4.9: Definición del BufferDriver. Este driver está preparado para la lectura y escritura de datos a modo de buffer.

```
1
2  class MW_EXPORT BufferDriver: public virtual ↙
       ↳ mwCore::Driver
3  {
4  public:
5      BufferDriver();
6      virtual ~BufferDriver();
7
8      virtual void computeSubBuffer(double minX, ↙
           ↳ double minY, double maxX, double maxY)=0;
9      virtual void computeBuffer()=0;
10     virtual mwCore::Buffer getBuffer()=0;
11     virtual void setWidth(size_t width)=0;
12     virtual void setHeight(size_t height)=0;
13     virtual size_t width()=0;
14     virtual size_t height()=0;
15 private:
16     /// Disallow copy constructor and assign operator
17     BufferDriver(const BufferDriver &);
18     BufferDriver &operator=(const BufferDriver &);
```

```
19
20 };
```

El remoteDriver cuya declaración se puede ver en la pieza de código 4.10 está pensado para emplearse en el acceso de datos remotos. Da la posibilidad de incluir una serie de parámetros de conexión: Configuración de un proxy y autentificación de la conexión.

Los datos recuperados con este tipo de drivers se guardan en un buffer de datos, o en un stream de datos. Esto permite que un fichero accedido remotamente se pueda guardar directamente a disco duro entre otras cosas.

Código 4.10: Definición del driver para acceso remoto de datos.

```
1
2  class MW_EXPORT RemoteDriver: public virtual ↙
       ↳ mwCore::Driver
3  {
4  public:
5      RemoteDriver();
6      virtual ~RemoteDriver();
7
8      virtual bool read(std::ostream& output, const ↙
           ↳ std::string &sub_url = "" )=0;
9      virtual mwCore::Buffer read(const std::string ↙
           ↳ &sub_url = "")=0;
10
11     virtual void setProxy(const std::string& host, ↙
           ↳ const std::string& port)=0;
12     virtual void setAuthentication(const ↙
           ↳ std::string& userName,const std::string& ↙
           ↳ password)=0;
```

```
13 private:
14     RemoteDriver(const RemoteDriver&);
15     RemoteDriver& operator=(const RemoteDriver& ↙
           ↳ driver);
16 };
```

Finalmente, el spatialDriver es una clase templatizada que sirve para dotar de métodos a los tipos anteriores o a cualquier driver del usuario. Esto permite generar definiciones de drivers hibridos que incluyan peticiones estándar y peticiones delimitadas por una extensión. La definición del template se puede ver más detallamente en la pieza de código 4.11

Código 4.11: Definición del template SpatialDriver. Este template permite combinar drivers haciendo que incluyan funcionalidades espaciales

```
1
2  template<class T>
3  class SpatialDriver: public virtual T
4  {
5  public:
6      enum { isSpatial = 1 };
7      SpatialDriver() { }
8      virtual ~SpatialDriver() { }
9
10     /** Only valid for write, not for read */
11     virtual void setExtension(double minX, double ↙
           ↳ minY, double maxX, double maxY, double ↙
           ↳ minZ, double maxZ)=0;
12     /** Gets max extension **/
13     virtual void getExtension(double &minX, double ↙
           ↳ &minY, double &maxX, double &maxY, double ↙
           ↳ &minZ, double &maxZ)=0;
```

```
14      /** Query a specific extension **/
15      virtual void queryExtension(double minX, double ↙
            ↳ minY, double maxX, double maxY, double ↙
            ↳ minZ, double maxZ)=0;
16
17      virtual bool isWGS84() =0;
18      virtual bool isUTM() =0;
19      virtual const std::string& getProjectionString() ↙
            ↳ const=0;
20  private:
21      /// Disallow copy constructor and assign operator
22      SpatialDriver(const SpatialDriver &);
23      SpatialDriver &operator=(const SpatialDriver &);
24  };
```

Ademas de estas definiciones de drivers, la librería de acceso a datos mwDataAccess provee una serie de implementaciones para el manejo de algunos tipos de servicios o librerías. En el transcurso de este trabajo se han creado tres plugins: CURLDriver, WMSDriver y GDALDriver que se detallarán a continuación.

4.2.2. Driver CURL

El driver CURL dota de acceso a archivos remotos mediante el empleo de la librería cURL. Esta implementación utiliza la interfaz de un driver de acceso remoto. Y permite el acceso a archivos distribuidos en una red tanto de lectura como de escritura. La utilización de la librería cURL permite emplear los siguientes protocolos para acceder a archivos: FTP, FTPS, HTTP, HTTPS, TFTP, SCP, SFTP, Telnet, DICT, FILE y LDAP. La definición de este driver se puede consultar en el código 4.12. La implementación actual no permite el uso de las capacidades SSH.

Código 4.12: Definición del driver de acceso remoto CURLdriver. Este driver emplea la librería cURL para acceder a ficheros remotos.

```
1
2  class MW_EXPORT CURLDriver: public RemoteDriver
3  {
4  public:
5      CURLDriver();
6      virtual ~CURLDriver();
7
8      virtual bool open(const std::string &
           ↪ &connectionString,OpenMode openMode);
9      virtual bool close();
10     virtual bool isOpen() const;
11     virtual bool flush();
12
13     bool read(std::ostream& output, const &
           ↪ std::string &sub_url = "");
14     mwCore::Buffer read(const std::string &sub_url = &
           ↪ "");
15
16     virtual void setProxy(const std::string &host, &
           ↪ const std::string &port);
17     virtual void setAuthentication(const std::string &
           ↪ &userName, const std::string &password);
18 protected:
19     CURL*                   _curlHandle; //This &
           ↪ pointer is not a smartRef because it's a &
           ↪ void pointer
20     bool                    _isOpen;
21     std::string             _connectionString;
22     bool                    _newUser;
23     std::string             _userName;
24     std::string             _userPassword;
25     bool                    _newProxy;
```

```
26      std::string                 _proxyHost;
27      std::string                 _proxyPort;
28      unsigned int                _timeout;
29      mwCore::ReentrantMutex      _curlMtx;
30
31  private:
32      CURLDriver(const CURLDriver&);
33      CURLDriver& operator=(const CURLDriver& driver);
34  };
```

4.2.3. Driver WMS

Uno de los servicios más extendidos para la consulta de mapas geolocalizados son los WMS o Web Map Service. Un servidor WMS ofrece, mediante un servicio Web, el acceso a una serie de Layers (capas) con diferentes tipos de información geolocalizada. Esta información puede darse en forma de imágenes raster (png, jpeg, etc) o datos vectoriales (VGS).

La librería mwDataAccess está provista con un driver de acceso a los servicios WMS. En el código 4.13 se puede observar la definición del driver en cuestión. El driver WMS está definido como un SpatialBufferDriver, es decir un buffer de datos espacializados. Como se ha explicado en el punto 4.2.1 es posible combinar la defin-ción de un SpatialDriver con la de cualquier otro. Esto nos permite tener interfaces genéricas con capacidades de espacialización de datos.

