UNIVERSIDAD POLITÉCNICA DE VALENCIA

Proyecto Final de Carrera

Visualización interactiva de escenas tridimensionales con OpenSceneGraph sobre dispositivos Android

Autor: Jorge Izquierdo Ciges
Titulación: Ingeniería Informática
Dirigido por: Javier Lluch Crespo y Jordi Torres Fabra

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

13 de septiembre de 2011

Palabras Clave

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

Resumen

La representación interactiva de escenas tridimensionales en dispositivos móviles es un problema con un largo recorrido. Con el aumento de las características de los dispositivos embebidos, la creación de contenidos gráficamente atractivos se ha convertido en un factor de diferenciación en el mercado.

La aparición de la plataforma de código abierto Android ha supuesto una revolución en el mercado de los dispositivos móviles. Su estructura basada en el empleo de la máquina virtual Dalvik no había permitido emplear librerías de representación gráfica que no estuvieran basadas en Java.

Con la introducción de la programación nativa, este trabajo aborda la compatibilización de OpenSceneGraph, el estándar OpenGL para grafos de escena. Durante este trabajo se presentan los cambios realizados sobre la librería, las pruebas de funcionamiento realizadas y la utilización del grafo de escena para optimizar la representación de escenas que superan los límites de memoria de los dispositivos físicos. Seguidamente se analizarán los resultados y se presentarán las conclusiones de este trabajo.

Índice general

Índice de figuras

Índice de tablas

1

Introducción

Los dispositivos móviles están entrando en una época de grandes cambios, los avances en la capacidad de los procesadores embebidos y la aceptación social de los diferentes dispositivos que se abarca en esta familia (Smartphones, tabletas, gadgets, etc) los han convertido en un mercado objetivo para muchas compañías y desarrolladores.

La representación gráfica sobre estos dispositivos se ha visto condicionada por las diversas limitaciones de estas plataformas, entre las cuales hay que señalar la falta de capacidad de procesamiento, la falta de memoria y su gran latencia, la necesidad de un consumo bajo de energía, etc. Los diferentes estudios de representación tridimensional sobre estos dispositivos han buscado maneras de superar estos límites mediante el empleo de arquitecturas externas, simplificación de la representación, renderizado externo y otros.

Comercialmente, las optimizaciones más empleadas han sido las que se podían implementar en el propio dispositivo. El uso de impostores, la simplificación de elementos o la representación de interfaces bidimensionales han sido la tónica general de las aplicaciones comerciales.

Con el aumento de la potencia de cálculo en los dispositivos actuales, el sector de los dispositivos embebidos se ha convertido en un mercado competitivo que, mediante las evoluciones tecnológicas, está intentando cubrir todas las necesidades posibles

del consumidor. En un mundo donde el diseño y lo visual marcan las modas y las tendencias, la capacidad para crear representaciones tridimensionales e incluso programas que ofrezcan contenido "3D" son características que el usuario demanda a los desarrolladores.

A su vez, el usuario busca encontrar una respuesta a las acciones que realiza sobre su dispositivo. Busca establecer una conversación, una comunicación fluida donde las dos partes interaccionen. Si el usuario no percibe que la aplicación responde con suficiente rapidez, tendrá la sensación de que esta no funciona correctamente.

Con las necesidades de las aplicaciones actuales en los dispositivos móviles, el renderizado interactivo de escenas tridimensionales es un problema muy importante a abordar. Sin embargo, no tiene solución definitiva. Cuando se aumenta el nivel de detalle o el número de elementos en cualquier escena, se termina superando los límites de memoria y de procesamiento de cualquier dispositivo. Así pues, el problema no consiste tanto en cambiar la metodología para que funcione, sino en escalar las escenas a las características del dispositivo objetivo.

Escalar las necesidades de una escena tridimensional es un problema muy complicado de abordar cuando el programador se mueve en términos de instrucciones gráficas de bajo nivel. Para abordar de una forma más simple el problema existe la metodología de los grafos de escena. Un grafo de escena es una estructura basada en un grafos acíclico no dirigido que ordena los elementos gráficos de una escena de forma jerárquica espacial donde cada nodo únicamente posee un padre. Esta metodología permite al desarrollador abstraerse a un nivel superior donde puede aplicar optimizaciones para adecuar la representación de una escena a las características de cualquier dispositivo sin tener que lidiar con el código de bajo nivel donde las relaciones entre elementos no son tan sencillas de percibir.

El objetivo de este trabajo es portabilizar la librería OpenSceneGraph (OSG), el estandar OpenGL para grafos de escena, al sistema operativo Android y, con ello, estudiar la problemática para realizar representaciones interactivas de escenas tridimensionales en dispositivos actuales.

La librería OSG es una librería de código libre y multiplataforma escrita en C++ que proporciona las características de un grafo de escena. Se ha empleado en multitud de programas de todo tipo, desde aplicaciones científicas y de simulación hasta juegos. Actualmente es uno de los referentes libres más empleados por su versatilidad, su bajo grado de especialización y su capacidad para realizar aplicaciones multiplataforma. Actualmente permite realizar aplicaciones sobre Windows, Linux, Unix y los sistemas

Apple de sobremesa y embebidos.

Si bien la librería OSG ya ha sido portada a dispositivos embebidos (iOS), hasta ahora no había sido posible realizar una portabilización con éxito a Android debido a la estructura del sistema operativo. Android está orientado hacia el uso de la máquina virtual Java/Dalvik. Hubiera sido necesario reimplementar todo el código fuente de la librería en el lenguaje Java para poder emplear la librería con Android. Paulatinamente, Android ha ido incorporando el uso de componentes y aplicaciones nativas que no empleen la máquina virtual a semejanza de los dispositivos embebidos de Apple.

Aún con la aceptación de la programación nativa y a pesar de que Android emplea el Kernel de Linux, las diferencias existentes en las librerías que emplea, la estructura de los programas y la dificultad que conlleva programar y depurar programas de forma remota han convertido en un reto este tipo de portabilizaciones. Pocas librerías han conseguido llevar a cabo una portabilización completa y funcional a Android.

En este trabajo se abordará el problema mediante el estudio de todas las posibilidades actuales de la plataforma Android y de las diferentes dependencias de la librería OSG. Con ello se buscará realizar los cambios mínimos e imprescindibles para incorporar la nueva plataforma a la librería.

Esta memoria se compone, principalmente, de cuatro partes. En primer lugar, se expone una visión sobre el estado del arte de los diferentes elementos que se van a emplear y referenciar a lo largo del trabajo, así como las diferentes técnicas gráficas empleadas en la creación de las aplicaciones de prueba.

La segunda parte comprende el análisis de la problemática que supone compatibilizar el grafo de escena OSG con la plataforma Android y las soluciones planteadas durante la fase de análisis de este trabajo.

Seguidamente, se presenta el desarrollo realizado en este trabajo resaltando los elementos más importantes y críticos durante el proceso de compatibilización y la creación de los test de funcionamiento y optimización de escenas.

Posteriormente se presentarán los resultados obtenidos durante el desarrollo del proyecto para finalizar con las conclusiones y los posibles desarrollos futuros basados en este proyecto.

2

Antecedentes

Esta sección cubre el estado actual del arte sobre el que se apoya el trabajo. A lo largo de esta sección se hablará de la evolución de los diferentes elementos empleados en el trabajo para dar una imagen de su uso en el trabajo. Seguidamente hablaremos de la evolución histórica del renderizado en dispositivos embebidos para finalizar con un comentario de técnicas de representación de terreno tridimensional.

2.1. OpenGL

OpenGL [Khr92] es una API multiplataforma diseñada por la Kronos Architecture Research Board. Actualmente, se encargan de su mantenimiento un consorcio de empresas de CAD y, de forma destacada, las empresas Nvidia y Ati. Es un lenguaje de representación gráfica tridimensional creado con el objetivo de obtener un estándar multiplataforma libre. Está diseñada siguiendo una arquitectura cliente-servidor. Esta visión de comunicación es la base que inspira toda la especificación. Una de sus características más célebres fue la inclusión de un sistema de extensiones, que permitía la introducción de nuevas características y técnicas sin necesidad de modificar la API.

OpenGL ha sido durante muchos años la API "puntera" que se empleaba para todo tipo de aplicación gráfica. Su comunicación con las tarjetas a bajo nivel, le permitía acceder con menor latencia a las tarjetas sin pasar por el sistema operativo. Recorde-

mos que DirectX [Mic92] (Competencia directa en el mercado de Windows) obligaba a pasar las llamadas a través del kernel del sistema operativo con las consecuentes penalizaciones que eso conlleva.

Desde su nacimiento, OpenGl no fue concebido para un lenguaje específico, sin embargo, la implementación oficial está pensada para un lenguaje de tipo imperativo. Existen versiones para lenguajes no imperativos que se limitan a enmascarar la imperatividad propia de la representación de elementos gráficos de OpenGL. La implementación de referencia sobre la que están basada la mayor parte de documentación es para ANSI C99. Existen una serie de bindings para diferentes lenguajes, entre los cuales se encuentra Haskell. No existe una versión especifica para C++ que sea orientada a objetos, la única versión que podría considerarse así es una implementación creada para Java.

El proceso por el cual se genera una representación a partir de una geometría, 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 geométricos que van avanzando a través de una serie de fases en las cuales, el programador no puede intervenir físicamente. El programador, únicamente puede ajustar una serie de parámetros predefinidos, los cuales controlan el funcionamiento de los diferentes procesos de la tubería. Este diseño proviene de las estaciones de renderizado que se empleaban en los principios 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.

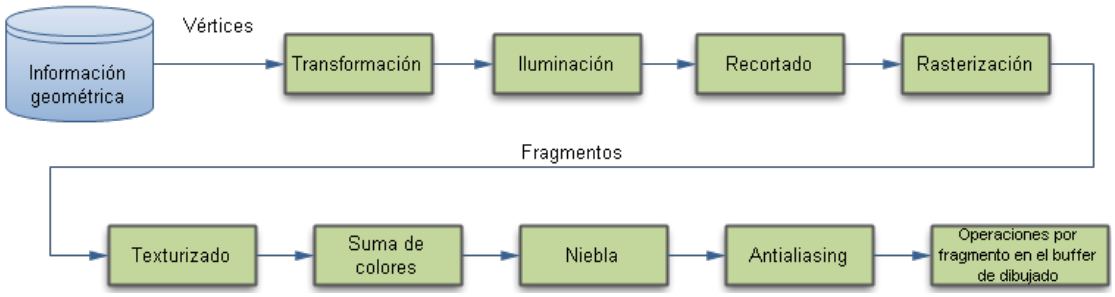

Figura 2.1: Diagrama de los procesos de la tubería de procesado fija en OpenGL.

Durante los años noventa, OpenGL adquirió una posición de ventaja competiti-

va en detrimento de alternativas como DirectX. Esta posición se debe, sobre todo, a la constante evolución que permitía el mecanismo de extensiones en OpenGL, este permitía ofrecer a los desarrolladores las características que todavía no habían sido integradas de forma fija en el lenguaje, de esta manera OpenGL se convierte en el sinónimo de API puntera.

En el último lustro, OpenGL entró en un periodo de letargo sin presentar novedades. La Kronos ARB se concentró en la creación de una nueva iteración de la API que mejorase la actual; una versión que eliminase los comandos y funciones replicadas. Finalmente, la API se concretó en dos nuevas versiones de OpenGL (3.0 y 4.0), ambas siguen sin romper con la parte fija de la librería, pero sientan las bases para su futura eliminación creando dos perfiles de uso: Núcleo y Compatibilidad.

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, habiendo desaparecido el uso intermedio que permitía la versión 2.1 de OpenGL. La tubería programable es una nueva manera 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 los procesos. Para ello, la API expone aquellos procesos no triviales para que el programador pueda programarlos a su voluntad, para ello el programador carga una serie de programas que se ejecutan desde la tarjeta gráfica, los shaders. Actualmente, existen tres tipos de shaders: los que afectan al procesado por cada vértice, los que afectan al procesado a nivel de primitiva geométrica y los que afectan a nivel de píxel visible. 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 en las limitaciones de la tubería fija.

El perfil de compatibilidad, mantiene todos los comandos y estados de las versiones anteriores. De la misma manera integra la tubería de procesado fija y permite el uso de todas las extensiones y elementos que se podían emplear en la versión 2.4. Esto significa que se pueden emplear Shaders utilizando una mezcla de tubería fija y tubería programable.

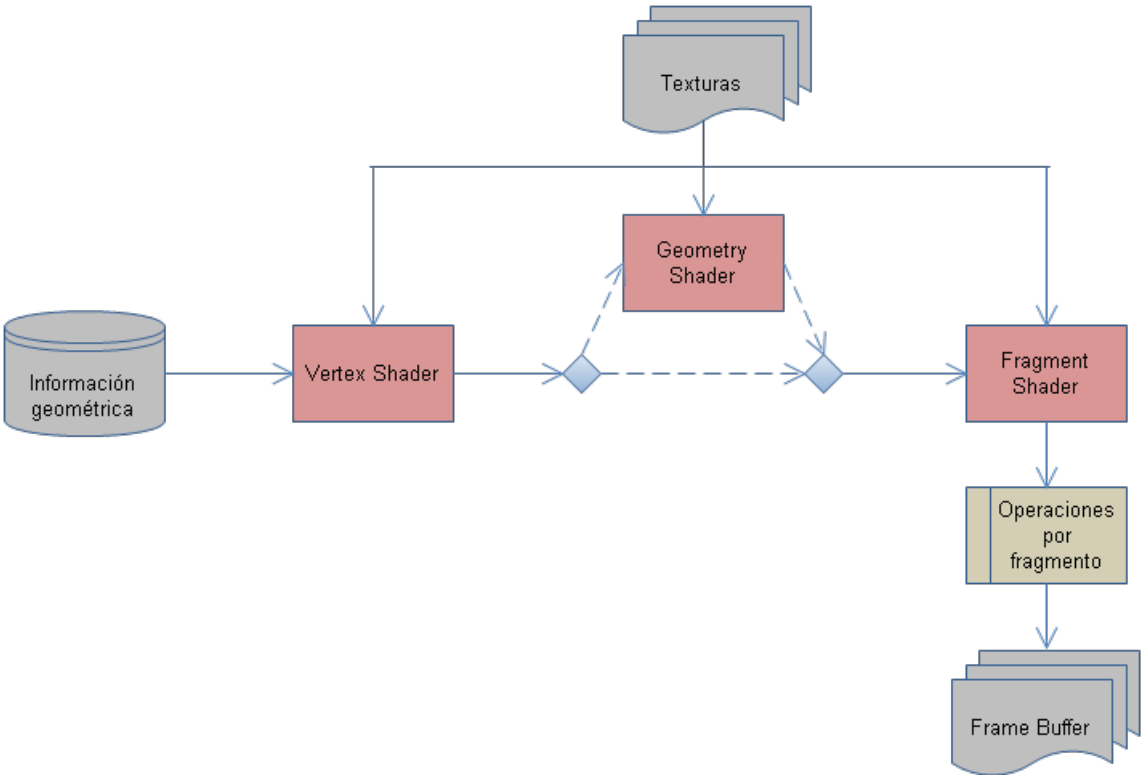

Figura 2.2: Diagrama de los procesos de la tubería de procesado programable.

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 estandarizar una API gráfica para dispositivos embebidos o de recursos limitados. El objetivo a la hora de crear esta API fue crear un subconjunto interoperable con la API de sobremesa, persiguiendo con ello la simplificación de la API. La evolución continua de OpenGL había supuesto la inclusión de muchos métodos que replicaban funciones con diferentes rendimientos por compatibilidad.

Por lo tanto en OpenGL ES encontramos con una versión simplificada que elimina las versiones más primitivas y lentas de envío de geometría a la tarjeta gráfica. Se prescinde de las instrucciones por vértices y de las listas de comandos conservando únicamente los Vertex Array, la alternativa con el mejor rendimiento. Se prescinde también de las instrucciones que permiten consultar las matrices, debido a su sobrecoste, y, en general, desaparecen los comandos que tienden a ser más lentos o que

realizan funciones de apoyo cuyo coste computacional no compense su uso.

En otras palabras, lo que encontramos es con una API compatible con su hermana de sobremesa siguiendo las referencias marcadas por OpenGL 1.4 y que emplea la tubería fija para realizar el tratamiento de los datos geométricos. Al igual que las versiones de sobremesa, las versiones embebidas tienen un sistema de extensiones que permiten incluir funcionalidades, que no estuvieran en un origen como las "Ambient Box", sin necesidad de cambiar la API. En la versión 1.0 de OpenGL ES existe una restricción sobre las texturas. Esta obliga a que tengan que ser cuadradas y potencias de dos. Sin embargo, esta restricción fue relajada en la versión 1.1. En dicha versión existen extensiones que permiten usar texturas que no tengan un tamaño de potencia de dos, si bien, por motivos de rendimiento se recomienda el uso de texturas cuadradas y con un tamaño que sea potencia de dos.

Finalmente hay que comentar una limitación que existía en OpenGL ES 1.0. Aunque se podía emplear aritmética de coma flotante para los gráficos, su uso ralentizaba mucho el renderizado de resultados. Para evitar esto, se recomendaba usar números decimales expresados en coma fija.

2.1.2. OpenGL ES 2.0

La versión embebida 2.0, está basada en la especificación de la versión 2.1 de OpenGL, sin embargo, en contra de la ES 1.X no busca ser totalmente compatible con una versión exacta. La especificación de OpenGL ES 2.0 se creó en el período de transición entre las versiones 2.1 y 3.0. Se eliminaron muchos elementos y comandos que duplicaban funcionalidades y se introdujo la obligatoriedad del uso de la tubería programable con los shaders de forma análoga a la versión 3.0.

En comparación con la versión 1.0, los mayores cambios fueron la introducción de tubería de procesado programable representada en: 2.1.2, la eliminación total del uso de la tubería de procesado fija y la retirada de limitaciones en los tamaños de las texturas; si bien las versiones de sobremesa siguen siendo compatibles en mayor o menor medida con la tubería fija, en OpenGL ES 2.0 la tubería desaparece completamente y no se puede compatibilizar con el driver de la 1.0 ya que son drivers independientes. La implementación de la tubería de procesado programable tiene algunas diferencias con la presentada en 3.0. En la versión de sobremesa existen los programas shader para procesar vértices, geometría y fragmentos. En esta versión, de forma parecida a la extensión que existía en 2.1, únicamente se permiten shaders para procesar vértices y fragmentos.