Código 4.13: Definición del driver buffer-espacial que accede a servidores WMS.

```
1
2  class MW_EXPORT WMSDriver: public virtual ↙
       ↳ SpatialBufferDriver
3  {
```

```
4  public:
5
6      enum WMSVersion
7      {
8          WMS1_3_0,
9          WMS1_1_1
10     };
11
12     WMSDriver();
13     virtual ~WMSDriver();
14
15     virtual bool open(const std::string &
           ↳ &connectionString, OpenMode mode);
16     virtual bool isOpen() const;
17     virtual bool close();
18     virtual bool flush();
19     virtual std::string getFileExtension() const;
20
21     ///Buffer operations
22     virtual void computeSubBuffer(double minX, &
           ↳ double minY, double maxX, double maxY);
23     virtual void computeBuffer();
24     virtual mwCore::Buffer getBuffer();
25     virtual void setWidth(size_t width);
26     virtual void setHeight(size_t height);
27     virtual size_t width();
28     virtual size_t height();
29     ///Spatial operations
30     virtual void setExtension(double minX, double &
           ↳ minY, double maxX, double maxY, double &
           ↳ minZ, double maxZ);
31     virtual void getExtension(double &minX, double &
           ↳ &minY, double &maxX, double &maxY, double &
           ↳ &minZ, double &maxZ);
```

```
32      virtual void queryExtension(double minX, double ↙
            ↳ minY, double maxX, double maxY, double ↙
            ↳ minZ, double maxZ);
33      virtual bool isWGS84();
34      virtual bool isUTM();
35      virtual const std::string& getProjectionString() ↙
            ↳ const;
36
37  protected:
38      mwCore::RefPtr<RemoteDriver> _remoteDriver;
39      mwCore::RefPtr<SpatialBufferDriver> _imageDriver;
40      WMSVersion _protocolVersion;
41      std::string _serverWMS;
42      bool   _isOpen;
43      std::string _xmlCapabilities;
44
45      //Service
46      std::string _serverName;
47      std::string _maxWidth;
48      std::string _maxHeight;
49
50      std::vector<std::string> _supportedFormats;
51      std::vector<mwCore::RefPtr<WMSLayer> > _layers;
52
53      bool _layerChanged;
54      std::string _currentLayer;
55      std::string _currentCRS;
56      mwGIS::Envelope2Dd _currentBB;
57      bool         _crsChanged;
58
59      std::string _currentFormat;
60      std::string _queryWidth;
61      std::string _queryHeight;
62
```

```
63      size_t _width;
64      size_t _height;
65
66      bool   _newQueryExtension;
67      double _minX;
68      double _maxX;
69      double _minY;
70      double _maxY;
71
72      mwCore::Buffer _dataBuffer;
73
74      std::string _filePath;
75
76      bool negotiation();
77      void parseCapabilities();
78      void parseCapabilities_1_3_0();
79      void parseCapabilities_1_1_1();
80
81      void setService(WmsService serviceData);
82
83      std::string getFirstLayerName();
84
85      WMSLayer* findLayerByName(const std::string &name, WMSLayer* parentlayer = 0);
86      WMSLayer* findLayerByTitle(const std::string &name, WMSLayer* parentlayer = 0);
87
88  private:
89      WMSDriver(const WMSDriver &);
90      WMSDriver& operator=(const WMSDriver&);
91  };
```

Esto permite dar al usuario la posibilidad de solicitar una extensión determinada al servicio y recuperar la información. Para realizar este proceso, el driver requiere de

acceso remoto a datos. Para esto, el driver WMS solicita el uso del driverCURL. Como se ha comentado en el punto 3.4.3 es posible encadenar el uso de diferentes drivers e, incluso, que un driver solicite el uso de un tecero para llevar a cabo su propósito.

Así pues, el driverWMS utiliza el driverCURL si este se encuentra localizable para realizar peticiones al servicio WMS. En la actualidad, este driver implementa dos operaciones diferentes sobre un servicio WMS: (1) Negociación (2) Petición de una extensión de un layer.

A la hora de conectar con un servicio WMS, el driver ha de realizar una negociación. En primer lugar se hace una petición "GetCapabilities" adjuntando el número de versión más alto que este driver permite (Actualmente, versión 1.3.0). Seguidamente recibimos la respuesta en un archivo XML. La respuesta contiene todos los servicios permitidos, así como un número de versión. Si el servicio WMS soporta la versión que el usuario ha demandado coincidirá, de lo contrario tendrá la versión más alta que soporta el servicio. Si el driver soporta dicha versión se pasa a parsear los diferentes servicios que ofrece el servidor. De lo contrario, el driver volverá a pedir el uso de servicio con una versión que conozca pero de menor número que la ofrecida por el servidor. Esta negociación concluirá negativamente en el momento que el driver se quede sin versiones soportadas por las que preguntar. En la actualidad el driver WMS soporta las versiones 1.1.1 y 1.3.0 del estándar WMS.

Una vez concluida la negociación, el fichero de capacidades se parseará de acuerdo a la especificación de versión de WMS. Este parsing devolverá información sobre las diferentes capas que existen en el servidor, así como sus propiedades (formatos, sistemas de cordenadas, límites de acceso, etc).

Código 4.14: Consulta de los formatos soportados utilizando el mecanismo de propiedades.

```
1 std::vector<std::string> supportedFormats;
2 prop = driver->getProperty("SupportedFormats");
3 if(prop!=0){
4   prop->get(supportedFormats);
5   LOG.notice("Formats:",supportedFormats);
6 }
```

El usuario puede consultar esta información y fijar los parámetros de consulta WMS mediante el uso del mecanismo de properties que incluyen los drivers. En el código 4.14 se puede ver un ejemplo de consulta que devuelve la propiedad solicitada.

4.2.4. Driver GDAL

Finalmente el driver GDAL es un driver para procesar datos out-of-core. Este plugin no accede directamente a ningún tipo de servicio, sino que se alimenta a partir de datos que ya hemos obtenido previamente. En el código de ejemplo 4.15 se puede ver la definición del plugin.

Código 4.15: Definición del driver buffer-espacial que utiliza la librería GDAL.

```
1
2  class GDALMWDriver : public SpatialBufferDriver
3  {
4
5  public:
6      enum FormatType
7      {
8          LUMINANCE,
9          LUMINANCE_ALPHA,
10         ALPHA,
11         RGB,
12         RGBA
13     };
14
15     enum DataType
16     {
17         NONE,
18         UNSIGNED_BYTE,
19         UNSIGNED_SHORT,
20         SHORT,
21         UNSIGNED_INT,
```

```
22          INT,
23          FLOAT,
24          DOUBLE
25      };
26
27      GDALMWDriver();
28      virtual ~GDALMWDriver();
29
30      // from Driver
31      virtual bool open(const std::string &connectionString,
32                        Driver::OpenMode mode);
33
34      virtual bool close();
35
36      virtual bool isOpen() const;
37      virtual bool flush();
38      virtual std::string getFileExtension() const;
39
40      // from Buffer
41      virtual void computeSubBuffer(double minX, double minY, double maxX, double maxY);
42      virtual void computeBuffer();
43      virtual mwCore::Buffer getBuffer();
44      virtual void setWidth(size_t width);
45      virtual void setHeight(size_t height);
46      virtual size_t width();
47      virtual size_t height();
48
49      // from Spatial
50      virtual void setExtension(double minX, double minY, double maxX, double maxY, double minZ, double maxZ);
51
```

```
52      virtual bool isWGS84();
53
54      virtual bool isUTM();
55
56      virtual const std::string &getProjectionString() ↙
            ↳ const;
57
58      /** Gets the extension of the whole data if it ↙
            ↳ is spatial **/
59      virtual void getExtension(double &minX, double ↙
            ↳ &minY, double &maxX, double &maxY, double ↙
            ↳ &minZ, double &maxZ);
60
61      virtual void queryExtension(double minX, double ↙
            ↳ minY, double maxX, double maxY, double ↙
            ↳ minZ, double maxZ);
62
63      /** gets the size of the whole data **/
64      virtual size_t size();
65
66  protected:
67      bool _isOpen;
68
69      // gdal handler
70      GDALDataset* _hDataset;
71
72      // retrieved or auxiliar data retrieved from the ↙
            ↳ datasource
73      std::string _projectionString;
74      bool _ecwLoad;
75      // extension of the data source
76      double _minX;
77      double _minY;
78      double _maxX;
```

```
79      double _maxY;
80
81      // query data
82      double _queryminX;
83      double _queryminY;
84      double _querymaxX;
85      double _querymaxY;
86
87      // retrieved buffer
88      mwCore::Buffer _bufferData;
89      size_t _width;
90      size_t _height;
91
92  private:
93      // Disallow copy
94      GDALMWDriver(const GDALMWDriver &GDALMWDriver);
95      // Disallow assignment
96      GDALMWDriver &operator=(const GDALMWDriver ↙
            ↳ &GDALMWDriver);
97
98      void flipVertical(mwCore::Buffer &buffer, ↙
            ↳ FormatType pixelFormat,DataType type);
99
100     unsigned int computePixelSizeInBits(FormatType ↙
            ↳ pixelFormat,DataType type) const;
101     void flipImageVertical(unsigned char* top, ↙
            ↳ unsigned char* bottom, unsigned int ↙
            ↳ rowSize, unsigned int rowStep);
102     unsigned int computeRowWidthInBytes(int ↙
            ↳ width,FormatType pixelFormat,DataType ↙
            ↳ type,int packing);
103 };
```

Actualmente este plugin permite gestionar el recortado por coordenadas de una

fuente imagen de origen con un marco de coordenadas de referencia. La salida de datos de este plugin es un buffer de memoria que contiene los datos de color por cada pixel de la imagen. Esta salida tiene un formato de color y pixel expresados con valores de los enumeradores FormatType y DataType.

4.3. mwLauncher

Como se ha explicado anteriormente en el apartado 3.3 el framework mWorld emplea un ejecutable, mwLauncher, como runtime de las aplicaciones diseñadas bajo el. En la actualidad el lanzador mwLauncher funciona en la siguientes plataformas:

- Windows
- Linux
- iOS
- Android

En el momento de la escritura de este trabajo, las versiones para equipos de sobremesa: Windows, Linux e iOS no presentan diferencias significativas para su compilación, ejecución y empleo. Por ello trataremos las tres versiones en el siguiente punto. A diferencia de estas plataformas, Android presenta diferentes particularidades y necesidades, pero eso existen dos lanzadores diferentes para la plataforma: JNI Android Launcher y Native Android Launcher.

4.3.1. Desktop Launcher

El lanzador para sistemas operativos de sobremesa es una aplicación de consola que acepta la entrada de una serie de parámetros. Estos parámetros sirven para:

- Indicar la librería dinámica con la aplicación a cargar
- Indicar una serie de plugins para cargar
- Añadir paths de búsqueda de archivos y librerías
- Activar el modo verboso del logger

- Añadir parámetros para la inicialización de la aplicación

Cuando se inicia el lanzador, se parsean los parámetros que se han recibido de entrada mediante el LauncherParameterParser cuyo código se puede observar en la pieza 4.1. En primer lugar se inicializa el singleton Registry y se configura el nivel de verbosidad del logger. Seguidamente se recupera la lista de plugins y la aplicación a cargar. Para cada uno de estos elementos se realiza una búsqueda en los paths por defecto de la aplicación. En esta búsqueda se emplean tanto los paths por defecto para cada sistema operativo como aquellos indicados mediante línea de comandos. Seguidamente se carga cada una de las librerías, plugins y la aplicación. Se ejecutan las funciones de inicio y se llama a la función que inicia al bucle de la aplicación. A partir de aquí, el flujo de la aplicación transcurre como se explica en el punto 4.1.7.

En este punto de desarrollo del framework, el lanzador de sobremesa no incluye funcionalidades dependientes del sistema operativo con las que una aplicación pueda interacturar. Esto supone que si el desarrollador intenta recuperar el adaptador del sistema operativo, únicamente recibe un puntero nulo.

4.3.2. Android Launcher

La adaptación del lanzador de mWorld para el sistema operativo Android consta de dos modalidades bien diferenciadas. Por un lado, se ha desarrollado un lanzador derivado de una actividad estándar que emplea un puente JNI para inicializar la parte nativa de la aplicación. Por el otro lado, tenemos un lanzador basado en una actividad nativa desarrollado completamente en C++. En ambos casos, el lanzador se integra como parte de la aplicación para facilitar la distribución de la aplicación, aunque es completamente posible realizar un lanzador único que lance diferentes aplicaciones.

4.3.3. JNI Android Launcher

El lanzador JNI es un paquete de dependencia en el proyecto del usuario. Este contiene una librería java junto a una librería nativa. El desarrollador genera un paquete con su actividad Android y llama a la función de inicio que expone el lanzador JNI. En la función de inicio, se pueden emplear los parámetros normales que ya se usan, de forma idéntica, en el de sobremesa. Esta función llama a la librería nativa, realiza el parsing de los parámetros pasados por el usuario y comienza el proceso de inicialización.

En primer lugar, recopila información necesaria sobre el dispositivo como la ruta actual de las librerías, el directorio de la aplicación o la ruta para el almacenamiento de datos en la sdcard. Seguidamente se realiza una búsqueda de todas las librerías que el usuario ha pedido abrir mediante la línea de comandos y se procede a su apertura. Finalmente, una vez todos los módulos, plugins, etc se encuentran cargados, se inicia la aplicación.

El desarrollador, puede emplear la actividad como una interfaz gráfica, o puede, por ejemplo, utilizar una GLSurface para representar gráficos mediante OpenGL. Este elemento, por ejemplo está integrado en la librería de dependencia para su uso común en aplicaciones. El elemento GLSurface se puede emplear en solitario, o como un elemento más de una interfaz de la aplicación.

La comunicación de los diferentes eventos de la actividad con la librería aplicación mWorld se realizan por dos vías. Por un lado tenemos una serie de eventos del sistema operativo que pasan la información a una serie de callbacks que el usuario puede registrar en el OSAdapter. Por otro lado, el desarrollador puede emplear el sistema de eventos de mWorld para pasar los eventos generados por disparadores en la actividad.

4.3.4. Native Android Launcher

El lanzador nativo de Android es una librería que se emplea como dependencia del proyecto. En la actualidad el usuario está obligado a implementar una función de inicio donde se pasan los parámetros de carga de librerías, aplicaciones y plugins, sin embargo, está planeado sustituir este sistema por la lectura de un archivo de configuración que realizará el mismo cometido.

A diferencia de una aplicación que quiera utilizar la interfaz de usuario de Android, el usuario no necesita realizar ninguna pieza de código fuera de la librería aplicación mWorld y la comunicación con los plugins que emplee. Únicamente deberá registrar en cualquier momento las funciones que desee utilizar para renderizado o respuesta de eventos del sistema. Para evitar la aparición de errores ANR el comportamiento por defecto considera que si no se encuentra ningúna callback a la que llamar en el OSAdapter, el lanzador se come el evento e informa al sistema operativo que el este se ha procesado satisfactoriamente.

4.4. Pruebas de funcionamiento

Para validar el trabajo realizado se ha realizado una batería de tests utilizando los modelos de ejemplo de la librería OSG. Las pruebas consisten en analizar el rendimiento, en frames por segundo, de una aplicación visor generada con el framework mWorld. Los modelos poligonales a emplear son los siguientes: Cow, Cessna, CessnaFire, Dumptruck, Fountain, Lz y Morphing. Las pruebas se han realizado sobre las plataformas Windows, Linux y Android. Los resultados se compararán con las implementaciones de referencia que tiene la librería OSG de una aplicacion visor.

4.5. Aplicación de prueba de concepto

Para finalizar el trabajo, se ha desarrollado una aplicación de prueba de concepto. Esta aplicación realiza la carga y representación tridimensional de capas raster. Estas capas se obtienen, mediante una conexión de red, desde servidores WMS. Una vez bajada la información de los parches de terreno se procesan y se introducen en una cache. Finalmente, el visor realiza una representación tridimensional de los parches de terreno. Esta representación varía dependiendo de la posición de la cámara y su distancia con la superficie a visualizar.