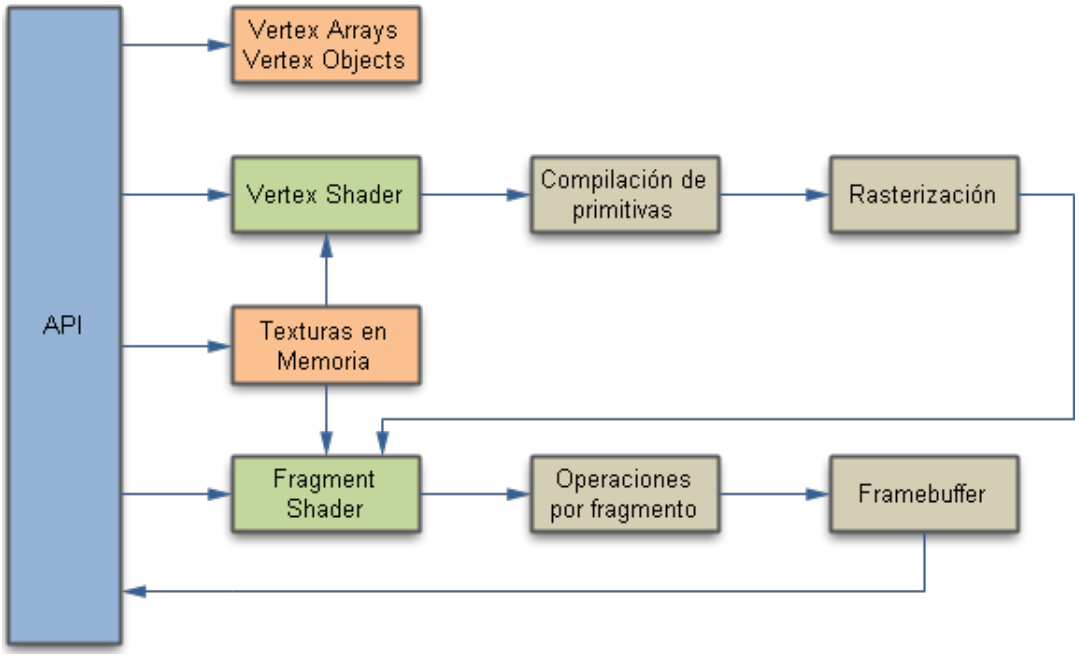

Figura 2.3: Diagrama de la tubería de proceso en OpenGL ES 2.0.

2.1.3. 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 está soportado a través de la librería webKit, lo cual permite su empleo en dispositivos embebidos 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. Apple, por su parte, no lo soporta de forma primaria en su navegador y emplea 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, este sistema operativo es apadrinado por la OpenHandsetAlliance(OHA). La OHA es un consorcio de diversas empresas que se han unido para ofrecer una respuesta libre con un estándar que reforme el panorama de los dispositivos embebidos. En la década de los noventa, el aumento de procesamiento invitó a crear dispositivos embebidos con mayores capacidades. A medida que se fueron desarrollando, las compañías, creaban dispositivos cuyo software era incompatible entre si. El mercado crecía sin ningún tipo de estándar o control, por lo que 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 ES fue uno de los estándares que tras un inicio tímido terminó perdurando en contra de otras API planteadas.

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. 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 al 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 alcanzado una cifra de 500.000 dispositivos activados diariamente y una tasa estimada de crecimiento de un 4.4 % cada semana..

Android ha evolucionado con las necesidades que ha ido presentando el mercado, pasando de ser un sistema para teléfonos móviles, a estar integrado en dispositivos multifunción (relojes, agendas) y en las tabletas, las cuales han recibido una especial atención por parte de Google que ha empleado toda una versión exclusiva para ellas, la versión Gingerbread (3.X) ha sido diseñada para su uso específico adaptando el sistema operativo a un manejo diferente al de los smartphones. Durante la conferencia de Google I/O de 2011 en Mayo se anunció la nueva versión del sistema operativo, Ice Cream Sandwich, los datos que se conocen hasta el momento señalan que se convertirá en una versión que unificará Android para su uso en tabletas y smartphones a la vez. El objetivo de esto es acabar con la fractura de mercado que originó Gingerbread entre tabletas y teléfonos.

La programación en Android, gira alrededor de la máquina virtual Java Dalvik. Esta máquina virtual es la que mueve todas las aplicaciones y procesos en un dispositivo con el afán de conseguir la mayor compatibilidad posible. El lenguaje de programación que se emplea es Java/Dalvik, sin embargo, Google permite desde la versión 1.5 del sistema la creación de programas Nativo mediante los lenguajes C/C++.

Las fuentes de documentación sobre la plataforma Android en las que se ha basado este documento 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.

El desarrollo de trabajo se ha basado principalmente en la programación nativa. En la conferencia [Gal11] se explica el uso, ventajas e inconvenientes del uso de la programación nativa. Para optimizar convenientemente los programas, se ha necesitado controlar el uso de la memoria, la búsqueda de pérdidas de memoria, así como las condiciones de liberación del recolector de basura de Android, se encuentran tratadas ampliamente en [Dub11].

2.2.1. Fundamentos de la plataforma

Android nace en sus orígenes como un proyecto de la Open Handset Alliance (OHA). Esta alianza buscaba crear un sistema operativo libre y abierto que diese un paso más allá de lo que existía en la época. 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 se debe de aplicar 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 de diferentes tipos: DEC Alpha, ARM, AVR32, Blackfin, ETRAX CRIS, FR-V, H8, IA64, M32R, m68k, MicroBlaze, MIPS, MN10300, PA-RISC, PowerPC, System/390, SuperH, SPARC, x86, x86 64 y Xtensa. El uso del Kernel Linux resulta en una gran ventaja, ya que no necesitan adaptar los mecanismos internos del sistema, únicamente deben adaptar 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 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. La máquina virtual Dalvik 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, la aplicación 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 se los concede 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, 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 el problema de la compatibilidad es por 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.

2.3. OpenSceneGraph

OSG [OSG] 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 empleada para crear aplicaciones de visualización, realidad aumentada, simuladores de vuelo o juegos.

El núcleo de OSG es el empleo de una metodología de grafos de escena, son grafos acíclicos 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 permiten visualizar, o no, ramas enteras dependiendo de nuestro criterio de visibilidad y, a partir de la selección generar un orden de visualización que sea óptimo para la renderización en la API. 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.

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 geográficas, 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 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 paque-

tes, 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 arquitectura en la cual se va a 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. La inexistencia de unos criterios comunes en las diferentes plataformas propició el uso de API gráficas propietarias sin compatibilidad. 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] o la reciente implementación de DirectX en el sistema móvil de Windows.

La estandarización de la API no supuso el fin de las restricciones de estos dispositivos, recordemos que estas surgen debido al hardware. La potencia de cálculo y la

capacidad de memoria de estos dispositivos no permitían representar escenas gráficamente complejas. Para salvar estas limitaciones aparecieron una serie de técnicas que permitían salvar las restricciones: renderizado basado en imágenes, en puntos, en geometría y simplificación en memoria externa basada en el punto de visión.

El renderizado basado en imágenes, consistía en emplear el uso de imágenes para reemplazar elementos geométricos. [CG02] propone el uso de una arquitectura cliente-servidor que renderizaba la escena en el servidor y envía la imagen al cliente. Por su parte, el renderizado basado en puntos consiste en representar la geometría dibujando una serie de puntos en la superficie del modelo [DD04] empleaba un mecanismo de representación de puntos jerárquico. El renderizado basado en la geometría es el mismo que se emplea en los sistemas de sobremesa, para adaptarlo al uso en dispositivos embebidos se emplean métodos como el que se plantea en [SZL02] que propone el uso de una arquitectura para buscar, recuperar y renderizar modelos complejos.

La técnica de simplificación en memoria externa basada en el punto de visión es un trabajo previo que se empleo en los trabajos [LGCV05] [Cam06]. Esta técnica emplea un modelo cliente-servidor para realizar el renderizado. La clave para salvar las limitaciones consistía en el empleo de un grafo de escena (OSG) para simplificar, mediante una serie de optimizaciones, la geometría dependiendo del punto actual de visión. El resultado era que el dispositivo cliente únicamente recibía la parte de la geometría que era necesaria, de esta manera se reducía el coste computacional de la representación y el espacio necesario para contenerlo en memoria.

Los trabajos previos han demostrado la utilidad del empleo de jerarquías y estructuras como los grafos de escena para la optimización de escenas como en el trabajo [ESC00]. El núcleo que moverá las escenas en este trabajo es OSG, que como grafo de escena, realiza optimizaciones gráficas como: culling, fustrum culling, LOD, y otras. Además permite incorporar técnicas de inspección del grafo de escena para operaciones complejas con la metodología de los "visitor". Su gran adaptabilidad permite adecuarlo con mínimos cambios para optimizar escenas complejas. Desde la versión 2.9 soporta el uso de OpenGL ES 1.X y 2.0. A diferencia de otros grafos de escena, OSG tiene una arquitectura modular basada en plugins, esto permite incluir únicamente los componentes que necesitamos, rebajando su ocupación en memoria, lo que representa un punto muy importante para el desarrollo de aplicaciones en estos dispositivos.

Los chipsets gráficos para dispositivos embebidos han evolucionado de forma paralela a sus homólogos de sobremesa. En la actualidad, la API OpenGL ES 1.0 está siendo sustituida por su evolución natural, OpenGL ES 2.0 [Khr04]. La novedad funda-

mental de esta nueva versión es el abandono del procesado de geometría con la tubería fija. Dada la potencia actual, se ha decidido seguir el camino de los chipsets de sobremesa incluyendo el procesado de geometría programable. Esto permite programar tal y como deseemos que se representen nuestros objetos geométricos, de esta manera se han generado una gran cantidad de técnicas aprovechando esta posibilidad. Para una referencia más extensa de las posibilidades que permite la nueva API, el articulo [Cat10] resume el funcionamiento y ofrece una serie de ejemplos de shaders y técnicas. También es recomendable la lectura del artículo [GBO09] que cubre las posibles optimizaciones que se pueden realizar con las API 1.0 y 2.0. Un punto importante de este artículo son las conclusiones sobre prácticas que se pueden convertir en cuellos de botella dependiendo de su uso.

Actualmente existe una serie de librerías que se están empleando para gestionar el renderizado en dispositivos embebidos. Algunas de estas librerías, como OSG ya habían sido utilizadas en los dispositivos iPhone; sin embargo, muchas de ellas todavía no tienen compatibilidad con Android debido a las dificultades para la programación nativa como ocurre con la librería Ogre que todavía no es compatible.

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

2.7.1. 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.

2.7.2. jPCT-AE

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

2.7.3. 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 nivel bajo 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 ser capaz de 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 cambiando 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 [USK06] 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.

3

Análisis del problema

A continuación se realizará un análisis de la problemática de visualizar interactivamente escenas tridimensionales en los dispositivos que emplean Android como sistema operativo. Se describen los problemas que supone migrar a la plataforma una librería, OpenSceneGraph(OSG), y las soluciones que se han planificado emplear en este proyecto. Finalmente se presenta la planificación del trabajo con un diagrama de Gantt señalando las tareas que se van a realizar y su planificación estimada.

3.1. La problemática de la representación interactiva en Android

La visualización interactiva de escenas es un problema sin solución óptima sea, o no, un dispositivo embebido. Una visualización interactiva obliga a responder al usuario en un tiempo suficientemente corto como para que no note que el mundo deja de responder a sus órdenes. Esto limita inicialmente la tasa de representación que se debe alcanzar como mínimo a diez frames por segundo. Aunque esta tasa parece lo suficientemente pequeña como para poder ajustarse a ella en cualquier situación, siempre existe un punto en el que una escena será incapaz de visualizarse en ese límite. En cualquier escena, si su detalle, número de elementos o requisito de memoria es amplificado terminará por ser imposible su representación con una tasa interactiva.

Así pues, aunque parezca que este problema es exclusivo de los dispositivos embebidos, es una limitación que existe en cualquier tipo de plataforma. La única diferencia es donde se encuentra el límite para cada sistema. Parafraseando la Ley de Parkinson, "Los desarrolladores emplearan todos los recursos disponibles". Así pues el problema consiste, sobretodo, en ser capaces de escalar las pretensiones y las necesidades de las escenas a nuestros límites.

Para conseguir escalar las escenas a los dispositivos, la implementación fija de una escena por código resulta ineficiente y costosa para entender y mantener por parte del programador. En los últimos años se ha ido introduciendo la metodología de los grafos de escena. Un grafo de escena sirve como una capa de abstracción para que el programador no tenga que escribir la escena en comandos gráficos de bajo nivel. Al poder abstraer la escena en términos de objetos, técnicas y otro elementos, el programador puede realizar optimizaciones desde un nivel superior. Esto permite a los programadores crear y gestionar formas simplificadas de una escena o implementar mecanismos de simplificación manuales o automáticos.

Por ello, el objetivo de este trabajo es "portabilizar" la librería OSG y emplearla, sin ninguna arquitectura externa o de apoyo, para representar escenas complejas con una tasa de dibujado interactiva. Esto se realizará empleando el sistema operativo Android. Si bien OSG ya es posible emplearlo en dispositivos embebidos con los iPhone, esto no se ha podido realizar hasta ahora en Android debido a las limitaciones de programación nativa que serán comentadas más adelante.

Este desarrollo plantea los siguientes problemas:

- La creación de una aplicación basada en OSG sobre Android.
- Las restricciones de memoria en los dispositivos embebidos.
- La diversidad de características en los dispositivos compatibles con Android.
- La falta de librerías necesarias para la representación y el manejo de elementos en OSG en Android.
- La gran cantidad de ruido en la entrada de datos táctil de algunos dispositivos.
- La integración del sistema de compilación de la versión Android en OSG.

En los siguientes apartados se tratarán las soluciones a estos problemas que han sido planificadas para este trabajo.

3.2. La creación de una aplicación basada en OSG sobre Android

Android es un sistema operativo que ha sido diseñado para conseguir la máxima compatibilidad posible entre todos los terminales. Para conseguirla, todo el sistema ha sido diseñado con una serie de criterios que lo alejan de sistemas operativos comunes. El aspecto más problemático que ha evitado este tipo de desarrollos se encuentra en la propia estructura del sistema operativo.

Conceptualmente, la estructura del sistema operativo Android es muy similar a la de los sistemas operativos de sobremesa. Existe una serie de niveles que se pueden mostrar como una serie de capas concéntricas donde se jerarquizan los accesos y la relevancia desde el interior hasta el exterior. La división de capas en Android consta de cinco niveles con dependencia creciente. En la figura: 3.2 se puede observar la representación de niveles.

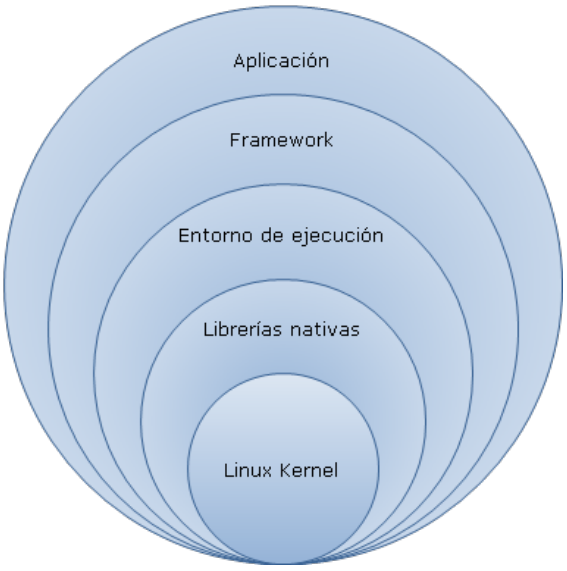

Figura 3.1: Diagrama de las capas del sistema operativo Android.

Al igual que en los sistemas operativos Windows o Linux, el núcleo del sistema está formado por el Kernel. Es el encargado de interaccionar con el hardware físico empleando unos módulos driver específicos para cada componente. Además, es el encargado de gestionar el uso de la memoria a bajo nivel, la gestión de procesos nativos, el soporte de hilos o la entrada salida. En el caso de Android, el Kernel no se ha hecho

específicamente como una nueva plataforma. Toma como Kernel el GNU/Linux.

Encima del kernel, se ejecutan las librerías nativas, estas se comunican directamente con el kernel del sistema para realizar sus peticiones. Entre las librerías que se encuentran implementadas en este nivel, podemos encontrar: OpenSSL, OpenAL, OpenGL, Zlib o Curl y otras.

A partir de las librerías se ha creado la capa del entorno de ejecución. Como se ha comentado anteriormente, el entorno de ejecución está compuesto por la máquina virtual Dalvik. Esta máquina virtual se ocupa de la gestión de la memoria a alto nivel y la ejecución propia de las aplicaciones. El entorno de ejecución a su vez provee al programador de un framework de clases con unas determinadas funcionalidades. El nivel de framework es donde se encuentran implementadas todas las clases de la API Android, como son las Activity, Intention, Services y otras se encuentran en este nivel.

En último lugar se encuentran las aplicaciones que ejecuta el usuario. Todas las aplicaciones que se encuentran en ejecución en el sistema, lo hacen en igualdad de condiciones sobre el entorno de ejecución. Al iniciar una aplicación, el entorno de ejecución crea una intención de ejecución de la actividad principal. En este proceso se genera una instancia de la máquina virtual que funciona de forma exclusiva para dicha actividad. En la instancia, la actividad puede demandar el uso de las diferentes clases del framework siempre y cuando la aplicación especifique que es compatible con su nivel API.

Así pues, **el desarrollo de una aplicación debe realizarse sobre el lenguaje java para ser ejecutado sobre la máquina virtual Dalvik**. En nuestro caso esto supone un gran problema por dos razones:

- La representación tridimensional a través de Java tiene una perdida sensible de rendimiento.
- Todas las librerías con capacidad de gestionar escenas tridimensionales están especializadas o no alcanzan el nivel de desarrollo de OSG.

Siguiendo lo expuesto aquí, **necesitaríamos reimplementar la librería OSG replicando sus funciones en Java para usarla en la máquina virtual Dalvik con la pérdida de rendimiento que ello conlleva**. Afortunadamente, existe otra solución, **desarrollar la aplicación con un enlace a una versión nativa de la librería OSG**.

Para la realización del trabajo, **es imprescindible que la librería OSG se integre, de forma nativa, en el sistema operativo Android**. Esto permitirá obtener un un mejor

rendimiento debido a la menor sobrecarga de comunicación con la API OpenGL ES. El desarrollo nativo de aplicaciones y la comunicación de librerías nativas con programas ejecutados en máquinas virtuales ha sido posible durante mucho tiempo en los computadores de sobremesa y algunos dispositivos embebidos. Esta es la razón por la cual, OSG lleva siendo compatible con los dispositivos basados en el sistema operativo iOS (iPhone, iPad). En el caso de Android, el desarrollo nativo de aplicaciones fue incluido a partir de la versión 1.5.

Oficialmente, Google desaconseja el desarrollo nativo por los posibles problemas de compatibilidad que pueden aparecer entre dispositivos. A diferencia de los dispositivos de Apple, Android no ha reducido el número de hardwares compatibles, sino que lo ha ido expandiendo introduciendo cambios, cuando ha sido necesario, para soportar nuevos tipos de pantallas, formatos de resolución, etc.

Aun así, la posibilidad de emplear el desarrollo nativo ha sido integrado para suplir las necesidades de los desarrolladores que requieren de características para las cuales, la máquina Dalvik supone un problema. El uso de gráficos intensivos, cálculos complejos que requieren de un rendimiento crítico o aplicaciones como juegos han propiciado que el soporte a la programación nativa y el número de funciones expuestas a los programadores sea ampliado en cada versión del "Kit de desarrollo nativo" (NDK). En la actualidad, el NDK tiene seis versiones principales y una serie de versiones menores que corrigen bugs que se han presentado. Oficialmente, los lenguajes permitidos por la programación nativa son C y C++, aunque es posible introducir compiladores nativos basados en Gcc para realizar la compilación en otros lenguajes.

A continuación se detalla el funcionamiento de las librerías y programas nativos en el sistema operativo y las limitaciones que impone el desarrollo nativo.

Como se ha comentado previamente en esta sección la máquina Dalvik, el entorno de ejecución, etc están implementados por encima de la capa nativa. La comunicación entre la parte nativa y el código Java/Dalvik se realiza a través de un puente JNI que permite acceder a las funciones del nivel nativo (figura 3.2.