Esta aplicación multiplataforma se ha desarrollado sobre el framework mWorld. En la figura 4.2 se puede ver la estructura de la aplicación. Por un lado tenemos una aplicación mWorld principal (mWorldSimpleApp). Esta aplicación tiene un visor OSG y se comunica con un plugin mWorld (mwTerrain). Este plugin crea una cache con unos tamaños y propiedades delimitadas por la aplicación principal en el momento del inicio de la aplicación. Una vez creada la cache, el plugin intenta conectarse contra un servicio WMS. Para esto se carga dinámicamente un driver de acceso a datos, en este caso el driver WMS. Este driver, a su vez, carga dinámicamente el driver cURL para realizar las peticiones contra el servidor remoto. Una vez se ha conseguido, con éxito, la apertura del servicio, el plugin genera una estructura basada en nodos de terreno con nivel de detalle compatible con OSG. Esta estructura se devuelve a la aplicación principal y se carga en el grafo de escena. Finalmente, dependiendo de la posición de la cámara, la estructura cargada en el grafo de escena realiza peticiones dinámicamente contra la cache, si ya se había bajado la extensión, o contra el driver WMS. Este, a su vez, realiza peticiones contra el servidor empleando el driver de cURL para, posteriormente, procesar los datos de imagen mediante el driver de GDAL. Así, el driver WMS devuelve, diréctamente, los datos RGB/RGBA de las extensiones raster solicitadas.

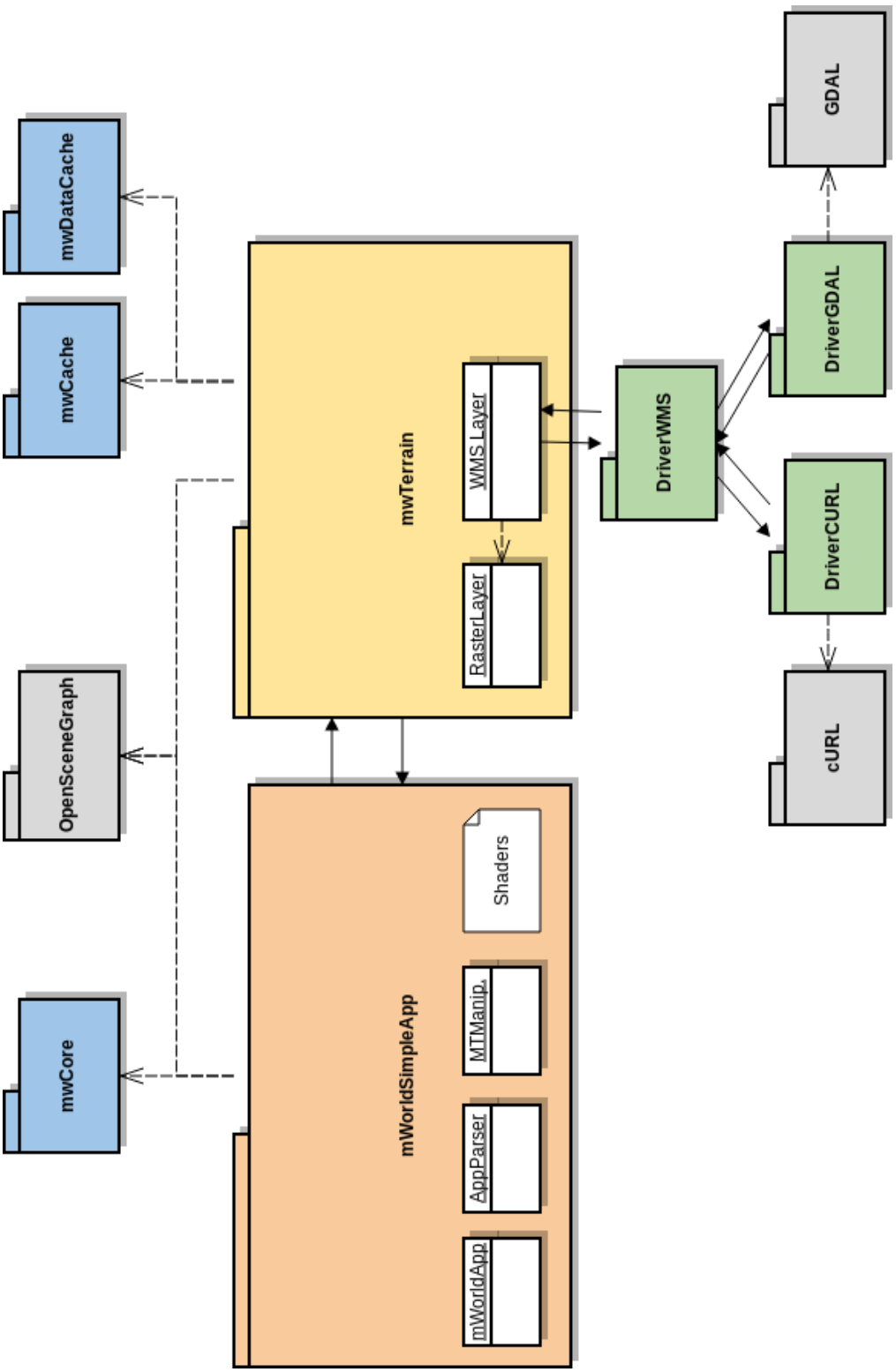

Figura 4.2: Estructura de la aplicación de concepto de mWorld. Esta aplicación se compone de una aplicación principal con un visor OSG, un plugin que gestiona el acceso y procesado de la capa WMS y una serie de drivers para acceder y procesar los datos.

5

Resultados

5.1. Características de los dispositivos de prueba

Durante este estudio se han empleado los siguientes dispositivos:

- Google Nexus 4.
- Asus Tf201
- Samsung Galaxy 3
- Portátil Acer Aspire 5742G
- Ordenador genérico, Windows

En la tabla 5.1 se encuentran las especificaciones de procesador, GPU, resolución y memoria de cada uno de los dispositivos. El sistema operativo elegido para los dispositivos Android es la versión 4.1.2. Para los ordenadores de sobremesa, se han empleado Windows 7 y una distribución Linux Ubuntu 12.04.

Modelo	Procesador	GPU	Memoria	Resolución
Acer Aspire 5742G	Intel Core i3	Ati M. Radeon HD5470	4Gbytes	1440x1080
O. Genérico	Intel Quad Core Q9990	Geforce 670	8Gbytes	1440x1080
Nexus 4	Q. Snapdragon S4 Procesador	Adreno 320	2Gbytes	1280x768
Asus Tf201	Nvidia Tegra3	Nvidia Tegra3	1Gbytes	1280x800
Samsung Galaxy S3	Exynos 4412	ARM Mali-400	2Gbytes	1280x720

Tabla 5.1: Características técnicas de los diferentes dispositivos de pruebas

5.2. Rendimiento de mWorld para el renderizado de modelos tridimensionales

Para obtener una referencia del impacto del uso del framework mWorld en contra de una aplicación pura, hemos realizado un aplicación que implementa un visor OSG. Esta aplicación nos permite comparar las diferencias de rendimiento en la representación de elementos tridimensionales. Esta prueba ha consistido en la visualización de la serie de modelos del set de ejemplos de la librería OSG.

Las tablas 5.2 y 5.3 nos permiten comparar el rendimiento, medido en frames por segundo, de nuestra aplicación en comparación con la aplicación visor que tiene la propia librería. De las cifras deducimos que existe una ligera sobrecarga en comparación a la implementación oficial, sin embargo estas diferencias se sitúan en el orden de un 1 % siendo esta pérdida de rendimiento prácticamente despreciable. Por esto concluimos que el framework mWorld no provocará pérdidas de rendimiento significativas en una aplicación, y que el desarrollador podrá obtener un rendimiento casi similar al que tendría generando una aplicación específica en una plataforma determinada utilizando las mismas librerías.