La programación nativa en Android no puede ser calificada como puramente nativa si no se emplea la metodología de NativeActivity que será explicada más adelante. Si una aplicación quiere emplear código nativo y no emplea esa metodología, debe inicializarse a partir de una actividad (Activity) creada en Java/Dalvik, es decir, la aplicación se ha de ejecutar a nivel del entorno de ejecución Dalvik y desde su entorno, las actividades realizan llamadas JNI, que son las que realmente llaman al código implementado en la parte nativa de una aplicación.

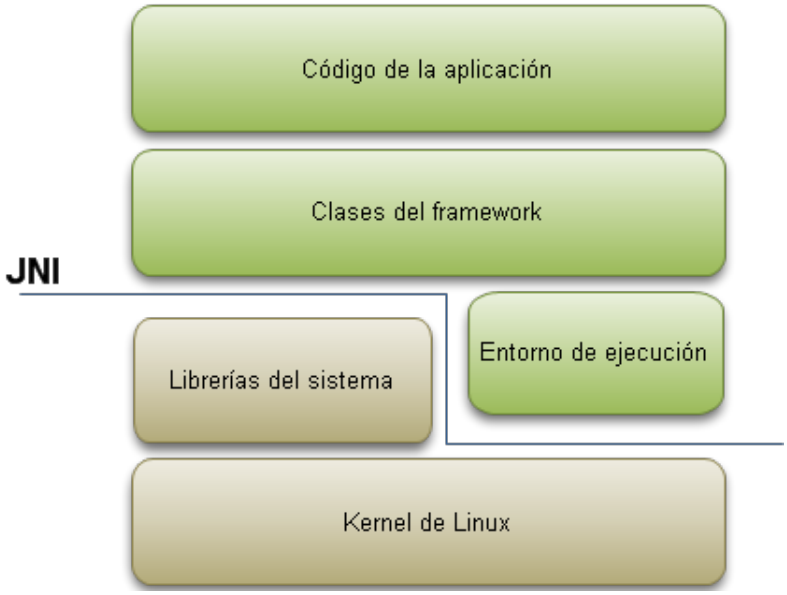

Figura 3.2: Diagrama de la estructura de niveles sin ejecutar código nativo.

Esta manera de proceder se implementó para obligar a los desarrolladores a emplear el máximo número de funciones de la API. Un ejemplo simple sería el acceso a archivos de recursos contenidos en un paquete. Para que una aplicación pueda funcionar siempre, el comportamiento correcto pasa por pedir a la API que devuelva un descriptor del recurso deseado. Sin embargo, habilitando la programación nativa existe la posibilidad de que los desarrolladores accedan a los recursos con métodos propios que rompan la compatibilidad de su aplicación cuando se cambie la gestión de los archivos de recursos. Por eso se ha intentado obligar a los desarrolladores a pasar por la API principal en Dalvik.

Una de las ventajas de la programación nativa es poder acceder a las librerías del sistema. Esto provoca una dificultad añadida para la compatibilidad. Es posible que una librería interna haya cambiado entre versiones. Cuando se crea una aplicación a alto nivel en Android, no hay ningún problema, la aplicación se ejecuta en el entorno de ejecución y el cambio en una biblioteca no se propaga hacia los niveles superiores, únicamente se ajusta el entorno para funcionar correctamente con el cambio de librería.

En el nivel nativo si ocasiona un grave error, ya que las librerías que se generan dependiendo de otra requieren exactamente la misma versión concreta. En la práctica existe la idea errónea de creer que dos compilaciones de la misma librería son idénticas. La verdad es que difícilmente son idénticas dependiendo del compilador y los

ajustes. El problema se genera cuando tienes un programa que ya ha sido compilado con una dependencia a una librería específica, a ese programa no le sirve cualquier librería, únicamente aquella para la que se compiló originalmente.

La solución que se propone en Android para estabilizar este problema consiste en declarar una serie de librerías mínimas que no presentan cambios de versión a versión, es decir, generaron una Application Binary Interface (ABI). Como se puede ver en el siguiente esquema: 3.2 la ABI se encuentra ahora como una capa de abstracción entre nuestras librerías nativas y las librerías reales del sistema operativo. Por comodidad las ABI tienen correlación con el nivel de API, si bien, al igual que ocurre en los niveles de la API, varios niveles API comparten una misma ABI.

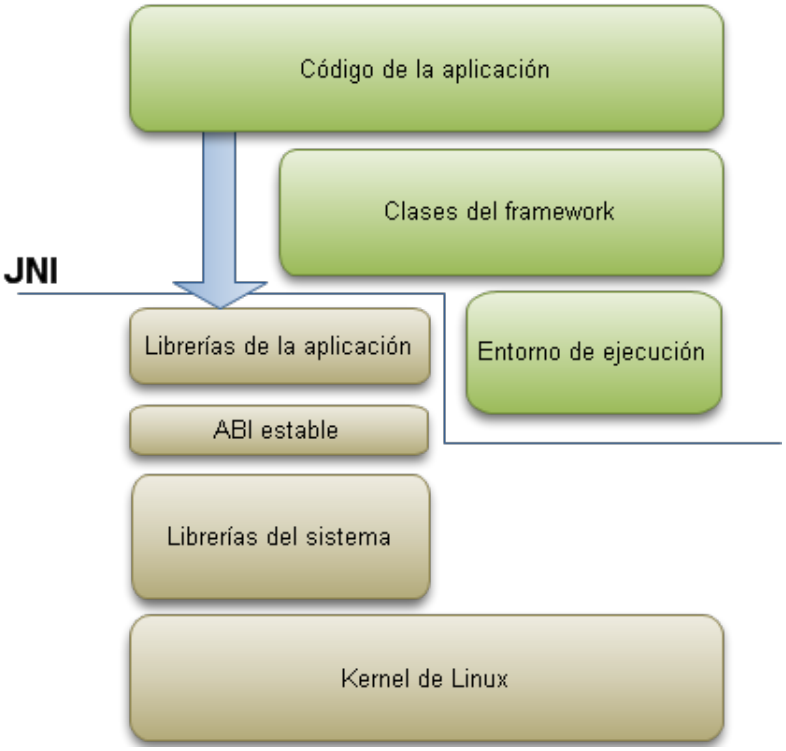

Figura 3.3: Diagrama de la estructura de niveles ejecutando código nativo.

En la versión 9 de la ABI el desarrollo nativo ha alcanzado un nuevo nivel con la integración de la Native Activity, esta permite, finalmente, desarrollar una aplicación totalmente nativa. La utilidad de esta metodología reside en evitar la comunicación no deseada con la máquina Dalvik y está aconsejada para las aplicaciones que prescindan del framework y la IGU de Android. Las aplicaciones para las que ha sido orientado

Versión ABI.	Nivel API
3	3
4	4
5	5 6 7
8	8
9	9 10 11 12 13

Tabla 3.1: Correspondencia entre las versiones API y las versiones ABI.

son, en mayor medida, juegos y aplicaciones que emplean sus propias IGU, esto es debido a que, aunque podamos implementar actividades nativas, si necesitamos acceder a algunas de las características del framework deberemos pasar a través de un puente JNI, si bien, en futuras versiones, es posible que se expongan parte de esas funcionalidades para su uso desde un programa nativo.

La lista de APIs que se pueden emplear desde la capa nativa de Android se encuentran en la siguiente tabla junto al número de API mínima necesaria.

La librería OSG se ha de compilar, obligatoriamente, para ejecutarse en el nivel nativo y se ha de llamar siguiendo las dos posibilidades que se ha comentado en este punto. La versión mínima requerida de la API es la versión cinco, ya que es la primera que incluye la compatibilidad con las dos versiones de la API OpenGL ES. Sin embargo, es recomendable fijar, como versión objetivo, la versión 2.1 o 2.2 ya que son versiones que incluyen muchos cambios en la API para la gestión de eventos, el uso de teclado virtual, etc. Crear una aplicación para una u otra versión supone optar a distribuirla a un 96.7 % de los dispositivos o al 81.5 % como se puede comprobar en la tabla: 3.3, siendo una decisión entre un mayor grado de desarrollo de Android o una mayor cuota de mercado posible.

La implementación de las librerías nativas de C en Android, **no es la implementación estándar de Linux**. En este sistema operativo se emplea una librería específica llamada Bionic que busca ser una implementación simplificada de las librerías de C, lo que significa que no incluye todos los métodos que existen en la implementación

APIs	Librería	Nivel mínimo
Entorno de ejecución de C	libc	3
Entorno de ejecución de C++	libstdc++	3
Librería Matemática	libm	3
Librería de enlazado dinámico	libdl	3
Loggin	liblog	3
Zlib	libz	3
OpenGL ES 1.1	libGLESv1_CM	4
OpenGL ES 2.0	libGLESv2	5
JNI Graphics	libjnigraphics	8
EGL	libEGL	4
OpenSL ES	libOpenSLES	9
Native Framework	libandroid	9

Tabla 3.2: Listado de APIs presente en la capa Nativa Android y la versión mínima requerida.

Plataforma	Nivel API	Distribución
Android 1.5	3	1.3 %
Android 1.6	4	2.0 %
Android 2.1	7	15.2 %
Android 2.2	8	55.9 %
Android 2.3 Android 2.3.2	9	0.6 %
Android 2.3.3 Android 2.3.4	10	23.7 %
Android 3.0	11	0.4 %
Android 3.1	12	0.7 %
Android 3.1	13	0.2 %

Tabla 3.3: Distribución de las cuota de mercado Android dependiendo de la versión del sistema operativo obtenido de: [goo11d] en Julio de 2011.

estándar. Esto se ve especialmente en la implementación de la librería de gestión de hilos, pThreads, donde falta una parte de los métodos. **Debido a que OSG depende, de ella para la ejecución de hilos, los métodos que no existen podrían generar un problema que solo será visible durante la realización del trabajo**

OSG es una librería que requiere dos características especiales del lenguaje C++: Excepciones y RTTI. Son funciones del lenguaje que no pertenecían a la especificación inicial y que han sido añadidas posteriormente en la librería STL. En las librerías nativas de Android se ofrece una versión no estándar y muy limitada de la STL que no tiene ninguna de esas características. Con el avance en las versiones del NDK, se han incluido también dos versiones más de la STL: una versión limitada de STLport y una versión estática de GNU STL.

La versión limitada de STLport contiende la mayor parte de características y estructuras de C++ a excepción de ambas características. En contra **la versión estática de la GNU STL presenta toda la implementación completa de STL** incluyendo ambas características. Esta librería fue incluida por primera vez en la versión cinco del NDK, siendo el mes de Diciembre de 2010 cuando realmente se pudo comenzar el desarrollo de este trabajo trabajo.

Finalmente, **el uso de la compilación nativa añade un nivel mayor de complejidad porque el programa final debe incluir la versión compilada para el procesador apropiado**. Esto deberá ser soportado en la compatibilización de OSG dando la posibilidad de optar entre las diferentes arquitecturas y optimizaciones que están soportadas actualmente.

3.3. Gestión de la memoria en Android

Uno de los puntos más habituales para convertirse en cuello de botella es en la memoria de los dispositivos embebidos. La falta de memoria es un problema conocido que se ha ido reduciendo en la presente generación de dispositivos. Los dispositivos de la actual generación llegan a tener 512Mbytes e incluso 1Gbyte reduciendo, parcialmente, la problemática.

Sin embargo, la falta de memoria, no es el único problema que podemos tener con la memoria. Las memorias empleadas en estos dispositivos suelen ofrecer una latencia muy alta que obligará a tener en cuenta si es preferible realizar cálculos matemáticos o realizar lecturas sobre posiciones de memoria que ya tengan los resultados. **Esto se estudiará durante el desarrollo del trabajo durante el programa de prueba de representación de terrenos generados en tiempo de renderizado**.

Pensando en la gestión de la memoria, hay algunos detalles que deben tenerse en cuenta en Android. La memoria en Android se divide en dos pilas con un comportamiento diferenciado. Por un lado está la pila de memoria de las aplicaciones en Dalvik

y por otro lado está la pila de memoria nativa. La plataforma además incluye el uso de un recolector de basura que actúa, únicamente, en la pila de memoria de Dalvik. La característica más curiosa es que el sistema no busca liberar la memoria de los programas que salen de ejecución ante la posibilidad de que puedan volver a entrar en ejecución. Así pues, la memoria de un dispositivo Android intenta estar ocupada en su totalidad y únicamente se libera espacio en la memoria cuando otra actividad con más prioridad la requiere.

Teniendo en cuenta estas condiciones, **será necesario emplear la metodología de los punteros de referencia que se emplea en la librería OSG**. Esta metodología evitará que tengamos pérdidas de memorias que, en un dispositivo donde la traza se ha de realizar externamente, resultan muy difíciles de encontrar.

Debido a la gran variabilidad de memoria según los dispositivos, **las optimizaciones para paliar el consumo de memoria deberán ser escalables**. De esta manera un mismo programa será capaz de ajustar la representación de una escena dependiendo de los recursos presentes en vez de optar por emplear la calidad de dibujado al peor caso posible.

3.4. Garantizar la compatibilidad entre dispositivos

La variabilidad del hardware compatible con la plataforma Android es muy elevada. Actualmente existen cerca de un centenar de dispositivos diferentes que emplean alguna versión del sistema operativo. El uso de una máquina virtual combinado con una serie de niveles de API intentan garantizar la compatibilidad de las aplicaciones. Aun cuando emplean una máquina virtual para asegurar esto, es imposible evolucionar las herramientas para dar más soporte y utilidades a los desarrolladores y a la vez, mantener una compatibilidad total con las versiones más viejas. Así pues en Android las roturas de compatibilidades no vienen dadas por la versión del sistema operativo. Estas vienen por los cambios que se realizan sobre la API.

Leyendo los documentos de las diferentes versiones del SO, vemos como los cambios en muchas versiones son únicamente para mejorar la compatibilidad, velocidad o estabilidad. En algunas de las versiones vemos que existe un cambio en la API que añade nuevas funcionalidades.

Cuando se desarrolla una aplicación, el programador puede emplear todo el conjunto de posibilidades de una API o solo un subconjunto, además puede utilizar una versión vieja de acceso a datos o una moderna. Esto es lo que determinará el requisito

mínimo de API que necesitará nuestra aplicación para ser ejecutado, de esta manera, pueden existir diferentes versiones del SO que son compatibles con el mismo nivel de API.

Versión del S.O.	Nivel API	Nombre
1.0	1	BASE
1.1	2	BASE_1__1
1.5	3	CUPCAKE
1.6	4	DONUT
2.0	5	ECLAIR
2.0.1	6	ECLAIR_0__1
2.1.X	7	ECLAIR_MR1
2.2.X	8	FROYO
2.3 2.3.1 2.3.2	9	GINGERBREAD
2.3.3 2.3.4	10	GINGERBREAD_MR1
3.0.X	11	HONEYCOMB
3.1.X	12	HONEYCOMB_MR1
3.2	13	HONEYCOMB_MR2

Tabla 3.4: Correspondencia de versiones Android con el nivel API y los nombre de versión

Por destacar algunas diferencias importantes, el nivel tres de la API es el mínimo para poder ejecutar una aplicación con partes creadas para ejecutarse de forma nativa independiente de la máquina Dalvik. La memoria máxima que se podía emplear en los dispositivos era de 256Mbytes hasta el nivel ocho cuando se eliminó esa restricción. La inclusión de un compilador JIT para aumento del rendimiento de las aplicaciones también aparece en el nivel ocho, así como las primeras tags de requisitos mínimos. Recientemente los niveles a partir del nueve incluyen el soporte para actividades nativas y optimizaciones dependiente de nuevos tipos de pantalla.

Junto al sistema de versión, se han implementado una serie de marcadores que pueden añadirse, como condiciones, al archivo de manifiesto de una aplicación, lo que permite incluir condiciones mínimas para que una aplicación pueda ser instalada. Algunas de las restricciones que podemos emplear discriminan los dispositivos por

modelos concretos, presencia de librerías, compatibilidad con extensiones de OpenGL o la posibilidad de emplear texturas comprimidas de un formato determinado.

```
<supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
<supports-gl-texture android:name="GL_OES_compressed_paletted_texture" />
```

Figura 3.4: Muestra de código con restricciones en el archivo de manifiesto.

En la figura: 3.4 se puede ver un ejemplo de restricción sobre un archivo de manifiesto que discrimina el dispositivo exigiendo que sea capaz de emplear la compresión ETC1 y la extensión de texturas paletizadas.

Siguiendo esta idea de especialización, se pueden emplear los cambios introducidos en la API de Honeycomb (3.X) que introduce nuevas posibilidades para distribuir las aplicaciones de forma especializada y concreta para cada dispositivo. Esta opción se empleará en el trabajo ya que permite realizar versiones específicas adaptadas a emplear una mayor o menor memoria dependiendo de la memoria disponible en el dispositivo.

Durante el trabajo se deberán especificar claramente las restricciones hardware para evitar que el programa final sea instalado en un dispositivo incompatible. Será conveniente definir la API gráfica usada, las extensiones requeridas y las compresiones de texturas que deben ser soportadas en el dispositivo.

3.5. La creación de un paquete Third-Party

La librería OSG está especializada únicamente en la gestión y representación de elementos gráficos, es decir, por si sola es incapaz de abrir muchos tipos de archivo. Para gestionar la apertura de archivos, OSG tiende a depender en una serie de librerías que ya están especializadas en la gestión de tipos de archivos concretos. Sin estas librerías, OSG únicamente soporta algunos formatos de archivos cuya decodificación ha sido implementada en el plugin sin depender de ninguna librería.

Si bien, OSG se puede emplear sin estos elementos, una de las grandes ventajas de la librería es, precisamente, que se encargaba de gestionar la apertura de los archivos y ponerlos a disposición del programador de forma transparente y no invasiva. **Sin esta**

característica, el desarrollador se queda limitado a unos pocos formatos compatibles que no son los habituales en una aplicación.

Por ello, es necesario realizar un paquete que añada el mayor número posible de estas dependencias para que el desarrollo de aplicaciones con OSG en Android sea atractivo de cara a los desarrolladores. Las principales librerías a tener en cuenta para ser incluidas son:

- Curl - Librería para el envío y recepción de datos por red.
- Freetype - Librería para la representación de fuentes de texto.
- Gdal - Librería para la gestión de bases GIS.
- giflib - Librería para el uso de imágenes gif.
- libjpeg - Librería para el uso de imágenes jpeg.
- libpng - Librería para el uso de imágenes png.
- libtiff - Librería para el uso de imágenes tiff.
- zlib - Librería para el uso de archivos comprimidos.

Para realizar este paquete, se generarán una serie de scripts que gestionen la compilación. Muchas de estas librerías se encuentran implementadas en el propio sistema operativo Android, pero **no se encuentran expuestas para su uso por el programador**. No forman parte de la ABI nativa y por lo tanto las versiones pueden cambiar con el tiempo, por ello lo correcto es incluir tu propia compilación de la librería junto al programa que la utilice para evitar incompatibilidades en el enlazado.

3.6. Filtrado de eventos de entrada táctil

En Android, la implementación de la gestión de la entrada táctil se deja a conveniencia de la empresa que fabrica el dispositivo. Así pues, la entrada que recibe el programador por parte del usuario, aunque cumple el estándar de eventos de Android, puede contener ruido que distorsione el gesto que recibe el programa.

El ruido en la entrada puede estar formado por diferencias de muestreo que hacen que la aplicación reciba un movimiento sin que el dedo se haya movido. Otro ruido

muy habitual ocurre cuando se detecta incorrectamente el número de punto de presión provocando que, para el programador, los puntos de presión se hayan invertido.

Para asegurar una experiencia de usuario correcta, es necesario filtrar estos ruidos a nivel de aplicación, ya que aparecerán en algunos dispositivos y, en principio, no lo podemos detectar sin comprobarlo físicamente. **Para resolver este problema, aplicaremos una técnica de filtrado a partir de una serie de muestreos previos** siguiendo el trabajo [Biz10]. Se utilizará una lista de muestras que se irán actualizando y que añadirán un pequeño retraso en la respuesta a cambio de obtener un funcionamiento correcto de cara al usuario.

3.7. Integración del sistema de compilado NDK con la librería OSG

El NDK emplea sus propios scripts de compilación, integrarlos directamente sobre OSG supondría obligar a que existieran dos cadenas de compilación, lo cual duplicaría el trabajo de mantener los archivos cuando se realizan modificaciones. Para soportar otras plataformas OSG emplea CMake como generador de scripts para compilar en cada plataforma cuando el usuario quiere compilar la librería. **CMake actualmente no soporta la generación de scripts para el NDK de Android**, pero dada su naturaleza de lenguaje de scripting, se puede programar sobre los scripts de CMake.

Para integrar la compilación Android en OSG, se empleará la opción de compilación cruzada que está presente en CMake, para ello se generará un fichero con los datos de la toolchain del compilador, fuentes y librerías que se encuentran en el NDK. Desde la versión número 5, es posible generar un directorio independiente con la estructura correcta para ser usado como toolchain para una compilación cruzada.

3.8. Planificación del proyecto

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

	Nombre	Duración	Inicio	Fin
1	Repaso del estado del arte	10d	03/01/2011	14/01/2011
2	Generación de ficheros script NDK	2d	17/01/2011	18/01/2011
3	Pruebas de compilación	1d	19/01/2011	19/01/2011
4	Compilación cruzada CMake	1d	20/01/2011	20/01/2011
5	Programación para generar scripts NDK con CMake	6d	21/01/2011	28/01/2011
6	Pruebas modelos 3D en OpenGL ES 1.0	5d	31/01/2011	04/02/2011
7	Pruebas modelos 3D en OpenGL ES 2.0	5d	07/02/2011	11/02/2011
8	Compatibilización librería jpeg	1d	14/02/2011	14/02/2011
9	Compatibilización librería png	1d	15/02/2011	15/02/2011
10	Compatibilización librería libtiff	1d	16/02/2011	16/02/2011
11	Compatibilización librería Curl	1d	17/02/2011	17/02/2011
12	Compatibilización librería Gdal	1d	18/02/2011	18/02/2011
13	Compatibilización librería libzip	1d	21/02/2011	21/02/2011
14	Compatibilización librería Freetype	1d	22/02/2011	22/02/2011
15	Creación paquete Third Party	1d	23/02/2011	23/02/2011
16	Pruebas Representación terreno pregenerado	15d	24/02/2011	16/03/2011
17	Preparación y envío de los cambios a la comunidad OSG	11d	17/03/2011	31/03/2011
18	Creación del prototipo de representación de terrenos pregenerados	5d	01/04/2011	07/04/2011
19	Optimización del prototipo de representación de terrenos pregenerad	16d	08/04/2011	29/04/2011
20	Creción del prototipo de generación de geometría en tiempo de dibuj	10d	02/05/2011	13/05/2011
21	Documentación Wiki de OSG	5d	16/05/2011	20/05/2011
22	Ejemplos para OSG	30d	23/05/2011	01/07/2011
23	Redacción Artículo	30d	04/07/2011	12/08/2011
24	Redacción de la memoria final	20d	15/08/2011	09/09/2011