Las pruebas de diferencias de rendimiento, no las extendemos a los dispositivos móviles, ya que no existe ninguna aplicación visor de referencia. Además Android presenta una serie de complejidades que invalidarían determinados tipos de comparaciones directas. El rendimiento de una aplicación, sobretodo una que utilice intensivamente el chip gráfico, cambia en gran medida de como se haya construido, los drivers gráficos presentes en el dispositivo y el dispositivo en si. Por ello, en Android

Modelo	O. Genérico App mWorld	O. Genérico osgViewer
Cow	1755.8 fps	1755.4 fps
Cessna	1750.4 fps	1756.2 fps
Cessna Fire	1758.6 fps	1762.3 fps
Dumptruck	1753.2 fps	1758.1 fps
Fountain	1760.4 fps	1760.2 fps
Lz	1757.5 fps	1760.4 fps
Morphing	1756.3 fps	1758.8 fps

Tabla 5.2: Estadísticas en frames por segundo de la representación de los modelos del set de ejemplo OSG sobre una aplicación mWorld con un plugin visor OSG en el ordenador genérico.

Modelo	Acer Aspire App mWorld	Acer Aspire osgViewer
Cow	328.0 fps	330.2 fps
Cessna	330.2 fps	338.9 fps
Cessna Fire	338.7 fps	340.3 fps
Dumptruck	329.1 fps	335.7 fps
Fountain	340.3 fps	343.3 fps
Lz	338.9 fps	340.6 fps
Morphing	337.3 fps	343.2 fps

Tabla 5.3: Estadísticas en frames por segundo de la representación de los modelos del set de ejemplo OSG sobre una aplicación mWorld con un plugin visor OSG en el ordenador genérico.

la prueba de aceptación consistía en obtener tasas interactivas de renderizado.

Como ya se ha mencionado anteriormente, una aplicación Android con representación gráfica se puede crear bien usando una actividad Java/Dalvik (Que utilice, o no una librería nativa mediante JNI), o bien usando una actividad nativa y, por ende un contexto nativo. Como ya se explicó en el punto 3.5.1 si una aplicación utiliza el contexto gráfico compartido en Java/Dalvik se produce una pérdida de rendimiento en la tasa de dibujado. Esto es debido ya que, al compartirse el contexto, con el resto de elementos visibles de la interfaz Android, el sistema operativo debe realizar una serie de guardias a lo largo de su uso que terminan provocando una perdida notable de rendimiento.

La tabla 5.4 representa las tasas de renderizado en frames por segundo de la aplicación visor OSG, realizada sobre mWorld, usando el lanzador mWorldJNI. Por otro

Modelo	Nexus 4	Asus Tf201	Samsung Galaxy S3
Cow	30.3 fps	48.8 fps	44.8 fps
Cessna	27.6 fps	47.4 fps	42.5 fps
Cessna Fire	25.4 fps	45.2 fps	41.9 fps
Dumptruck	22.6 fps	46.0 fps	44.0 fps
Fountain	22.8 fps	45.4 fps	41.3 fps
Lz	21.2 fps	44.0 fps	43.1 fps
Morphing	27.9 fps	30.9 fps	30.4 fps

Tabla 5.4: Estadísticas en frames por segundo de la representación de los modelos del set de ejemplo OSG sobre Android empleando el runtime JNI del framework mWorld.

Modelo	Nexus 4	Asus Tf201	Samsung Galaxy S3
Cow	55.2 fps	59.9 fps	65.1 fps
Cessna	51.8 fps	58.7 fps	63.7 fps
Cessna Fire	50.9 fps	58.3 fps	62.9 fps
Dumptruck	50.2 fps	58.1 fps	64.8 fps
Fountain	52.8 fps	56.2 fps	63.5 fps
Lz	53.9 fps	55.4 fps	65.6 fps
Morphing	50.1 fps	42.8 fps	55.2 fps

Tabla 5.5: Estadísticas en frames por segundo de la representación de los modelos del set de ejemplo OSG sobre Android empleando el runtime nativo del framework mWorld.

lado, la tabla 5.5 representa las tasas obtenidas por la misma aplicación pero empleando el lanzador mWorldNative. Estos datos demuestran una diferencia significativa en el rendimiento de ambos modelos, con diferencias superiores al 10 %.

Esta pérdida de rendimiento puede ser aceptable dependiendo del software a realizar, en algunos casos será más beneficioso para el desarrollador tener una pérdida de rendimiento conocida a cambio de disponer libremente de la interfaz Android y viceversa.

5.3. Resultados de la aplicación de representación de capas WMS

La aplicación de ejemplo mWorld utiliza, a la vez, un driver de renderizado de terrenos que emplea la librería OSG, tres drivers de acceso a datos (Curl, WMS y GDAL),

además de un plugin OSG para representar los terrenos, así como un principio de librería GIS que se está desarrollando sobre mWorld. Esta aplicación ha sido probada con éxito en Windows, Linux, iOS y Android. La imagen 5.1 muestra el renderizado de una capa raster de terreno obtenida mediante el servicio WMS de la Generalitat Valenciana: http://terramapas.icv.gva.es/ La imagen en cuestión presenta una vista cenital de la ciudad de Valencia.

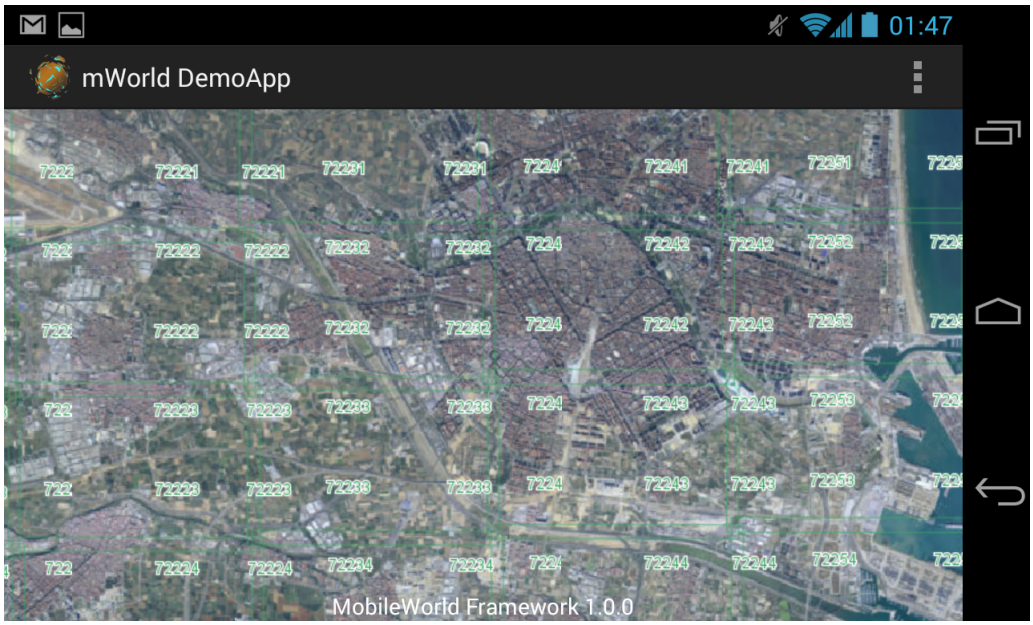

Figura 5.1: Imagen del programa mWorld utilizándola capa de ortofoto de 'http://terramapas.icv.gva.es/'. Vista lejana de Valencia.

La aplicación realiza la carga de los diferentes parches de terreno conforme a la navegación del usuario. Estos parches se introducen en una caché que es guardada en el disco duro o la memoria sd del dispositivo Android. Esta cache es totalmente configurable, tanto en tamaños de imagen, niveles de cache y tamaño total. Como se puede ver en la imagen 5.2 el usuario ha acercado la visión a una ciudad aledaña a Valencia, Torrente y la aplicación termina representando imágenes más cercanas de esa zona. En la imagen 5.3 se puede ver una imagen que ilustra el proceso de carga y sustitución de parches de terreno. Debido a que la carga se realiza a enviando peticiones de pequeñas regiones a través de la red (Una base de datos WMS tiende a ocupar decenas de Gbytes), no siempre se encuentra disponible la sección que está viendo el usuario. A medida que se cargan, los parches de terreno sustituyen a los de menor resolución dando la sensación de tener un terreno "parcheado".