Figura 3.5: Diagrama de Gantt de la planificación del proyecto página: 1/6

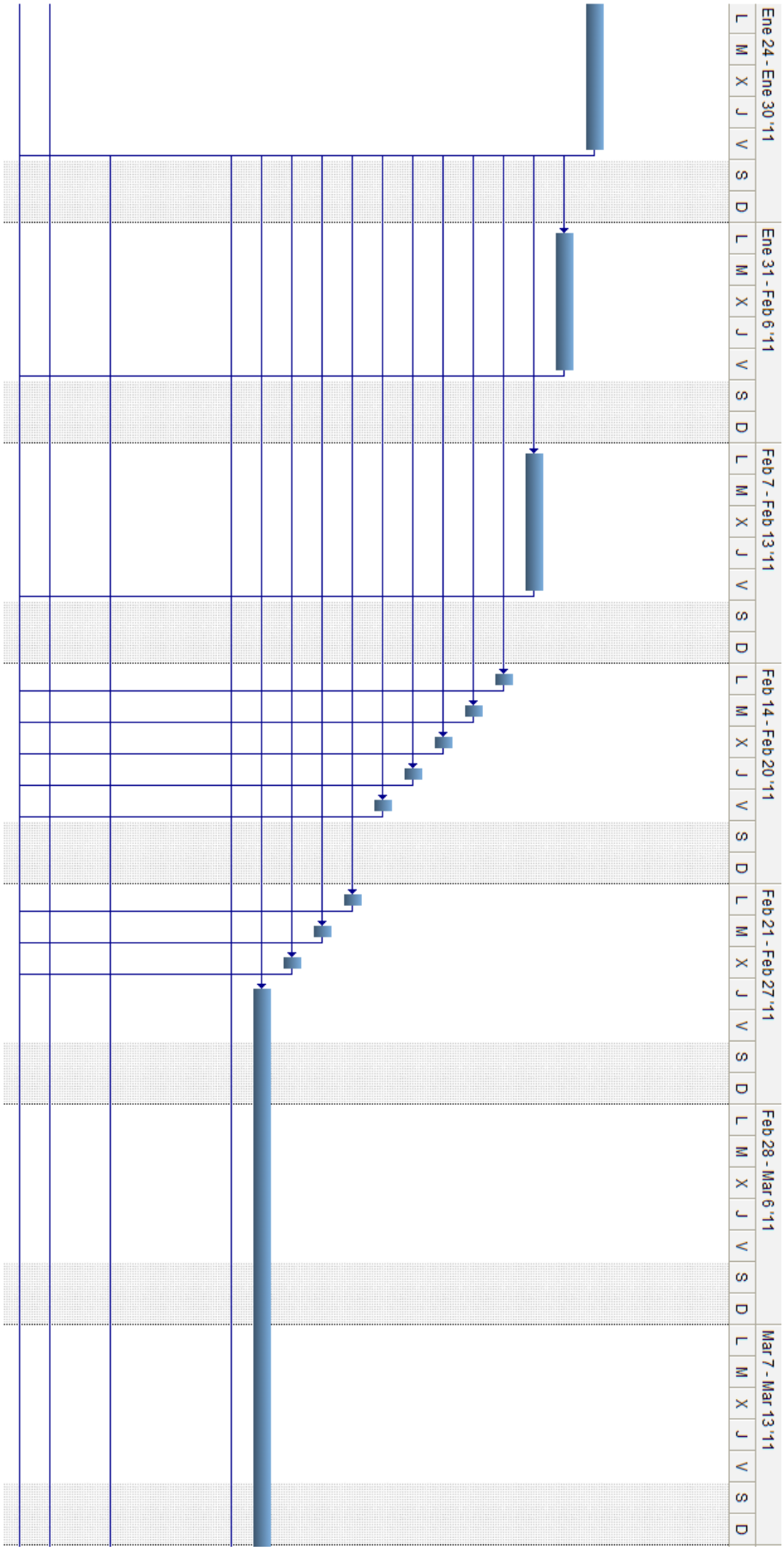

Figura 3.6: Diagrama de Gantt de la planificación del proyecto página: 2/6

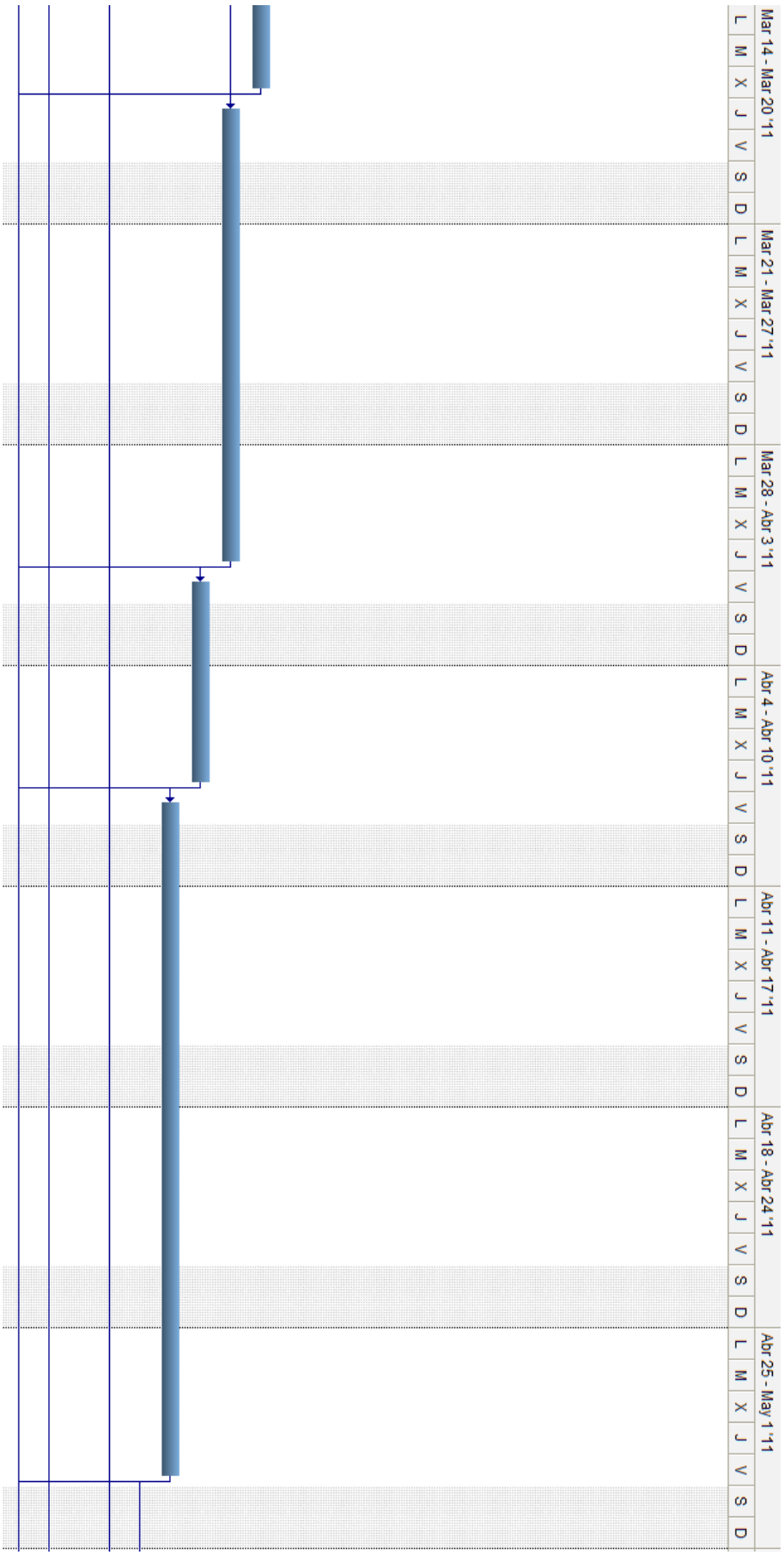

Figura 3.7: Diagrama de Gantt de la planificación del proyecto página: 3/6

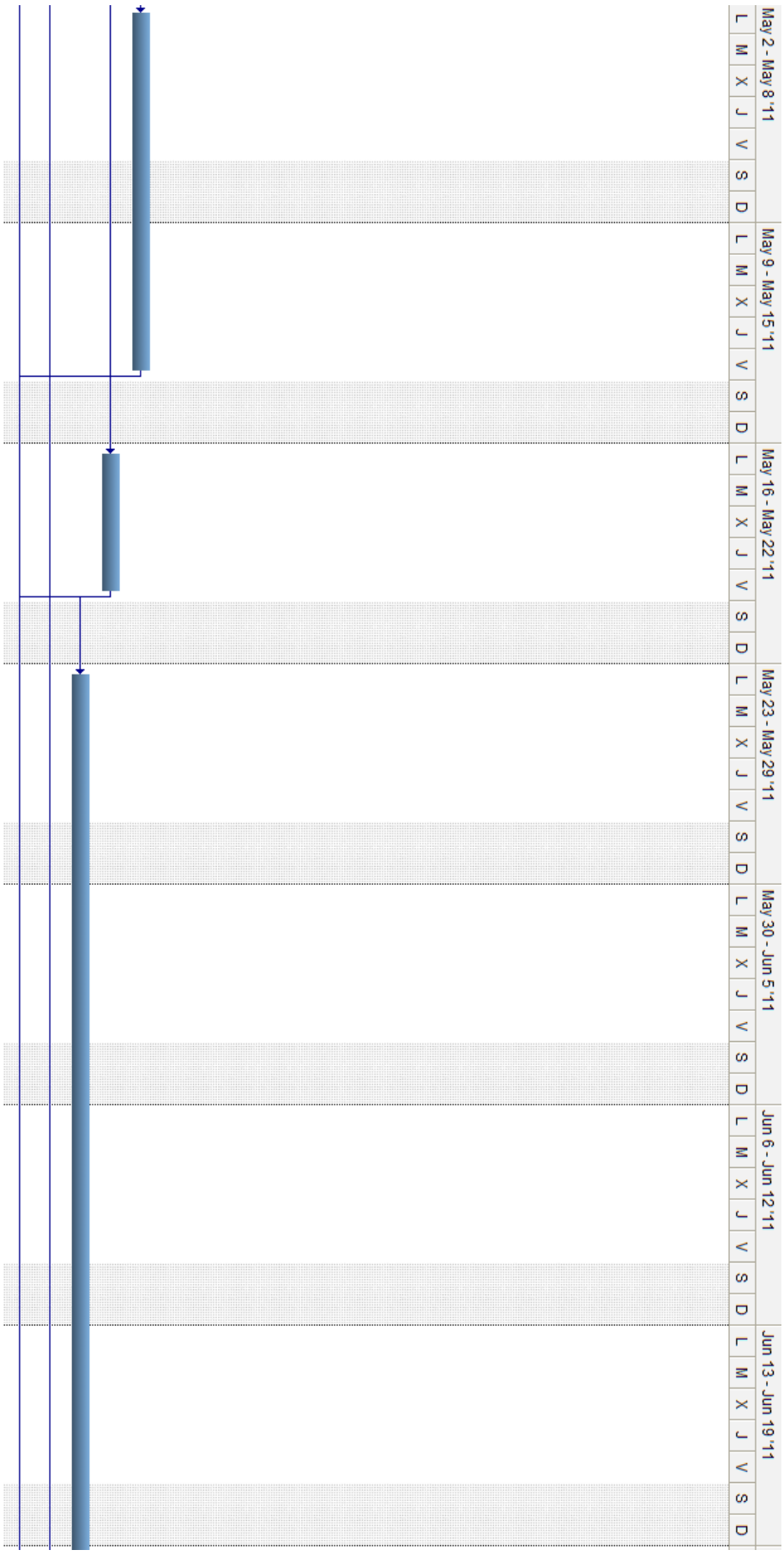

Figura 3.8: Diagrama de Gantt de la planificación del proyecto página: 4/6

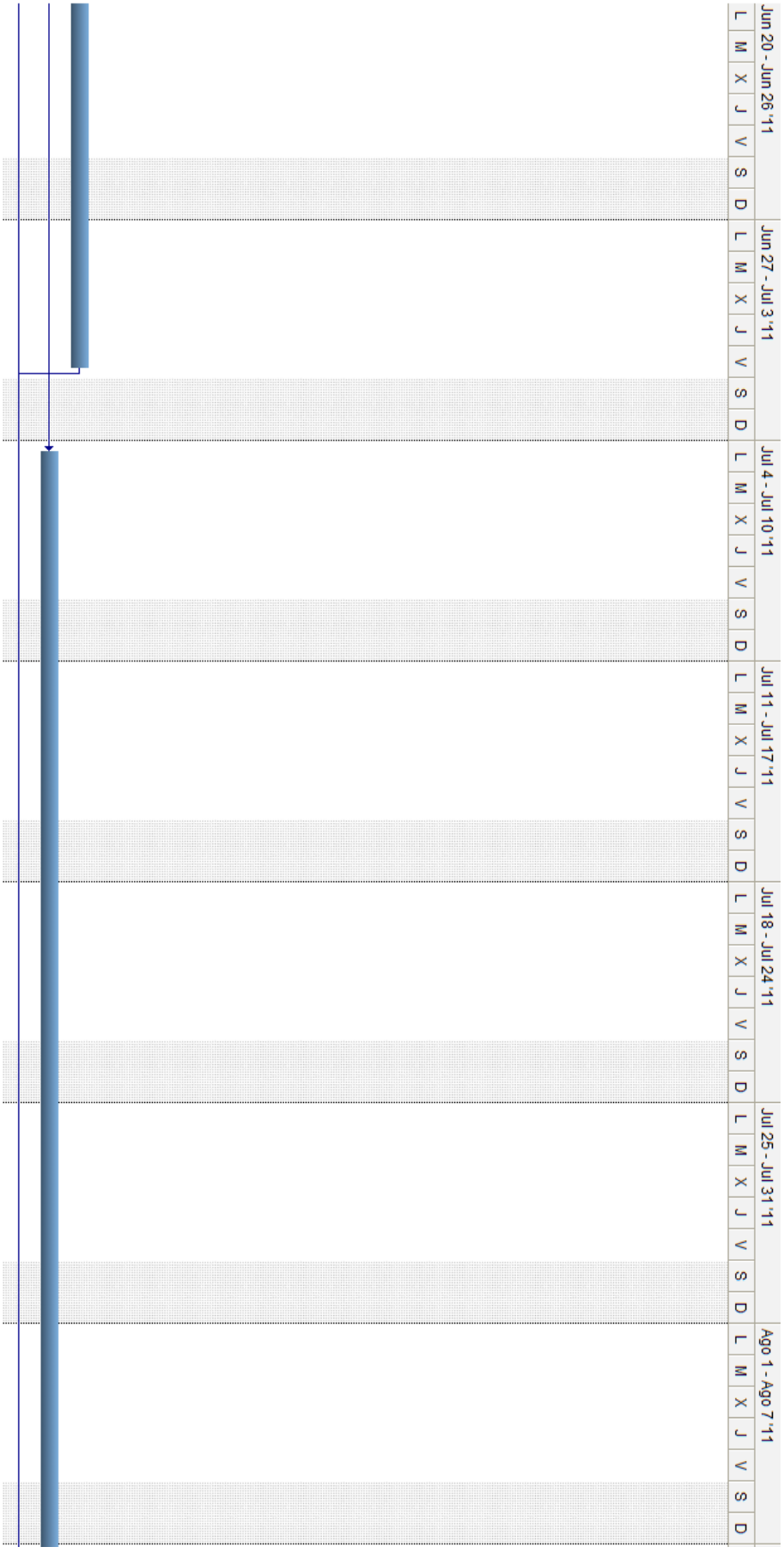

Figura 3.9: Diagrama de Gantt de la planificación del proyecto página: 5/6

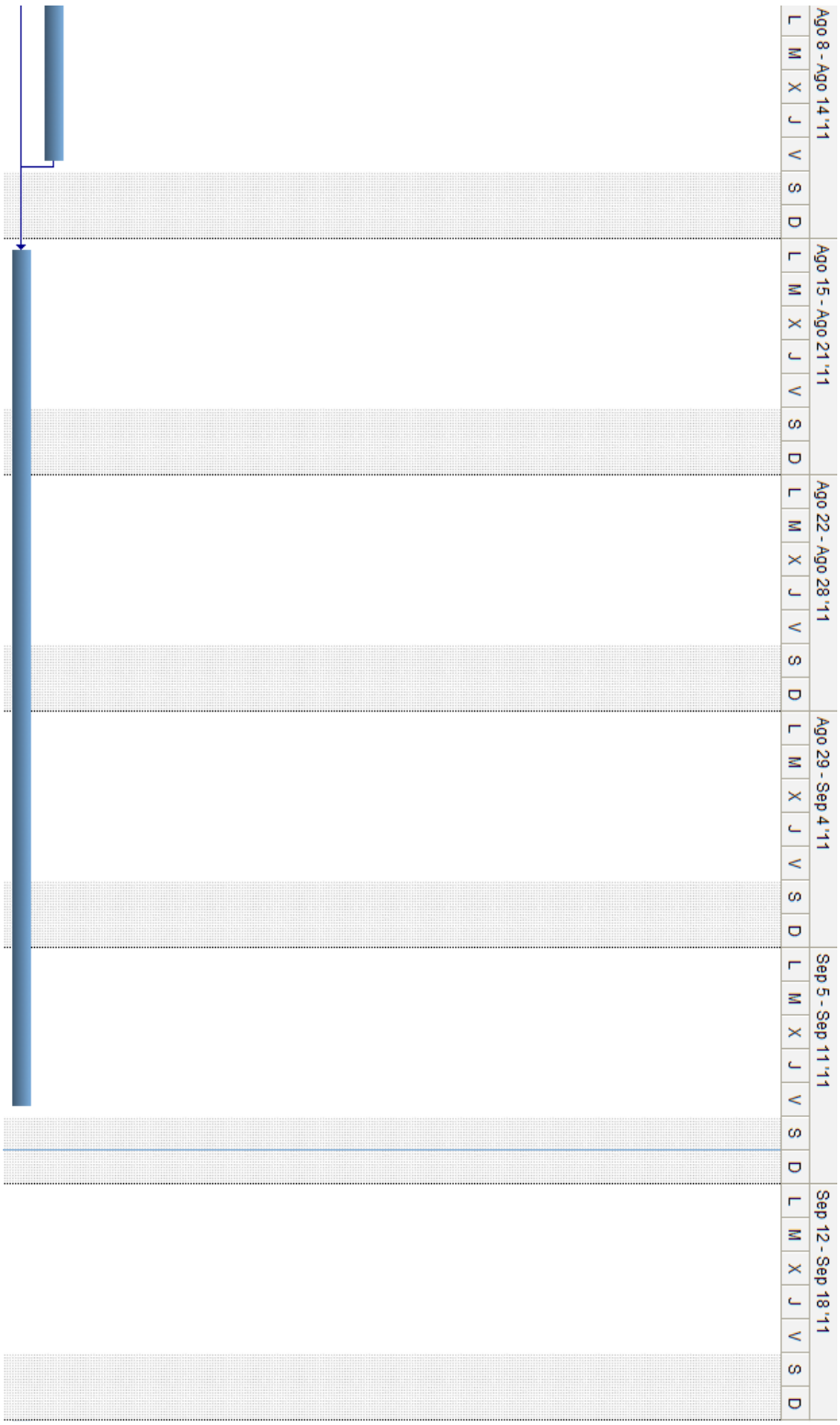

Figura 3.10: Diagrama de Gantt de la planificación del proyecto página: 6/6

4

Desarrollo

En el transcurso de este capítulo se hablará de la adaptación realizada sobre la librería OpenSceneGraph para compatibilizarla con la plataforma Android. Se hablará de las principales etapas en el desarrollo del trabajo, los diferentes problemas surgidos, sus soluciones y las diferentes aplicaciones de pruebas que se han generado.

4.1. Adaptación de la librería OpenSceneGraph

Cuando se adapta una librería a una plataforma, es importante tener en cuenta sus características. En la sección 3 se han cubierto ampliamente las características particulares de Android, su metodología de programación y las posibilidades de programación de código C/C++ nativo.

OSG comenzó siendo diseñada para sistemas Unix. Posteriormente, con el transcurso de los años, ha ido incorporando diferentes plataformas como Windows, Linux, iOS y algunas versiones específicas de Unix. El código que lo forma está escrito en C++ usando la Standard Template Library (STL) sin incorporar ningún segmento de código máquina, por lo tanto, a excepción de las partes más específicas de los sistemas operativos como los hilos y la gestión de librerías, el código puede ser portado sin ningún cambio a cualquier plataforma que permita usar C++ y STL.

En el caso de Android, como ya se ha comentado, C++ se puede emplear como len-

guaje en la programación nativa, sin embargo la implementación de las librerías de C no corresponde a la estándar. Android ha reimplementado la mayor parte de librerías de C en una llamada Bionic. El objetivo de esta librería era crear una implementación eficiente y que fuese lo más simple posible, a cambio no tiene implementados todos los métodos que se encuentran en la de referencia.

Otra limitación que imponen las librerías de Android viene debida a la implementación de la librería STL. Hasta la versión cinco del Native Development Kit (NDK) no podíamos contar con una versión estándar de la GNU STL. Google ofrecía una simplificada con un soporte mínimo. En la versión 5 añadió el soporte para emplear una librería STL independiente (STLport) y la GNU STL, sin embargo, el uso de ambas librerías tiene limitaciones. La versión de STLport en Android, todavía no tiene soporte para RTTI o las excepciones de C++. Por su parte, la librería GNU STL únicamente se puede enlazar de forma estática, lo que conlleva un sobrecoste en el tamaño de las aplicaciones cuando varias librerías comparten su uso.

El proceso de adaptación de la librería tuvo tres fases principales:

- Compilación de la librería mediante los scripts Android NDK e integración en el sistema de compilación CMake de la librería OSG.
- Generación de los programas de prueba.
- Creación de la documentación y programas de ejemplo para la comunidad OSG.

4.2. Integración de un sistema de compilado Android NDK en la librería OpenSceneGraph

Para este trabajo, inicialmente, se crearon archivos de script siguiendo la sintaxis de los ficheros de compilación .mk del NDK de Android, así como crear un fichero por cada librería y plugin de OSG. Cada uno de esos tenía la estructura siguiente.

```
#ANDROID makefile osg
```

```
LOCAL_PATH := /home/jizquierdo/
    repositorio_OpenSceneGraph/src/osg

include $(CLEAR_VARS)

LOCAL_CPP_EXTENSION := cpp

LOCAL_LDLIBS := -lOpenThreads -lGLESv1_CM -ldl

LOCAL_MODULE := osg

LOCAL_SRC_FILES := AlphaFunc.cpp AnimationPath.cpp
    ...

LOCAL_C_INCLUDES := /home/jizquierdo/
    repositorio_OpenSceneGraph/include /home/jizquierdo
    /repositorio_OpenSceneGraph/android_build_gles1/
    include

LOCAL_CFLAGS := -DANDROID -DOSG_LIBRARY_STATIC

LOCAL_CPPFLAGS := -DANDROID -DOSG_LIBRARY_STATIC
```

Código 4.1: Fragmento de código de un archivo de script para una librería de OSG.

Analizando los elementos del script:

- `LOCAL_PATH`: Directorio de referencia sobre el que se buscarán el resto de archivos.
- `include $(CLEAR_VARS)`: Incluye el script NDK de limpiar variables. Limpia cualquier variable excepto `LOCAL_PATH`.
- `LOCAL_CPP_EXTENSION`: Define la extensión de los archivos.
- `LOCAL_LDLIBS`: Define las librerías que deberán enlazarse con la librería actual.

- `LOCAL_MODULE`: Define el nombre de la librería.
- `LOCAL_SRC_FILES`: Lista de los archivos que se han de compilar para esta librería.
- `LOCAL_C_INCLUDES`: Lista de directorios con las cabeceras.
- `LOCAL_CFLAGS/LOCAL_CPPFLAGS`: Definiciones que se incluyen para todos los archivos.
- `include $(BUILD_STATIC_LIBRARY)`: Incluye el script que configura esta librería como estática.

Con todas las librerías que forman OSG preparadas con su script de compilación NDK, únicamente faltó añadir los scripts principales para compilar los distintos módulos de la librería. El script inicial está formado por dos archivos: Android.mk y Application.mk

```
#ANDROID makefile in src

LOCAL_PATH := $(call my-dir)
SRC_ROOT := $(LOCAL_PATH)

include /home/jizquierdo/repositorio_OpenSceneGraph/3
    rdparty/zlib/Android.mk
 . . .
include /home/jizquierdo/repositorio_OpenSceneGraph/
    android_build_gles1/src/osgPlugins/pvr/Android.mk
```

Código 4.2: Fragmento de código del archivo principal "Android.mk".

Como se puede ver en el extracto del archivo principal, podemos distinguir las siguientes órdenes:

- `LOCAL_PATH`: Variable que guarda el directorio actual obtenido mediante `$(call my-dir)`

- `SRC_ROOT`: Define el directorio de archivos de código mediante la variable `$(LOCAL_PATH)`
- `include`: Lista de los archivos de cada librería.

```
#ANDROID APPLICATION MAKEFILE
APP_BUILD_SCRIPT := $(call my-dir)/Android.mk
APP_PROJECT_PATH := $(call my-dir)

APP_OPTIM := release

APP_PLATFORM := android-5
APP_STL := gnustl_static
APP_CPPFLAGS := -fexceptions -frtti

APP_ABI := armeabi armeabi-v7a

APP_MODULES :=  zlib OpenThreads osg osgDB osgUtil
   osgGA osgText osgViewer osgAnimation osgFX
   osgManipulator osgParticle osgPresentation
   osgShadow osgSim osgTerrain osgWidget osgVolume
```

Código 4.3: Fragmento de código del archivo principal "Application.mk".

- `APP_BUILD_SCRIPT`: Variable que indica donde se encuentra el archivo Android.mk.
- `APP_PROJECT_PATH`: Variable que indica al compilador la ruta sobre la que se trabaja. La información del directorio se obtiene con: `$(call my-dir)`
- `APP_OPTIM`: Variable que indica al compilador si debe compilarlo con o sin símbolos de debug.
- `APP_PLATFORM`: Variable que indica al compilador la versión de plataforma.
- `APP_STL`: Configura la STL que se va a emplear.
- `APP_CPPFLAGS`: Configuraciones de C++.

- `APP_ABI`: Arquitecturas para las que se realiza la compilación.
- `APP_MODULES`: Lista de librerías que se han de compilar para este proyecto.

4.2.1. Problemas presentados durante la compilación

Los primeros intentos de compilación, evidenciaron una serie errores. Algunos de estos ya se habían considerado y detectado como tales durante la fase de análisis:

- Errores por las diferencias en la librería pThreads.
- Errores por la falta de soporte para los tipos de carácter ancho (wchar).
- Definiciones de cabeceras.
- Definición de tipos GL.

Como se ha dicho anteriormente, la implementación de las librerías de C no es completa, la siguiente lista detalla algunos de los métodos que, existiendo en la especificación en Linux y siendo requeridos por OSG, no aparecen en la implementación de Android.

- pThreads_testcancel
- pThreads_cancel
- pThreads_setcancelstate
- pThreads_setcanceltype

Tal y como se indica en la documentación de Google:

```
pthread_cancel() will *not* be supported in Bionic,
    because doing this would involve making the C
    library significantly bigger for very little
    benefit.
```

Es decir, las instrucciones cancelación de hilos no serán implementadas debido al aumento de tamaño que provocaría en la librería y el poco beneficio de su implementación. Por lo tanto, no es factible incorporar la funcionalidad de dichos elementos.

La librería OSG, emplea para abstraer la gestión de hilos la librería OpenThreads (OT). Esta tiene implementaciones dependientes para cada sistema operativo, y una de ellas es la de pThreads, el problema surge por la falta de dichos métodos. Tras estudiar cuidadosamente el uso de los comandos de cancelado de OT por parte de OSG, se pudo concluir que no se empleaba la cancelación a lo largo de todo el código, lo que permitió emplear el siguiente procedimiento: Generar rutas vacías que serán empleadas, o no, en tiempo de compilación mediante el uso de una constante de plataforma (ANDROID). Esto se generará mediante macros de precompilación introducidas en el código de la librería OT.

Es cierto que esa solución no es factible para todo programa que necesite la operación de cancelación de hilos, pero el presente trabajo únicamente se ha centrado en la compatibilidad de la librería OSG, no en el uso general de la librería OT para otras aplicaciones. **Toda aplicación que quiera portarse a Android, deberá replantear su uso de la cancelación de hilos ya que, como está publicado en la documentación oficial; "jamás" se incorporará la operación de cancelación de hilos en la librería Bionic**.

Los errores debido a la falta de soporte para los tipos de caracteres anchos, caracteres que necesitan más de un byte, son debidos a que wchar no se encuentra realmente recogido en todas las versiones de C, por ello, Google no los ha incorporado. El propósito de los caracteres anchos es la representación de caracteres en codificaciones diferentes de ANSI (UTF16/UTF32), en la librería únicamente afectan para el uso de cadenas que no empleen ASCII en la base de datos (osgDB). Existen dos soluciones posibles para este problema. Por un lado, el programador puede usar una versión modificada del NDK que incluye el soporte para wchar, esta solución es desaconsejable por los problemas de compatibilidad que puedan aparecer en el futuro. Desde el inicio del proyecto, se ha buscado seguir lo más cercanamente posible a los estándares de Google. **Recurrir a versiones modificadas sería algo que podría hacer que la librería dejase de poder compilarse con el tiempo**.

Nuestra solución sigue el mismo patrón que la solución ya implementada para OSG en algunas plataformas compatibles con Linux sin wchar. Creamos una redefinición de tipo sobre un char, si bien esto no da las funciones reales, es una solución que ha sido aceptada por la comunidad de OSG, delegando la vigilancia de usar caracteres ASCII en la base de datos de OSG a los programadores.

Los problemas que aparecieron debido a las cabeceras, se deben a la selección de cabeceras dependiendo de la plataforma objetivo. Estos cambios fueron menores y repartidos a lo largo de diversos archivos. Consistían en incluir en las condiciones de uso la variable "ANDROID". Esta es la razón por la cual en los scripts de compilación se incluye la línea: `LOCAL_CPPFLAGS := -DANDROID`.

Al añadir esa flag al compilador, incluimos una constante que es usada como definición. lo que permite realizar condiciones de precompilación específicas para Android.

Finalmente el último error que encontramos se debe a los tipos de OpenGL. OSG realiza redefiniciones de tipos y de constantes OpenGL, evita propagar los archivos de OpenGL, debido a que pueden haber cambios por plataforma y versión, propagando únicamente las definiciones y constantes que él crea. Ha sido necesario crear una ruta de elecciones para la plataforma Android que definiese exclusivamente los tipos de OpenGL siguiendo la especificación de OpenGL ES 1.0 y 2.0 que emplean los mismos tipos.

La inclusión de estos cambios y la cadena de scripts de compilación NDK permiten generar los archivos binarios para ser enlazados de forma estática con nuestras aplicaciones. En el punto actual del desarrollo, únicamente se ha permitido emplear la compilación de los componentes de OSG como librerías estáticas. Esto se debe a dos razones:

- La librería STL contra la que se enlazan las librerías OSG es estática, si cada módulo se compilara para ser dinámico incluiría para si las partes que requiere de la librería STL. Esto supone que los binarios finales tendrían un peso muy superior al incluir elementos de código repetido al no poder compartir de forma dinámica la librería STL. En este momento, al ser compiladas de forma estática, únicamente se introducen los métodos usados de la STL cuando se compila el programa final.

- La carga de librerías dinámica en OSG está creada pensando en un sistema de sobremesa donde las librerías han sido instaladas en un lugar determinado. En nuestro caso, los métodos de apertura y la situación de los elementos de la librería variarán de dispositivo a dispositivo.

Aun así, es posible emplear la librería OSG de forma estática, siempre y cuando empleemos las macros de definición de módulos y plugins para que el núcleo de OSG

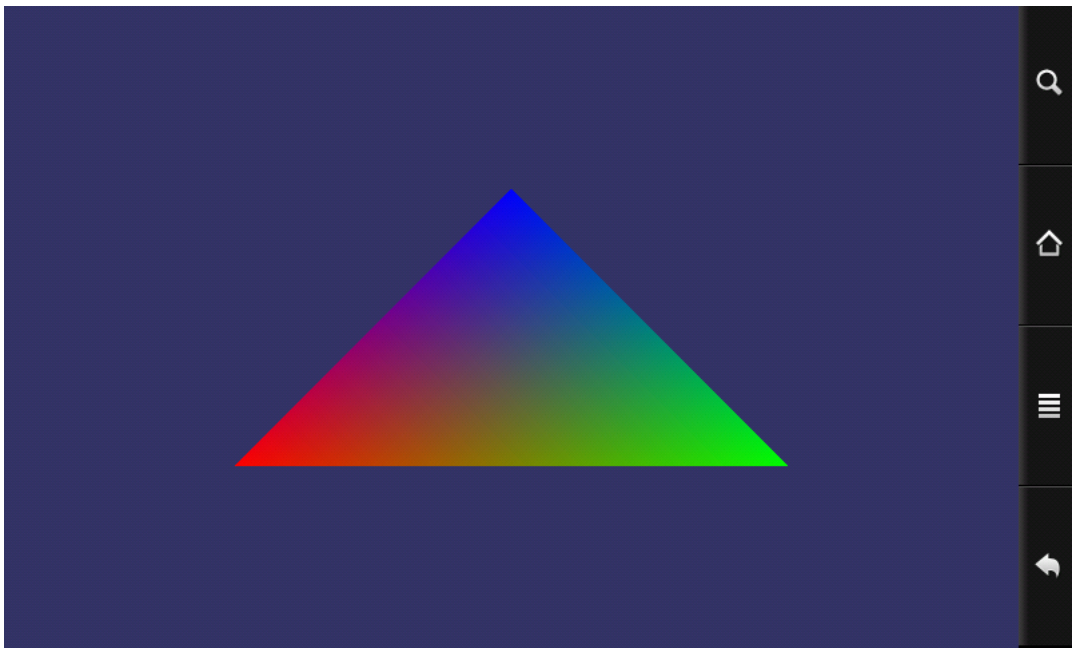

Figura 4.1: Imagen del primer programa funcional de OSG en Android dibujando un triángulo RGB.

detecte que puede usarlos ya que se encuentran en su código. Con esto es posible obtener nuestros primeros resultados visibles como se observa en la figura 4.2.1

4.2.2. Integración de los scripts NDK en los scripts CMake

En la última fase del proyecto, se buscó una forma de integrar de forma sencilla y poco intrusiva los scripts de compilación. OSG es una librería con más de 400 archivos de código diferente, cada vez que se introducen cambios significativos como modificaciones de nombre, nuevos archivos, nuevas opciones; se han de retocar los diferentes archivos de script CMake que ya posee OSG, si añadimos una segunda secuencia de compilado únicamente para Android, esto supone el doble de trabajo, además, los scripts de Android deben ser retocados en mayor o menor medida cuando el usuario desee ajustar algunos elementos. Por lo tanto se requiere de un método que permita reajustar de forma sencilla todos los scripts cuando se ajustan los scripts de CMake. A la vez es necesario que podamos dar posibilidad a que los usuarios puedan definir sus opciones de plataformas de destino, versiones de NDK, etc. Recordemos que OSG emplea CMake como sistema de scripts para asegurar la generación automática de scripts de compilación en cada plataforma.

Compilar para Android tiene una serie de particularidades:

- Es una compilación cruzada, se realiza fuera de la plataforma destino.
- Aunque actualmente Google emplee una versión de Gcc con una estructura de archivos similar a la de Linux, esto puede cambiar sin previo aviso, lo que haría inútil el uso (todavía experimental) de los scripts de compilación cruzada que puede generar CMake.

El primer intento de unificación se realizó añadiendo un archivo para obligar a CMake a emplear una toolchain diferente a la del sistema operativo origen, esto permite realizar la compilación cruzada directamente con CMake. Esta aproximación hubo de ser descartada tras comprobar que, con la versión NDK r5, CMake emitía falsos positivos en sus comprobaciones durante la compilación cruzada. Ante esta tesitura se decidió abordar un sistema que permitiese generar automáticamente los scripts de NDK a partir de los scripts de CMake cuando el usuario lo ejecutase en su plataforma.

CMake es un lenguaje de scripting con la capacidad de programar sobre él, esto permite extender las operaciones del programa sin necesidad de integrar nuevas reglas de generación de archivos en su código fuente. De esta manera, se pueden incluir elementos y realizar operaciones para las que no estaba diseñado originalmente. Para este trabajo, se empleará esta capacidad para hacer que el lenguaje genere los archivos de script NDK a partir de unos scripts modelos diseñados para tal propósito. Cuando CMake ejecuta los contenidos del script, crea los archivos copiando la estructura

de los archivos modelos e insertando las definiciones necesarias para la compilación: nombres de archivos a compilar, opciones de compilación, nombre del módulo, dependencias, etc.

Para realizar esto, se creó una macro que permitía crear el archivo de script NDK final de una librería. La macro usa, como estructura modelo de código el fragmento: 4.1. Los diversos valores son rellenados por los el contenido de las variables que emplea CMake para generar los scripts de compilación normalmente.

El funcionamiento de esta se puede ver en el siguiente fragmento: 4.4

```
MACRO(SETUP_ANDROID_LIBRARY LIB_NAME)

  foreach(arg ${TARGET_LIBRARIES})
      set(MODULE_LIBS "${MODULE_LIBS} -l${arg}")
  endforeach(arg ${TARGET_LIBRARIES})

  foreach(arg ${TARGET_SRC})
      string(REPLACE "${CMAKE_CURRENT_SOURCE_DIR}/" ""
          n_f ${arg})
      set(MODULE_SOURCES "${MODULE_SOURCES} ${n_f}")
  endforeach(arg ${TARGET_SRC})

      #SET(MODULE_INCLUDES "${CMAKE_SOURCE_DIR}/
          include include")
  GET_DIRECTORY_PROPERTY(loc_includes
     INCLUDE_DIRECTORIES)
  foreach(arg ${loc_includes})
      IF(NOT "${arg}" MATCHES "/usr/include" AND NOT "
          ${arg}" MATCHES "/usr/local/include")
          set(MODULE_INCLUDES "${MODULE_INCLUDES} ${
              arg}")
      ENDIF()
  endforeach(arg ${loc_includes})

  GET_DIRECTORY_PROPERTY(loc_definitions
```

```
    COMPILE_DEFINITIONS)
foreach(arg ${loc_definitions})
    set(DEFINITIONS "${DEFINITIONS} -D${arg}")
endforeach(arg ${loc_definitions})

message(STATUS "name: ${LIB_NAME}")

set(MODULE_NAME             ${LIB_NAME})
set(MODULE_DIR              ${CMAKE_CURRENT_SOURCE_DIR})
set(MODULE_FLAGS_C            ${DEFINITIONS})
set(MODULE_FLAGS_CPP        ${DEFINITIONS})

IF(OSG_GLES1_AVAILABLE)
    SET(OPENGLES_LIBRARY -lGLESv1_CM)
ELSEIF(OSG_GLES2_AVAILABLE)
    SET(OPENGLES_LIBRARY -lGLESv2)
ENDIF()

set(MODULE_LIBS "${MODULE_LIBS} ${OPENGLES_LIBRARY}
    -ldl")

if(NOT CPP_EXTENSION)
    set(CPP_EXTENSION "cpp")
endif()

IF(NOT MODULE_USER_STATIC_OR_DYNAMIC)
    MESSAGE(FATAL_ERROR "Not defined
        MODULE_USER_STATIC_OR_DYNAMIC")
ENDIF()

IF("MODULE_USER_STATIC_OR_DYNAMIC" MATCHES "STATIC")
    SET(MODULE_BUILD_TYPE "\$\(BUILD_STATIC_LIBRARY
        \)")
ELSE()
```

```
        SET(MODULE_BUILD_TYPE "\$\(BUILD_DYNAMIC_LIBRARY
            \)")
    ENDIF()

    set(ENV{AND_OSG_LIB_NAMES} "$ENV{AND_OSG_LIB_NAMES} 
        ${LIB_NAME}")
    set(ENV{AND_OSG_LIB_PATHS} "$ENV{AND_OSG_LIB_PATHS}
        include ${CMAKE_CURRENT_BINARY_DIR}/Android.mk \n
        ")

    configure_file("${OSG_ANDROID_TEMPLATES}/Android.mk.
        modules.in" "${CMAKE_CURRENT_BINARY_DIR}/Android.
        mk")

ENDMACRO()
```

Código 4.4: Fragmento de código de la macro para generar archivos de librería Android.

Con esta macro, el usuario era capaz de crear los scripts para las bibliotecas de Android, sin embargo para poder integrarla de forma limpia, hubo que realizar una refactorización de los scripts CMake de OSG. Las partes de decisiones, opciones y configuraciones de mantenimiento se mantuvieron igual. La única parte que se cambió fue la definición de módulos en CMake, estas definiciones fueron encapsuladas dentro de una macro genérica que se puede ver en: 4.5. Dentro de esa macro, se observa como incluimos la toma de decisión de usar, o no, la generación de archivos de script de Android, de esta manera, evitábamos crear comprobaciones de plataforma en todos los módulos de CMake. El código queda más limpio y seguro al reducir las comprobaciones a una única función que es llamada por todos los módulos.

```
MACRO(SETUP_LIBRARY LIB_NAME)
    IF(ANDROID)
```

```
    SETUP_ANDROID_LIBRARY(${LIB_NAME})
  ELSE()
    SET(TARGET_NAME ${LIB_NAME} )
    SET(TARGET_TARGETNAME ${LIB_NAME} )
    ADD_LIBRARY(${LIB_NAME}
      ${OPENSCENEGRAPH_USER_DEFINED_DYNAMIC_OR_STATIC}
      ${TARGET_H}
      ${TARGET_H_NO_MODULE_INSTALL}
      ${TARGET_SRC}
    )
    SET_TARGET_PROPERTIES(${LIB_NAME} PROPERTIES
        FOLDER "OSG_Core")
    IF(TARGET_LABEL)
      SET_TARGET_PROPERTIES(${TARGET_TARGETNAME}
          PROPERTIES PROJECT_LABEL "${TARGET_LABEL}")
    ENDIF(TARGET_LABEL)

    IF(TARGET_LIBRARIES)
      LINK_INTERNAL(${LIB_NAME} ${TARGET_LIBRARIES})
    ENDIF()
    IF(TARGET_EXTERNAL_LIBRARIES)
      LINK_EXTERNAL(${LIB_NAME} ${
          TARGET_EXTERNAL_LIBRARIES})
    ENDIF()
    IF(TARGET_LIBRARIES_VARS)
      LINK_WITH_VARIABLES(${LIB_NAME} ${
          TARGET_LIBRARIES_VARS})
    ENDIF(TARGET_LIBRARIES_VARS)
    LINK_CORELIB_DEFAULT(${LIB_NAME})

  ENDIF()
  INCLUDE(ModuleInstall OPTIONAL)
ENDMACRO(SETUP_LIBRARY LIB_NAME)
```

Código 4.5: Fragmento de código de la macro para definir librerías.

Finalmente, se generaron unas macros especiales para asegurar el funcionamiento estándar de los comandos ”make“ y ”make install“ que se suelen emplear. De esta manera, el comando make ejecuta el script del NDK sobre los archivos de script que se han generado. Posteriormente con el comando de instalación, creamos un directorio limpio con las cabeceras y las librerías en todas las versiones de plataforma que hayan sido generadas.

Independientemente de la labor para permitir la generación del script de compilación final, se han incluido secciones de código de búsqueda y configuración para complementar las opciones de los desarrolladores. Se ha creado una serie de scripts para la detección automática del NDK de Android, así como del paquete de librerías Third Party para evitar que los desarrolladores tengan que configurar estas opciones manualmente. Finalmente, durante las últimas fases de desarrollo, se ha incluido la opción para añadir o quitar las arquitecturas que se compilan, las optimizaciones que se emplearán y la compilación con símbolos de depuración.

Esta última tanda de opciones se ha incluido tras descubrir que algunos dispositivos, con chipset Tegra 2, tenían incompatibilidades con algunas optimizaciones al carecer de las unidades apropiadas en el procesador. Por ello en el trabajo final se ha optado por ofrecer la mayor variabilidad de opciones posible para que el programador las ajuste a sus necesidades.

4.3. Creación de las aplicaciones de Test

En esta fase, se generaron una serie de aplicaciones para comprobar las funcionalidades de OSG y como se puede realizar un renderizado interactivo de terrenos con el uso del grafo de escena.

La lista de aplicaciones generadas son:

- Aplicación test modelos baja resolución OpenGL ES 1.0
- Aplicación test modelos baja resolución OpenGL ES 2.0

- Aplicación test modelos alta resolución OpenGL ES 1.0
- Aplicación test modelos alta resolución OpenGL ES 2.0
- Representación de terrenos pregenerados escala planetaria.
- Representación de terrenos pregenerados escala local planar.
- Aplicación de representación de terrenos en tiempo de dibujado.

4.3.1. 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.

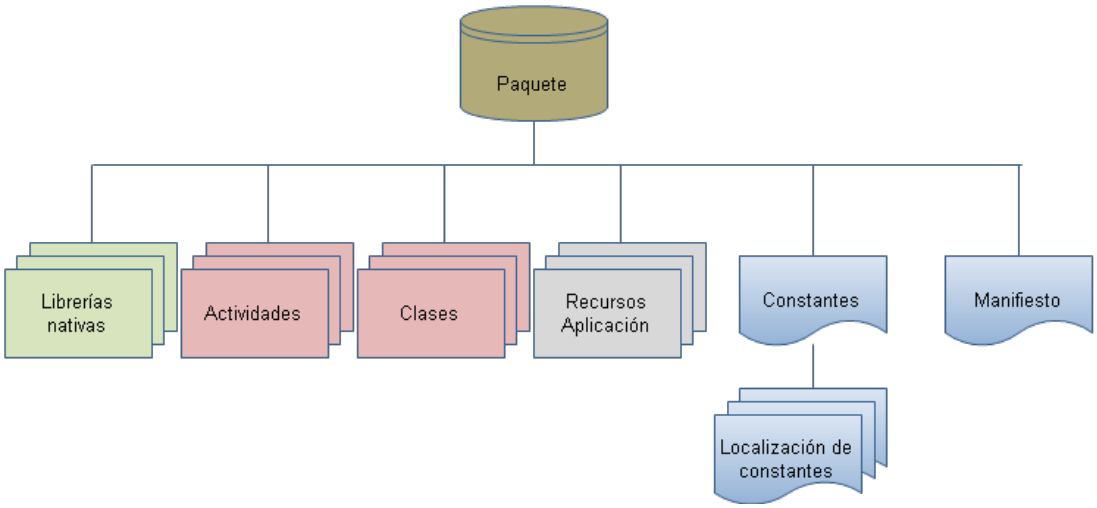

Figura 4.2: Diagrama de los contenidos de un paquete APK.

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 Activity y que serán ejecutadas en la aplicación. No se puede realizar una intención de ejecución 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, actualmente existen restricciones por tamaños de pantalla, densidades, compatibilidad con librerías, la presencia de determinados componentes hardware, la posibilidad de compresión de texturas, etc.

Centrándonos más en la estructura propia se puede ver el directorio que tiene las clases compiladas en formato Dalvik (.dex) y el directorio de recursos. Los recursos de una aplicación se encuentran en esta sección. Estos pueden, o no, estar comprimidos, la elección la toma de forma arbitraria el propio empaquetador del SDK. Todos los recursos de una aplicación se sitúan en el directorio de recursos. Los tipos de recursos que se integran son gráficos, sonoros y archivos de definiciones, configuraciones y constantes.

En Android el empleo de archivos .xml para guardar las constantes que tu programa pueda utilizar así como otras definiciones, es una práctica aceptada y consolidada dentro de la API. Esto sirve para separar el texto que se pueda encontrar en una aplicación y ofrecerlo como una variable directamente en el programa. De esta manera se evita mezclar el código de la aplicación con el texto, lo que permite además una mayor facilidad para la localización de un programa. Android permite que, dependiendo del idioma, se utilice un archivo de constantes de texto u otro sin necesidad de crear código específico para cada lenguaje. Una aplicación únicamente pide el valor del recurso texto con nombre X, ya que la API se encarga de encontrar oportunamente el valor dependiendo del idioma.

De la misma manera, en el directorio de recursos es donde se incluyen los archivos .xml que definen las interfaces gráficas o los menús. Estos archivos son totalmente opcionales, ya que la construcción de las interfaces gráficas y menús se puede crear por código sin necesidad de leer los archivos .xml.

Finalmente, si una aplicación es nativa, 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 texturas.

Una aplicación, así pues, 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 en el uso normal de un teléfono móvil. Un usuario normal espera poder emplear su agenda diaria en su teléfono y, si recibe una llamada poder contestar a la llamada 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, muchas veces la cogerá sin salvar los datos; por lo tanto el comportamiento que desearía el usuario es que la aplicación permanezca a la espera. De esa manera, el usuario, si lo desea, la puede devolver a ejecución desde el punto en el que la dejó en espera.

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 es a nivel de aplicación sino a nivel de actividad. Este ciclo se

puede ver en la ilustración: 4.3.1

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 que cada actividad realice los ajustes necesarios para su comportamiento.

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, esta actividad 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. Adicionalmente, Android incluye los servicios, a diferencia de los Threads, los servicios 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

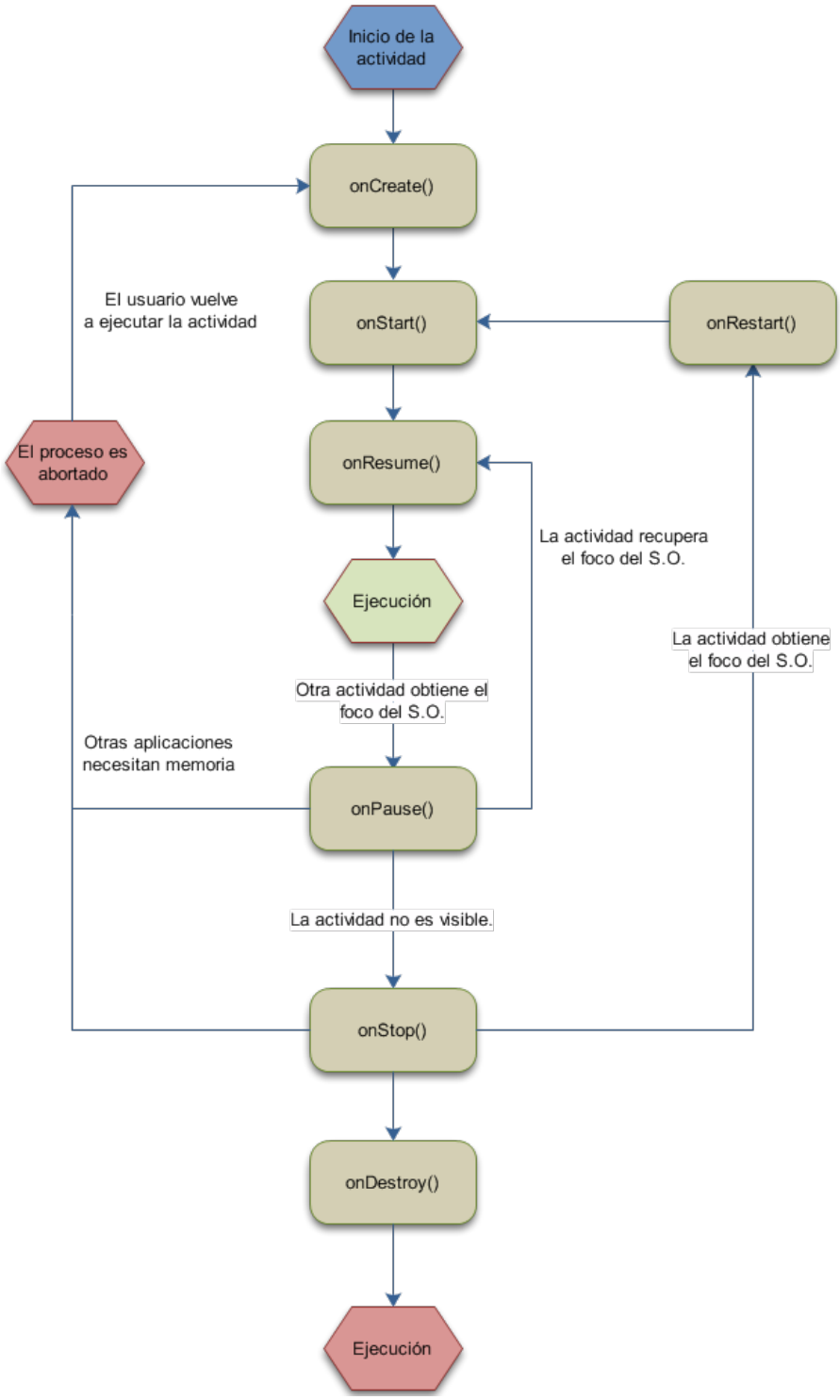

Figura 4.3: Diagrama del ciclo de vida de una actividad en Android.

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. En este trabajo, no se van a emplear todas las posibilidades que están presentes en la plataforma. Hay que prestar una atención especial al error ANR y se debe procurar inicializar la biblioteca OSG en un hilo para evitar el bloqueo al iniciar las aplicaciones en Android.

Después de la inclusión de la Native Activity en la versión 2.3 de Android, las aplicaciones de OSG sobre Android pueden tener dos estructuras diferentes. La principal diferencia que existe entre ambas estructuras es el uso, o no del framework nativo. Las ventajas de este método ya han sido discutidas, sin embargo hay que comentar también que si el desarrollador desea usar la GUI u otros elementos de la API que todavía no han sido expuestos a nivel nativo, debe emplear un puente JNI para comunicarse con el framework superior de Dalvik.