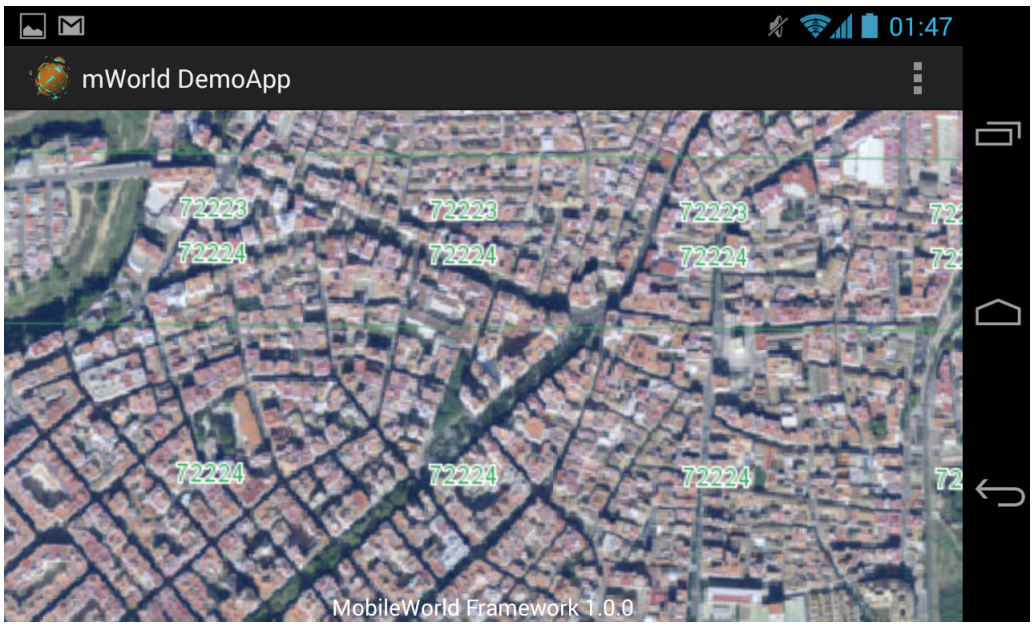

Figura 5.2: Imagen del programa mWorld utilizando la capa de ortofoto de 'http://terramapas.icv.gva.es/'. Vista cercana de Torrente.

A diferencia de trabajos previos, esta aplicación realiza la carga y el procesado de los datos WMS en el momento de su lectura. Esta aplicación no trabaja con datos preprocesados. Esto quiere decir que los servicios de mapas que utilizamos no están estratificados de una forma diseñada de antemano. Nuestra aplicación, realiza la carga de los datos de la capa, y va generando una estratificación o partición de forma automática. Esto introduce una sobrecarga computacional en comparación a las aplicaciones que utilizan datos preprocesados, pero confiere mayor versatilidad al programa. En la imagen 5.4 se puede ver que no controlamos el nivel máximo de la imagen, ya que no existe este tipo de información. Nuestro programa pide nuevos niveles hasta llegar al tope prefijado en la cache de la capa. Estos valores como he dicho antes, se pueden modificar cuando se crea la cache.

Finalmente, hay que destacar que la aplicación no está realizada en dos dimensiones. En la figura 5.5 se puede ver claramente el terreno abatido. Actualmente, los parches de terreno permiten incluir información sobre las alturas, es decir pasar de un parche plano a uno con forma tridimensional. Aunque estos datos están soportados, todavía no está implementada la mezcla de diferentes tipos de capas de datos.

En cuanto al rendimiento, hemos obtenido tasas de renderizado interactivas para los diferentes modelos de dispositivos Android. En la tabla 5.6 se encuentra el rendimiento obtenido. Por otro lado, la misma aplicación en los lanzadores de sobremesa

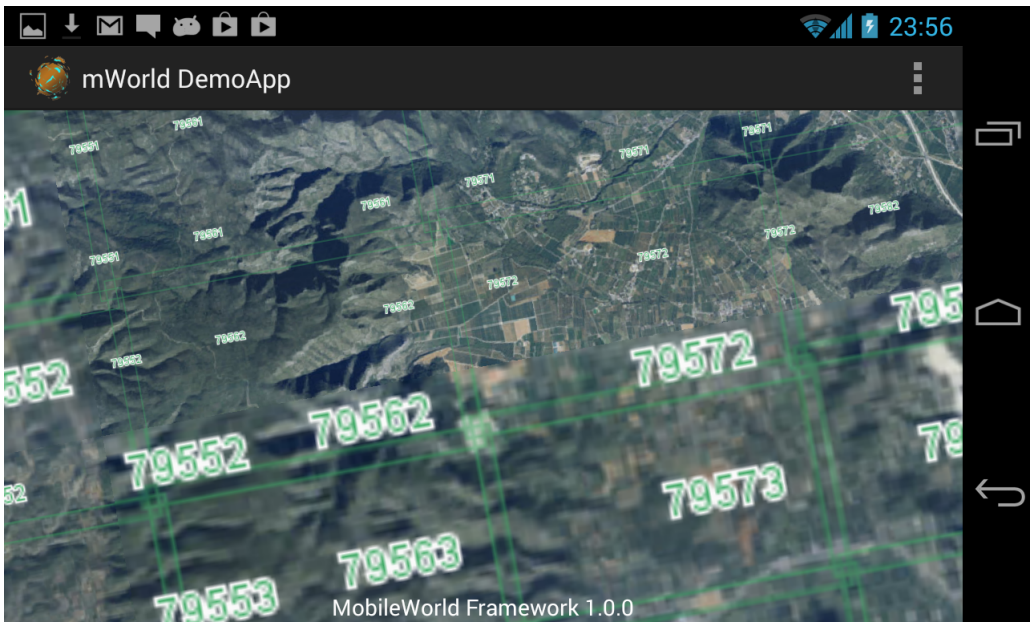

Figura 5.3: Imagen del programa mWorld utilizando la capa de ortofoto de 'http://terramapas.icv.gva.es/'. Como se puede observar en esta imagen, la carga de los trozos de terreno se realiza subdividiendo las placas de terreno. Esto ocasiona que visualmente se vean en algunos momentos diferentes niveles de detalle al lado uno de otro hasta que se ha completado la carga del nuevo nivel.

ha obtenido los rendimientos que se indican en la tabla 5.7

Launcher	Nexus 4	Asus Tf201	Samsung Galaxy S3
JNI	29.8 fps	35.6 fps	42.4 fps
Nativo	32.5 fps	43.8 fps	47.5 fps

Tabla 5.6: Estadísticas en frames por segundo de la aplicación ejemplo mWorld sobre Android.

O. Genérico	Acer Aspire
560.2 fps	95.8 fps

Tabla 5.7: Estadísticas en frames por segundo de la aplicación ejemplo mWorld en plataformas de sobremesa.

La aplicación de ejemplo de mWorld tiene un coste de memoria que, en el peor caso, se sitúa en el entorno de los 150Mbytes. Para mejorar la estabilidad de la aplicación

Figura 5.4: Imagen del programa mWorld utilizando la capa de ortofoto de 'http://terramapas.icv.gva.es/'. Se está utilizando el máximo zoom posible de la capa.