Una aplicación puramente nativa, sigue de igual manera el ciclo de vida expuesto anteriormente. Las únicas diferencias reales con una actividad de Android no nativa son: el uso del lenguaje C/C++ y la imposibilidad de emplear todo el framework de Dalvik, ya que este no se encuentra totalmente expuesto al desarrollador para su uso en el nivel nativo. Una de las ventajas que tienen las aplicaciones totalmente nativas es la generación de un contexto gráfico desde el nivel nativo. Hasta la aparición de esta posibilidad, la parte Dalvik era la dueña del contexto EGL y no podía ser transmitido para incorporarlo en OSG.

Por el contrario, una aplicación que no sea puramente nativa, debe emplear una actividad Java que controle y encapsule todos los eventos de entrada. En el momento de inicio de la actividad, ha de crearse un contexto mediante EGL, debido a que este se encuentra en el nivel no nativo y no puede ser compartido con la parte nativa de la aplicación OSG, además la clase que gestiona el contexto, debe encapsular las llamadas de renderización y cambios de representación.

Las figuras 4.5 y 4.4 representan una modelo básico de estructura de aplicación OSG para Android. La figura de la versión parcialmente nativa, es la que se encuentra implementada, con variaciones, en los prototipos y pruebas de este trabajo. Debido al coste económico de los dispositivos necesarios para hacer tests con una aplicación enteramente nativa, la estructura presentada es un esbozo que sigue los patrones de diseño marcados por Google. Su funcionamiento no ha sido comprobado físicamente. Como se puede ver en la figura: 4.5 la actividad principal se genera como Java. Esta actividad se comunica con una clase derivada de GLSurfaceView (EGLview) que se

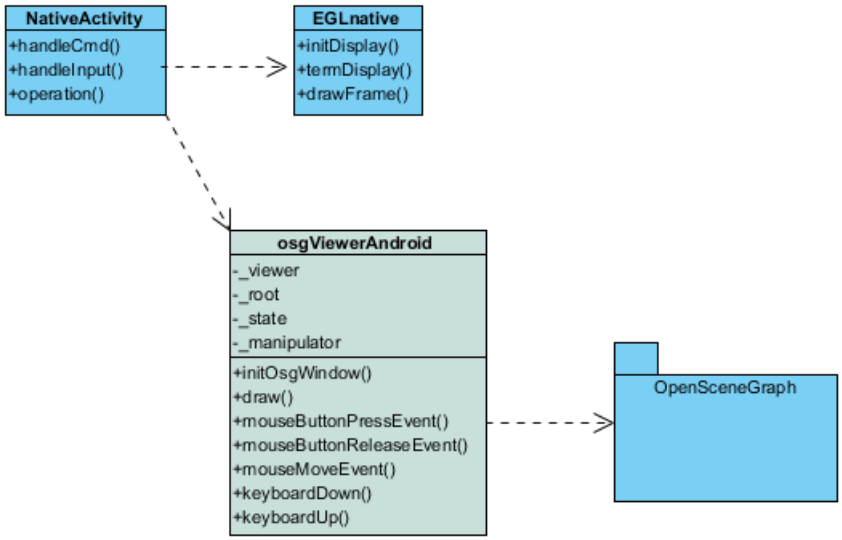

Figura 4.4: Diagrama de clases de la estructura de una aplicación OSG en Android usando NativeActivity.

encarga de realizar la configuración del contexto gráfico y de registrar la función de renderizado. Para realizar las llamadas a OSG utilizan la clase osgNativeLib, esta clase contiene los métodos expuestos mediante el puente JNI. Ya en el nivel nativo, podemos ver como los métodos nativos emplean la clase osgViewerAndroid para realizar sus operaciones. En contraposición, la versión puramente nativa representada en: 4.4 únicamente tiene una clase general con los métodos a registrar para su uso por parte de la NativeActivity, estos a su vez, pueden acceder a la clase osgViewerAndroid para gestionar la representación.

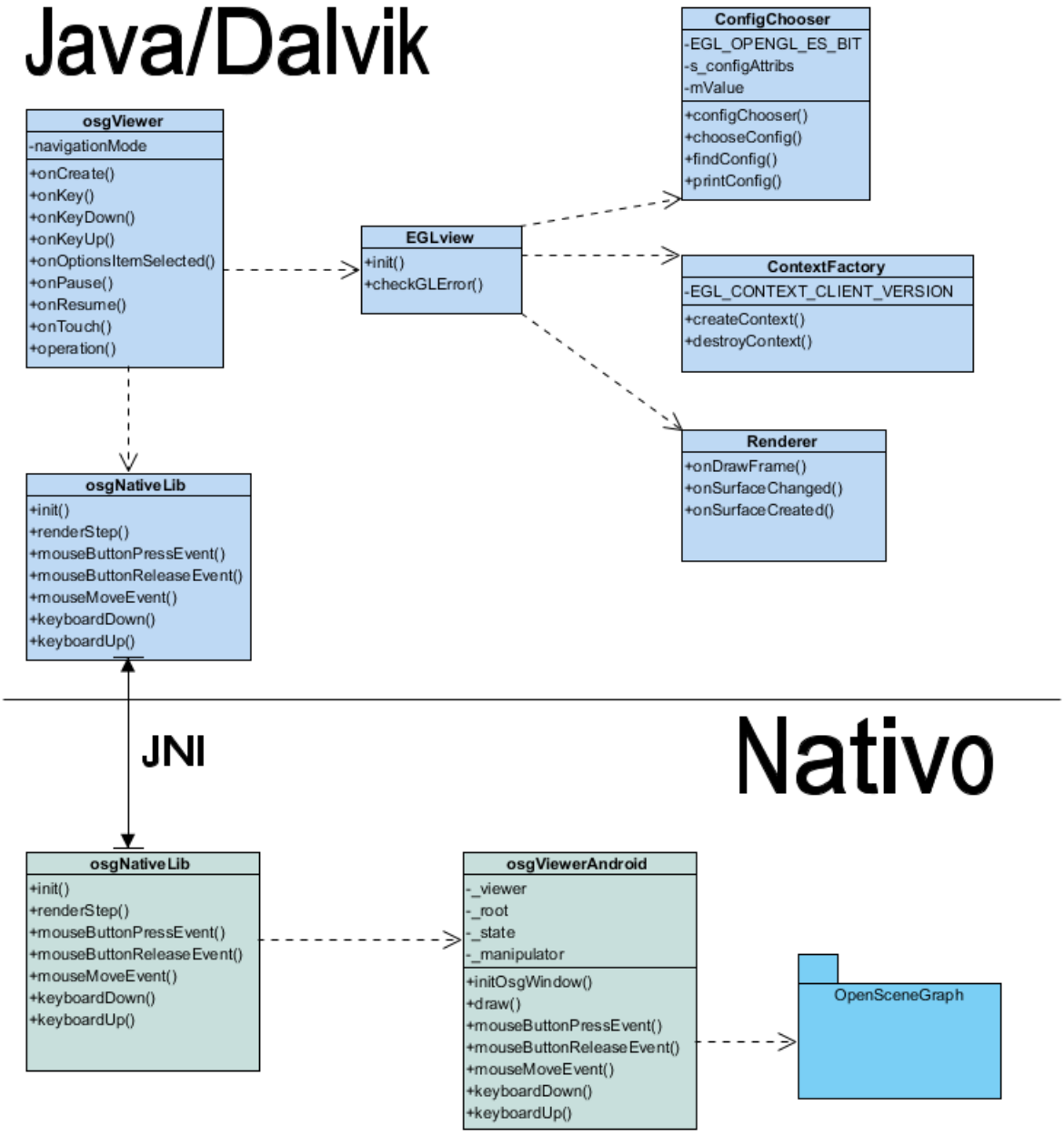

Figura 4.5: Diagrama de clases de la estructura de una aplicación OSG en Android.

4.3.2. Generación de un paquete de librerías third party para OSG

Para la creación de los programas de tests anteriores, es necesario que la librería OSG sea capaz de cargar y representar tipos de texturas y modelos complejos que no se encuentran soportados por la propia librería de OSG, para ello, la librería emplea una serie de plugins que se encargan de la gestión de dichos archivos, la carga o el guardado, y el plugin oportuno lo introduce como un elemento del grafo de escena.

Aunque muchos de los plugins y librerías de terceros se emplean para los formatos, algunas de las librerías son empleadas para opciones complejas alejadas de la gestión de un tipo de archivo. La librería Curl, por ejemplo, permite realizar peticiones Http para la transferencia de archivos. La librería Zlib por ejemplo da la opción de cargar modelos que se encuentren dentro de un archivo comprimido de forma directa y la librería Freetype se encarga de generar la representación de las fuentes que deseemos representar.

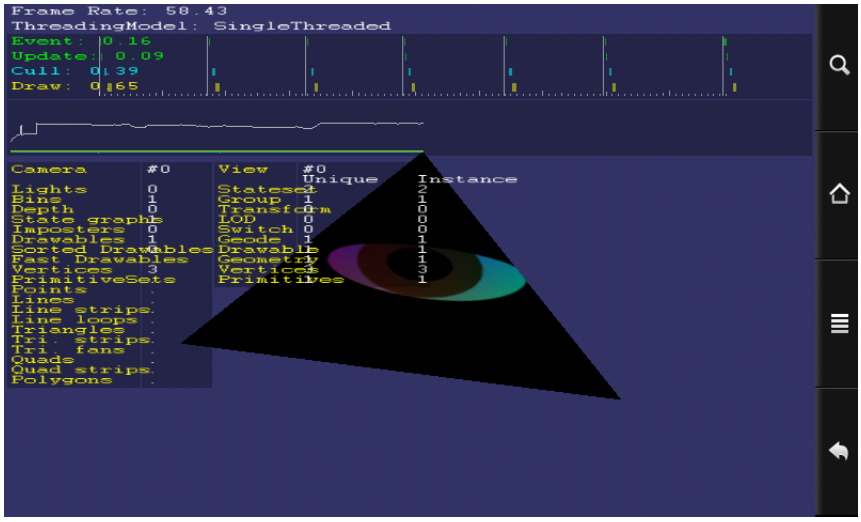

Figura 4.6: Primera prueba de funcionamiento del plugin de representación de texturas con formato PNG.

Sin estas librerías OSG seguiría siendo funcional, pero quedaría lastrado al tener que prescindir de muchas opciones muy empleadas en el desarrollo de aplicaciones. En la figura 4.6 se puede observar el uso de la librería libpng y su plugin en OSG para realizar la representación de la textura sobre el triángulo del primer programa OSG en Android.

Las librerías integradas en el paquete son:

- Curl - Librería para el envío y recepción de datos por red.
- Freetype - Librería para la representación de fuentes de texto.
- Gdal - Librería para la gestión de bases GIS.
- giflib - Librería para el uso de imágenes gif.
- libjpeg - Librería para el uso de imágenes jpeg.
- libpng - Librería para el uso de imágenes png.
- libtiff - Librería para el uso de imágenes tiff.
- zlib - Librería para el uso de archivos comprimidos.

El paquete generado es una estructura de carpetas que incluye directorios preparados para compilar, conjuntamente con OSG, los códigos fuente de las librerías integrándolas en la estructura de compilación de OSG Android. También se incluye un directorio que incluye las librerías compiladas listas para usarse en cualquier tipo de aplicación basada en Android. Las librerías han sido preparadas una a una con scripts NDK generados por los listados de archivos a compilar y con las opciones básicas.

4.3.3. Pruebas funcionales

Para comprobar las funcionalidades y el rendimiento de OSG, se han creado una serie de casos de control. Partimos de un visor básico que tiene activadas todas las librerías principales de OSG y un subconjunto de plugins que cubren: jpeg, png, Freetype, curl, osg, osg2. La estructura de la aplicación sigue las directrices básicas que han sido tratadas en el punto: 4.3.1. Debido a las diferencias existentes entre las dos API gráficas de OpenGL ES, se han creado dos visores para recoger las estadísticas en ambos casos. La estructura de la aplicación era idéntica en ambos casos.

Las diferencias que existen entre las dos aplicaciones son:

- La versión OpenGL ES con la que fue compilada la librería OSG.
- El uso de shaders para representar los modelos de las pruebas, esto es debido a las normas de funcionamiento que impone la versión 2.0 de la API gráfica.

Debido a que existen diferencias significativas si se usa un shader complejo, para este estudio se han empleado los shaders de referencia que tiene publicada la Kronos ARB, estos programas replican la funcionalidad de la tubería de procesado fija, computando la iluminación por vértices de forma análoga a la versión 1.0 de la API.

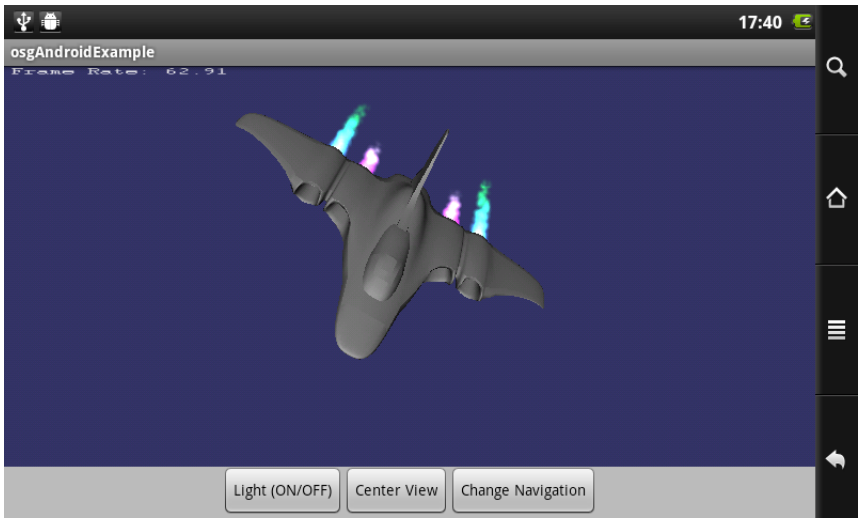

Figura 4.7: Prueba de funcionamiento del modelo ship de OSG con los efectos de partículas.

Los modelos de prueba a representar están divididos en dos grupos. Tenemos un conjunto de modelos básicos de bajo detalle poligonal y uno de modelos de alta resolución poligonal. La tabla 4.1 tiene las características de las escenas a representar. El conjunto de modelos de baja resolución proviene de los modelos que se usan en las aplicaciones de prueba de OSG que emplean una serie de características usadas de forma habitual. El conjunto de modelos de alta resolución está formado por los modelos generados con medición láser de Standford. El propósito de los tests de baja resolución es probar características generales de funcionamiento y en el caso de los tests de alta resolución probar el rendimiento puro sin optimizaciones. En la ilustración 4.7 se puede observar el renderizado tridimensional de un modelos (ship.osg) con un efecto de partículas en sus motores.

4.3.4. Prototipos de representación tridimensional de terreno

Para comprobar las posibilidades de su uso en el campo GIS y en aplicaciones comerciales, se han realizado tres prototipos funcionales. Los dos primeros, se basan en la representación de terrenos que han sido generados de forma estática y el tercero en

Baja resolución	Técnicas	Alta resolución	Número de polígonos
Cow	Environmental Mapping	Bunny	69.451
Cessna	Modelo comprimido	Horse	96.966
Cessna Fire	Partículas	Hand	654.666
Dumptruck	Modelo en red	Dragon	871.414
Fountain	Partículas	Happy	1.087.716
Lz	Nodos de Terreno, Billboards		
Morphing	Morphing		

Tabla 4.1: Listado de los diferentes modelos de prueba y sus características reseñables.

la generación de geometría tridimensional en tiempo de dibujado mediante el empleo de **programas shader**.

Los prototipos de representación con terrenos pregenerados, representan dos casos habituales de uso en el marco de aplicaciones. Por un lado, tenemos una representación de terreno gruesa a nivel planetario; el programa trabaja sobre una base de datos que contiene todo el planeta tierra. Ha sido generada mediante Virtual Planet Builder (VPB) de forma esférica con una profundidad de 8 niveles de detalle y un tamaño de 5Gbytes. Para su generación se han empleado las imágenes ortográficas, "Blue Marble" de la NASA y el mapa de alturas global Lansat con precisión de 5km/píxel.

El segundo ejemplo busca representar una sección menor de terreno de alta calidad. El ejemplo representa una zona planar limitada entre los términos de Anna y Enguera situada en la parte septentrional del Macizo del Caroig. La base de datos ha sido generada con siete niveles de detalle y un tamaño de 256Mbytes. Los datos empleados para su elaboración han sido obtenidos del Plan Nacional de Ortografía (PNOA). Estos prototipos han sido generados empleando la versión 1.x de OpenGL ES debido a que la pregeneración de la escena no incluye los shaders para representar el terreno por lo que deberíamos modificar el grafo de escena cada vez que un elemento fuese añadido al grafo de escena.

El prototipo de representación de terrenos con geometría generado en tiempo real, representa una sección detallada de terreno. Los datos de esta representación forman parte de los archivos LSAS de Standford y representan la zona del Gran Cañón en los Estados Unidos de América. En este prototipo se comprueban los diferentes rendimientos según el tipo de técnica que se emplea para generar la geometría. Más específicamente, se han comprobado las diferencias de rendimiento el uso, o no, de VertexBufferObjects, un modelo de transmisión de datos a la tarjeta gráfica de alto

rendimiento, y del uso, o no de lecturas sobre una textura que guarda un precalculo de normales.

4.3.5. Sistema de control de memoria

Como se ha comentado anteriormente, la cantidad de memoria asequible de estos dispositivos se convierten en el mayor de los cuellos de botella. Cuando tratamos con una aplicación que representa una cantidad de geometría tan grande como un planeta es necesario tener algún tipo de política de control de memoria.

Las escenas pregeneradas con la geometría de terrenos están formadas por una serie de nodos de nivel de detalle paginado (PagedLod). Estos nodos funcionan de forma similar a los nodos Lod, representan una versión más simple o más compleja dependiendo de la distancia al punto de visión.

En el caso de los Paged Lod, la particularidad consiste en que cada una de las versiones visibles solo se carga por demanda cuando es necesario para representarse. De esta manera se pueden hacer peticiones de zonas en diferentes archivos e incluso a través de una conexión de internet.

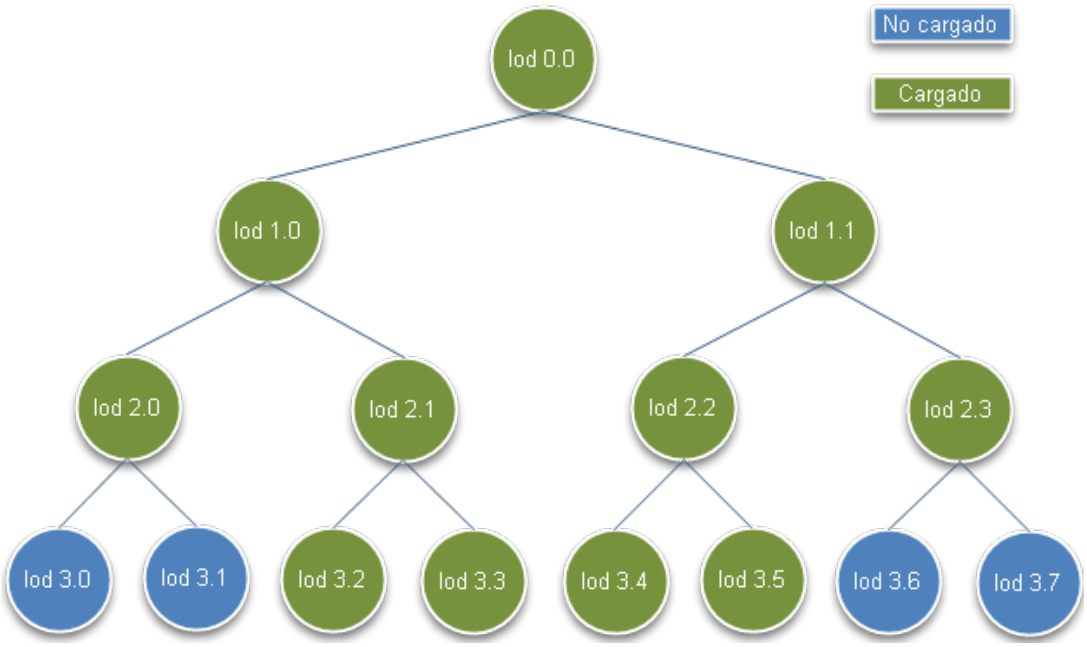

Figura 4.8: Esquema de los nodos cargados en un instante en una estructura similar a la presentada por las bases de datos de terreno.