Figura 5.5: Imagen del programa mWorld utilizando la capa de ortofoto de 'http://terramapas.icv.gva.es/'. El programa ejemplo representa los tiles de terreno de forma tridimensional. En la imagen, se puede ver el abatimiento del terreno.

en los dispositivos que no se puedan permitir esta cantidad de memoria, se ha estudiado la inserción del mecanismo de regulación de calidad que se estudió en el trabajo previo. Al trabajar con elementos de un tamaño conocido, somos capaces de conocer en tiempo real una estimación bastante exacta de la memoria que está consumiendo nuestra aplicación. De cara a la representación, no podemos limitar directamente el número de elementos que se han de ver de forma exacta, así que empleamos un regulador. Este regulador comprueba el coste aproximado de representar la escena y, si se encuentra dentro de los límites de tolerancia no realiza ningún cambio. En el momento que una escena supera los límites, el regulador obliga al visor a que todos los elementos que se enseñen deban estar más cerca. Al usar esto con los elementos LOD que se utilizan para la representación del terreno, la aplicación consigue podar una serie de elementos del grafo de escena, liberando la memoria.

Si, por el contrario, la escena está por debajo del consumo mínimo, alargamos la distancia desde la cual un nivel de terreno es visible. Esto causa que se cargue una una mayor cantidad de elementos de terreno aumentando el consumo de memoria. En la figura 5.6 se puede ver la evolución del consumo de memoria a lo largo del tiempo de ejecución. En ese caso de ejemplo, los rangos se situaron entre 45 MBytes como mínimo de memoria y 65 Mbytes como máximo. Esto permite reducir el consumo de memoria de la aplicación a costa de una merma de la calidad visual.

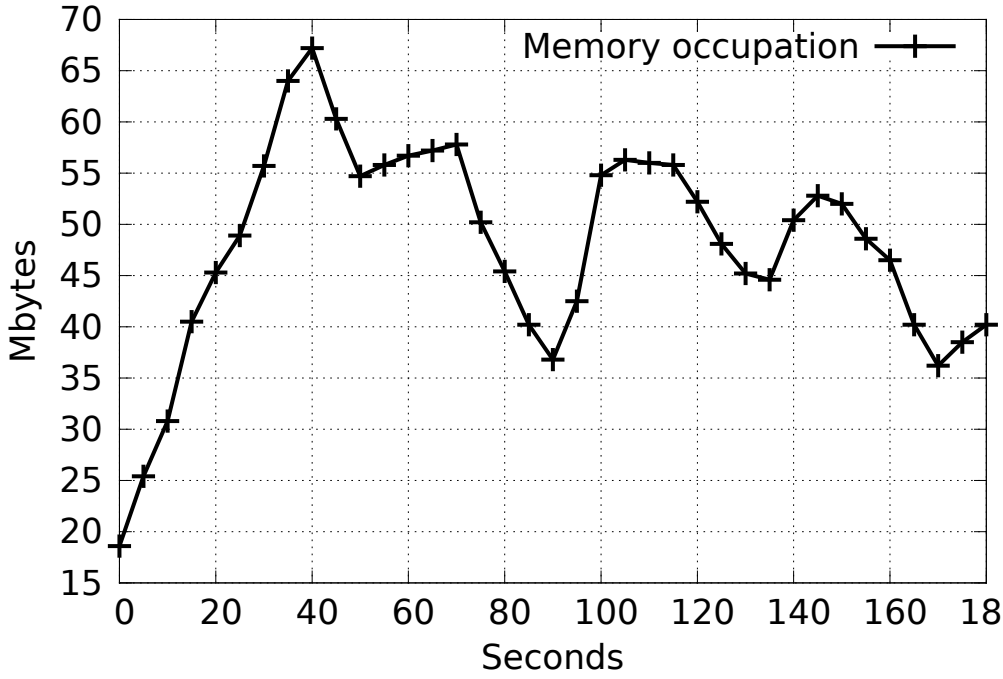

Figura 5.6: Gráfica del consumo de memoria de la aplicación ejemplo mWorld utilizando el sistema de ajuste de memoria. El rango de uso de memoria se ha establecido entre 45 y 65.

6

Conclusiones y trabajo futuro

Finalizamos el presente trabajo con un repaso de los puntos más importantes que se han tratado en el proyecto para, en último lugar establecer las líneas de trabajo futuras de las que partir.

El objetivo principal de este proyecto, ha sido la generación de un framework multiplataforma para el desarrollo de aplicaciones comerciales y científicas que, partiendo desde un único código de aplicación, el desarrollador pueda desplegarlo en diferentes plataformas. Como prueba de concepto y funcionamiento, se ha desarrollado una aplicación de prueba multiplataforma que realiza la representación tridimensional de capas GIS obtenidas de forma remota mediante un servicio WMS. Para el desarrollo de la parte gráfica de la aplicación, se ha empleado la librería OSG, que se usa como dependencia de la librería de terreno que incorpora mWorld

Este proyecto es continuación del trabajo final de carrera: Representación interactiva de escenas tridimensionales con OpenSceneGraph(OSG) en Android. En ese trabajo se presentaba la portabilización de la librería OpenSceneGraph al sistema Android, así como las posibilidades de uso del grafo de escena para su uso en desarrollos multiplataforma. En él se demostró como se podía adaptar, dinámicamente una aplicación a las limitaciones del dispositivo en tiempo real, así como la representación de terrenos utilizando bases de datos preprocesadas.

En ese trabajo se propusieron entre otras las siguientes líneas de trabajo futuro:

- Mejorar la integración de OpenSceneGraph con la plataforma.
- Desarrollo de actividades mediante la "NativeActivity".
- Desarrollo de la compilación y ejecución de OpenSceneGraph como librería dinámica.

A lo largo de este trabajo, se ha ido ampliando la compatibilización de la librería OSG con Android mediante ligeras modificaciones para mantener la compatibilidad a lo largo del tiempo. Estas aportaciones se han realizado por los cauces habituales en la comunidad OSG.

Además de estas modificaciones, se estudió la posibilidad de compilar, dinámicamente, la librería. Para esto, se ha generado una cadena compilación nueva basada en el uso de ToolChains de Cmake que permite la compilación dinámica de la librería, así como la carga de plugins dependientes que se encuentren en la misma ruta que el paquete aplicación.

Por otro lado, se ha conseguido una versión funcional del grafo de escena que emplea un visor basado en una aplicación nativa de Android. Gestiona la vida de la aplicación Android y se encarga de manejar la inicialización de un contexto gráfico nativo. Esta base de conocimientos ha sido empleada a lo largo del desarrollo y adaptación del framework para la plataforma Android.

En el momento de la redacción de este trabajo, se está preparando el envío de los cambios que permiten la compilación dinámica de OSG en Android, ya que es un cambio muy sensible al afectar los códigos de compilación de todas las plataformas.

El framework mWorld ha sido desarrollado con la intención de facilitar el desarrollo de aplicaciones multiplataforma. Se ha establecido un ciclo de vida de la aplicación mWorld que nos permite abarcar los diferentes modelos de ejecución en cada sistema operativo. Además, se ha establecido una metodología de plugins para añadir funcionalidades a la aplicación sin necesidad de dependencias directas contra los sistemas operativos. Cada plugin es un elemento cerrado independiente del resto de elementos de la aplicación, y que puede implementar cualquier funcionalidad que el desarrollador desee incluir en la aplicación. Así, es posible crear plugins que no tengan código dependiente del sistema operativo, o bien, es posible crear plugins que utilicen librerías específicas para una plataforma concreta. Al no existir una dependencia fija entre la aplicación mWorld y los plugins, el desarrollador tiene total libertad para intercambiar, módulos que respondan a los eventos de la misma forma.

Como se ha demostrado en el apartado de desarrollo, las aplicaciones mWorld ofrecen un rendimiento equiparable a las generadas nativamente para cada una de las plataformas. La aplicación de ejemplo, valida los diferentes requisitos que se habían planteado en el inicio del desarrollo. Esta aplicación, emplea diferentes librerías desarrolladas sobre el core mWorld, entre ellas la librería de acceso a datos, para acceder a un servidor remoto de datos. Este acceso se realiza mediante una serie de drivers que proporcionan la lectura de estos datos y su uso empleando una metodología out-of-core. En la actualidad, este framework se está empleando con aplicaciones en fase de producción en la empresa Mirage Technologies.

Este trabajo es un proyecto base. No es un una investigación o un desarrollo cerrado que termina en el presente trabajo. mWorld es un framework que sirve como piedra angular para futuros desarrollos e investigaciones. Es un framework con una base de conocimiento muy simplificada para facilitar la entrada a los usuarios y que es capaz de integrarse con la mayor parte de librerías existentes para generar aplicaciones complejas que se puedan lanzar contra diferentes sistemas operativos, incluidos móviles.

Las diferentes líneas de trabajo futuro pasan, en un primer lugar, por completar el soporte del framework para su uso en dispositivos móviles iOS que tiene una problemática y un ecosistema diferente a los sistemas que abarcamos en el presente.

Por otro lado, durante la realización de este proyecto, se ha generado una librería de representación de elementos GIS. En el momento de la escritura de este trabajo, la librería GIS de mWorld únicamente permite el renderizado (empleando OSG) de capas raster. Así pues, se podrían realizar sendos estudios para incluir la representación de capas vectoriales y datos lidar, enfocándolo a un uso multiplataforma donde se deba controlar el gasto de memoria. Otra posible línea de trabajo, sería la ampliación del soporte de las capas Raster. Actualmente, no está soportada la mezcla de capas de datos.

Otras líneas de investigación donde se puede aprovechar el framework mWorld es en campos como la realidad aumentada. Debido a la flexibilidad del diseño del framework, es muy facil generar aplicaciones que empleando librerías de terceros o de los sistemas operativos puedan tener acceso a cámaras y realizar procesado de vídeo junto a la representación de escenas tridimensionales.

7

Agradecimientos

Deseo agradecer a mi director de proyecto, Javier Lluch, por haber hecho posible este trabajo. También he de agradecer especialmente a Rafa Gaitán y Jordi Torres con quienes he seguido participando en el desarrollo de este proyecto.

Por otra parte quiero agradecer a los compañeros que he tenido durante los dos últimos años. Especialmente quiero agradecer la compañía, los consejos y las risas de Leo Salom, María Ten, Jesús Zarzoso y David Anes.

También quiero dar las gracias a mi familia que me han apoyado durante todo el trayecto de mi educación y en especial la figura de mi padre que ha tenido que soportar y corregir las repetidas lecturas de este trabajo.

Finalmente quiero dar las gracias a Isabel María Gracián, mi compañera de vida y mi correctora particular.

Bibliografía

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[BPT05] Kadi Bouatouch, Gérald Point, and Gwenola Thomas. A Client-Server Approach to Image-Based Rendering on Mobile Terminals. Research Report RR-5447, INRIA, 2005. http://www.inria.fr/rrrt/rr-5447.html.

[Cas07] Ignacio Castaño. *High Quality DXT Compression using CUDA*, 2007.

[Cat10] Ken Catterall. Migration to opengl es 2.0. In *GPU Pro*. AK Peters, 2010.

[CG02] Chun-Fa Chang and Shyh-Haur Ger. Enhancing 3d graphics on mobile devices by image-based rendering. In *Proceedings of the Third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing*, PCM '02, pages 1105–1111, London, UK, UK, 2002. Springer-Verlag.

[Cig11] Jorge Izquierdo Ciges. Representación interactiva de escenas tridimensionales con openscenegraph en android. September 2011.

[DD04] Florent Duguet and George Drettakis. Flexible point-based rendering on mobile devices. *IEEE Computer Graphics and Applications*, pages 57–63, July/August 2004.

[Don05] William Donnelly. Per-pixel displacement mapping with distance functions. In *GPU Gems 2*, pages 123–136. Addison-Wesley, 2005.

[Dub11] Patrick Dubroy. *Google IO 2011 conference: Memory Management for Android Apps*, 2011.

[For11] James Forshaw. Webgl - a new dimension for browser exploitation. Technical report, Context, 2011.

[FSJ11] James Forshaw, Paul Stone, and Michael Jordon. Webgl – more webgl security flaws. Technical report, Context, 2011.

[Gal11] Dan Galpin. *Google IO 2011 conference: Bringing C and C++ Games to Android*, 2011.

[GBO09] Mikael Gustavsson, Kristof Beets, and Erik Olsson. Optimizing your first opengl es application. In *ShaderX 7 Advanced Rendering Techniques*. Charles River Media, 2009.

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *In Rendering Techniques '98*, pages 181–192. Springer, 1998.

[goo10] *Google IO 2010- http://www.google.com/events/io/2010*, 2010.

[goo11a] *Google - http://developer.android.com/ndk*, 2011.

[goo11b] *Google - http://developer.android.com/sdk*, 2011.

[goo11c] *Google IO 2011- http://www.google.com/events/io/2011*, 2011.

[HL07] Zhiying He and Xiaohui Liang. A multiresolution object space point-based rendering approach for mobile devices. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, AFRIGRAPH '07, pages 7–13, New York, NY, USA, 2007. ACM.

[HMK02] D. Hekmatzada, Jan Meseth, and Reinhard Klein. Non-photorealistic rendering of complex 3d models on mobile devices. In *8th Annual Conference of the International Association for Mathematical Geology*, volume 2, pages 93–98. Alfred-Wegener-Stiftung, September 2002.

[htt] http://www.gdal.org/. *GDAL*.

[htt97] http://curl.haxx.se/. *cURL*, 1997.

[HW09] Bin Hu and Jingnong Weng. Component-based virtual globe visualization engine design. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1 –4, dec. 2009.

[Ima92] Imagination Technologies, http://www.imgtec.com/powervr/powervr-technology.asp. *PowerVR*, 1992.

[Khr92] Khronos Group, http://www.khronos.org/opengl/. *OpenGL ES - The Industry's Foundation for High Performance Graphics*, 1992.

[Khr04] Khronos Group, http://www.khronos.org/opengles/. *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*, 2004.

[Kry05] Yuri Kryachko. Using vertex texture displacement for realistic water rendering. pages 283–295, 2005.

[Ler04] Pierre Leroy. *Pocket GL, 3D library for Pocket PC*. http://pierrel5.free.fr/, 2004.

[LGCV05] Javier Lluch, Rafael Gaitán, Emilio Camahort, and Roberto Vivó. Interactive three-dimensional rendering on mobile computer devices. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 254–257, New York, NY, USA, 2005. ACM.

[LGEC06] Javier Lluch, Rafa Gaitán, Miguel Escrivá, and Emilio Camahort. Multiresolution 3d rendering on mobile devices. In *Proceedings of the 6th international conference on Computational Science - Volume Part II*, ICCS'06, pages 287–294, 2006.

[Mic92] Microsoft, http://msdn.microsoft.com/en-us/directx/. *DirectX*, 1992.

[Mik10] Morten S. Mikkelsen. Bump mapping unparametrized surfaces on the gpu. *J. Graphics, GPU, and Game Tools*, 15(1):49–61, 2010.

[Nvi] Nvidia: http://developer.nvidia.com/gpu-accelerated-texture-compression. *GPU Accelerated Texture Compression*.

[OGC00] OGC standards - http://www.opengeospatial.org/standards/wms. *WMS*, 2000.

[OSG12] OSG Community: http://www.openscenegraph.org. OpenSceneGraph. *Open Source high performance 3D graphics toolkit*, 2012.

[SkU06] László Szirmay-kalos and Tamás Umenhoffer. Displacement mapping on the gpu — state of the art, 2006.

[SP09] M. Sterk and M.A.C. Palacio. Virtual globe on the android - remote vs. local rendering. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 634 –639, april 2009.

[SZL02] Andrea Sanna, Claudio Zunino, and Fabrizio Lamberti. A distributed architecture for searching, retrieving and visualizing complex 3d models on personal digital assistants. *Internet Technology*, 3(4):235–244, 2002.