En el esquema 4.8 se puede ver una representación de la carga de nodos en un instante determinado. Como se puede observar, todos los nodos que son requeridos para la visualización se encuentran cargados, pero también el nodo padre de cada uno de ellos. Esto supone que con cada nivel que se profundice, el consumo de la memoria se eleve de forma exponencial. Para ello se necesita implementar un sistema que controle el gasto de memoria y que pueda ser adaptable para cada dispositivo.

La solución desarrollada para corregir este problema, pasa por una revisión continua del grafo de escena. Esto es empleado para obtener una medición del consumo de memoria actual. Para asegurar su escalabilidad, la idea es que el algoritmo de balanceado del grafo se empleará cuando se sobrepasen una serie de limites en el consumo de memoria. Estos límites son valores que se pueden configurar desde la propia aplicación, lo que permitirá ajustar el nivel de detalle para cada dispositivo.

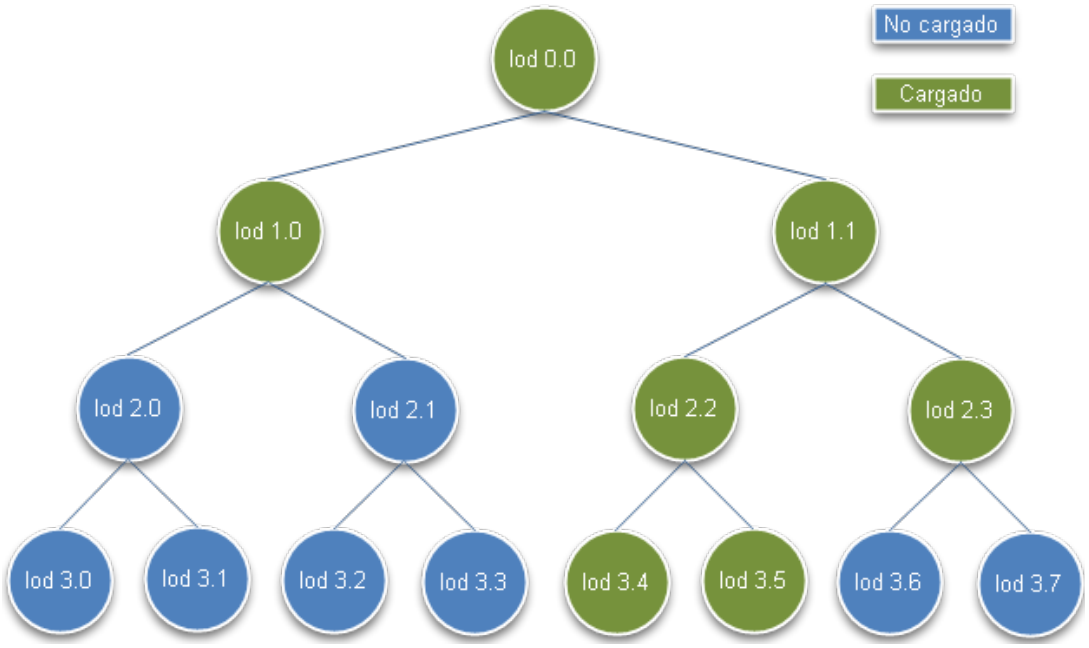

Figura 4.9: Esquema de la evolución de la carga de nodos debido al uso del sistema de control de memoria.

La técnica para balancear el grafo de escena, consiste en alterar la percepción de la distancia de los nodos Lod con el punto de visión. El algoritmo recorre el grafo de escena modificando la distancia mínima para que se carguen, o no, los nodos hijos de un PagedLod. De esta forma, como se puede ver en la ilustración: 4.9 cuando el algoritmo entra en acción provoca que los nodos padres consideren a sus hijos como no

representables ya que la distancia mínima ha sido aumentada. Esto penalizará mucho a los nodos laterales y alejados del punto de visión para poder ser representados, en cambio aquellos que se encuentren en la trayectoria del punto de visión, es decir, los más visibles e importantes representarán un nivel de detalle mucho mayor.

Figura 4.10: Representación visual donde el centro de la imagen posee un mayor nivel de detalle, mientras que a los lados se puede observar un menor nivel.

4.3.6. Problemas presentados durante el desarrollo de la segunda fase

Durante el transcurso de las primeras pruebas y la generación de los diversos programas, se encontró un nuevo problema con la librería. Cuando un programa intentaba emplear OSG para representar gráficos que utilizaran la API 2.0 de OpenGL ES, el programa provocaba una violación de que obligaba al sistema operativo a "matar" la aplicación. Cuando Android termina una aplicación de forma anormal, lanza a la salida de logs un volcado de pila con los datos de ejecución cuando se produjo la finalización. En el fragmento siguiente4.6, se puede observar un ejemplo de volcado de pila de Android.

```
Build fingerprint: 'acer/picasso_generic1/picasso:3.1/
    HMJ37/1309327721:user/release-keys'
pid: 4904, tid: 4913  >>> osg.AndroidExample <<<
signal 4 (SIGILL), code 1 (ILL_ILLOPC), fault addr
    8256b5be
 r0 00000120  r1 00000001  r2 00000000  r3 00000000
 r4 001ead00  r5 826d6cf0  r6 001eacd8  r7 826e12f4
 r8 b00133b8  r9 826dc000  10 006ee000  fp 82000000
 ip 00000000  sp 59217498  lr 8242b781  pc 8256b5be
    cpsr 20000030
 d0  497ffff8000fffff  d1  40dfae14497ffff0
 d2  bf800000c47a0000  d3  3f800000000000000
 d4  3f800000000000000  d5  3ff000003f800000
 d6  3f000000000000000  d7  00000000bf800000
 d8  0000000000000000  d9  0000000000000000
 d10 0000000000000000  d11 0000000000000000
 d12 0000000000000000  d13 0000000000000000
 d14 0000000000000000  d15 0000000000000000
scr 80000012
        #00  pc 0056b5be  /data/data/osg.
            AndroidExample/lib/libosgNativeLib.so (
            _ZN3osg7Matrixd12makeIdentityEv)
        #01  lr 8242b781  /data/data/osg.
            AndroidExample/lib/libosgNativeLib.so
```

```
libc base address: afc3d000
```

Código 4.6: Fragmento de un volcado de pila sobre un dispositivo Acer con Android 3.1.

La información más importante que se puede obtener del volcado de pila es, en primer lugar la señal que ha provocado la muerte de la aplicación, seguidamente los valores de registro, especialmente la dirección de la última instrucción ejecutada, así como las diferentes llamadas de la pila. Con esta información, se pudo determinar que en algún momento el código realizaba un salto a la instrucción de dirección 0x00000000. Esto ocurría siempre que se llegaba a una instrucción de OpenGL ES 2.0. Tras una revisión exhaustiva del código, se determinó que el problema ocurría porque no se llegaba a enlazar la librería OpenGL Es 2.0 con el código.

La solución del problema fue crear un segmento de código específico para la plataforma que realizaba la apertura, el enlace de la librería y finalmente su cierre cuando el programa terminaba. En un principio se intentó emplear la propia versión de enlace que se utiliza en Linux, pero su no funcionamiento obligó a emplear las funciones de la librería ldl tal y como aparecen en los ejemplos de utilización de Android. Seguidamente en: 4.7 se puede ver el código añadido para realizar la apertura de la librería OpenGL ES.

```
void* osg::getGLExtensionFuncPtr(const char *funcName)
{
#if defined(ANDROID)
    #if defined(OSG_GLES1_AVAILABLE)
        static void *handle = dlopen("libGLESv1_CM.so"
            , RTLD_NOW);
    #elif defined(OSG_GLES2_AVAILABLE)
        static void *handle = dlopen("libGLESv2.so",
            RTLD_NOW);
```

```
    #endif
    return dlsym(handle, funcName);
...
}
```

Código 4.7: Fragmento añadido a la función que se encarga de enlazar las funciones de OpenGL.

4.4. Creación de la documentación y programas de ejemplo

La última fase del trabajo ha consistido en dar soporte para la comunidad OSG mediante la elaboración de documentación en la pagina wiki de la librería. La documentación aportada describe los pasos para realizar la compilación de la librería OSG para emplearse con programas Android, así como comentar algunas problemáticas del estado actual de la compatibilización.

Adicionalmente, se han elaborado dos programas de ejemplos siguiendo la estructura mixta Java/Dalvik con código nativo OSG. El propósito de estos programas es mostrar a los desarrolladores interesados los pasos que tienen que tomar para integrar y comunicar la parte nativa de su aplicación con el framework Android de Dalvik.

Entre las características que muestran los ejemplos se encuentran:

- Configuración del manifiesto para bloquear la resolución.
- Configuración para instalación, por defecto, en la tarjeta SD externa.
- Menú de opciones en la aplicación.
- Diálogos personalizados.
- Empleo del teclado virtual para pasar órdenes a la parte nativa.
- Carga de modelos para las API OpenGL ES 1.0 y 2.0.
- Configuración de librerías básicas para ser usadas estáticamente mediante el uso de macros.

- Redirección de la salida estándar de errores hacia el Logcat de Android.

5

Resultados

En este apartado, se detallarán los resultados de los estudios realizados sobre la portabilización de la libraría al sistema operativo Android. En primer lugar se detallarán las condiciones de las pruebas y, seguidamente, se expondrán los resultados de los test de funcionamiento.

5.1. Características de los dispositivos de pruebas

Durante este estudio, se han empleado tres modelos físicos diferentes para obtener unos datos fiables. Los modelos empleados en este estudio son:

- HTC Nexus
- Archos 70i tablet
- Samsung Galaxy S I-9000

En la tabla 5.1 se encuentran detallas las características de procesador, resolución y memoria de los diferentes dispositivos.

La versión del sistema operativo empleada durante las pruebas siguientes es la versión 2.2, debido a su mayor estabilidad en comparación con la versión 2.1. Aunque durante el transcurso de este proyecto se ha comprobado el funcionamiento por

Modelo	HTC	Samsung Galaxy S I-9000	Archos 70i Tablet
Procesador	Qualcomm QSD8250 Adreno 200	1Ghz ARM Cortex-A8 PowerVR SGX540	1Ghz ARM Cortex-A8
Resolución	480x800	480x800	800x480
Memoria	512Mbytes	512Mbytes	256Mbytes

Tabla 5.1: Características técnicas de los dispositivos publicadas por sus empresas

una serie de voluntarios de la comunidad OSG en diferentes modelos y versiones, los resultados aquí presentados son únicamente de los dispositivos con los que se ha podido trabajar directamente.

De acuerdo a las pruebas realizados por los usuarios de la comunidad OSG, se puede afirmar que la compatibilización funciona correctamente con las versiones 2.3 y 3.1 con lo cual, se puede afirmar que la solución presentada en este trabajo ha alcanzado un nivel aceptable de funcionamiento de cara al desarrollo de futuras aplicaciones.

5.2. Resultados modelos de baja resolución

Los tests de funcionamiento son una muestra de control de varias técnicas empleadas de forma común en los gráficos. Las técnicas que han funcionado correctamente empleando la API OpenGL ES 1.0 son las siguientes:

- Carga de ficheros comprimidos
- Partículas (ilustración: 5.1)
- Carga de ficheros en red
- Nodos Terrain
- Billboards
- Morphing

La característica: "Environmental Mapping" no ha funcionado correctamente. Esto es debido a que no se encuentra soportada en OpenGL ES como una característica principal de la API, está soportada como una extensión opcional y todavía no ha sido integrada, en esta forma, en la librería OSG.

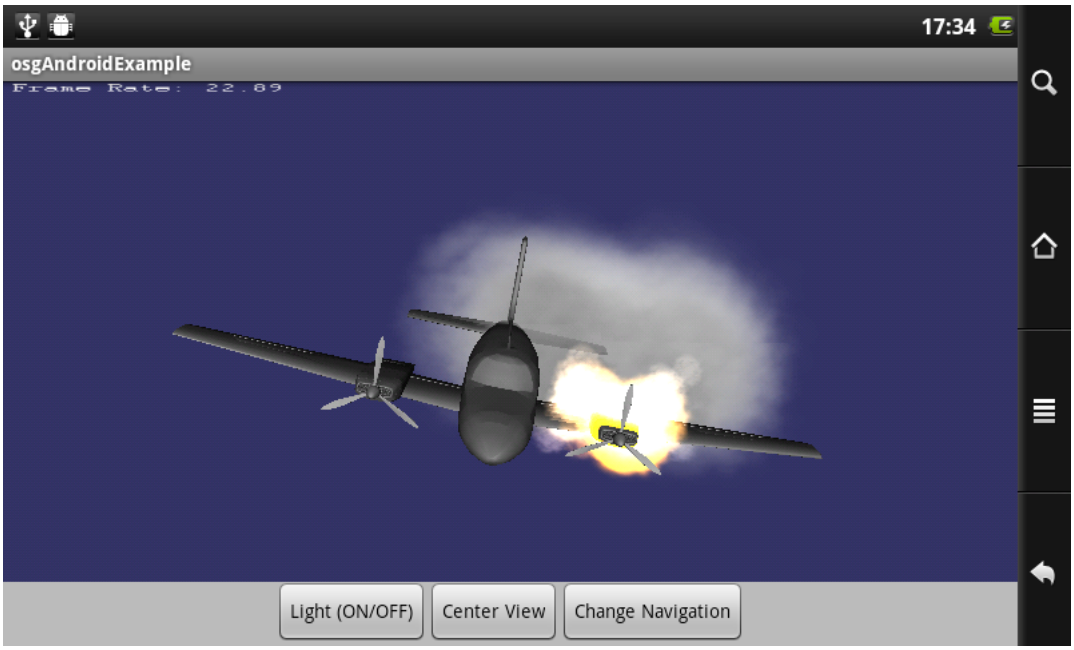

Figura 5.1: Aplicación OSG sobre Android representando el modelo "cessnafire.osg" que emplea un fuego generado con la técnica de partículas.

El rendimiento empleando la API 1.0 se puede ver en la tabla: **??**. La tasa de frames por segundo es aceptable para la mayor parte de casos. Únicamente se ve un repunte negativo en la técnica de morphing, debido a la cantidad de cálculos en coma flotante que requiere. Se puede notar también como, debido limitaciones impuestas por el driver, el móvil Samsung nunca supera un umbral de frames por segundo, esto es algo habitual para reducir el consumo de batería en estos dispositivos.

En el caso de la API OpenGL ES 2.0, hay que recordar que la situación cambia debido al uso de Shaders. La librería OSG emplea un generador de Shaders para emular la tubería de renderizado fijo, sin embargo, los shaders pregenerados provocan un error de compilación en un dispositivo real a pesar de cumplir el estándar GLSL. Esto se debe a que, en OpenGL ES 2.0, existe una modificación al estándar que no está reflejada en la especificación GLSL. En el libro [GSM08] se comenta este cambio que obliga, por norma, a emplear un valor de precisión para toda variable uniforme de un pro-

Modelo	HTC	Samsung Galaxy S I-9000	Archos
Cow	20.3 fps	52.8 fps	64.8 fps
Cessna	4.3 fps	53.0 fps	59.7 fps
Cessna Fire	20.2 fps	53.5 fps	53.8 fps
Dumptruck	3.9 fps	55.3 fps	68.0 fps
Fountain	45.8 fps	53.7 fps	40.0 fps
Lz	42.2 fps	51.0 fps	62.9 fps
Morphing	4.9 fps	20.9 fps	17.6 fps

Tabla 5.2: Estadísticas de frames por segundo de los modelos testeados sobre Android con OpenGL ES 1.0

grama shader. Actualmente los shaders pregenerados no incluyen esta característica y por ello no pueden ser utilizados. Aun así, es posible emplear OSG de forma normal siempre que el programador genere sus propios shaders de forma correcta y no dependa de los autogenerados por OSG.En la figura: 5.2 vemos el resultado de emplear un shader Cartoon sobre el modelo de la vaca de OSG. Como se puede ver se realiza el dibujado correctamente empleando las instrucciones del shader.

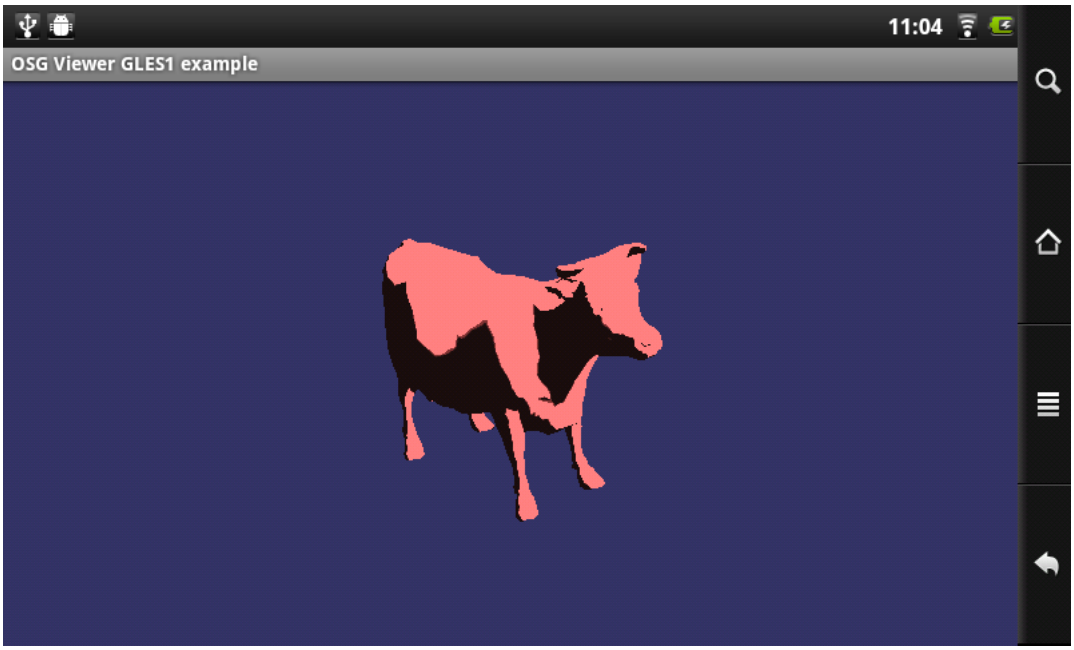

Figura 5.2: Aplicación OSG sobre android representando el modelo "cow.osg" sobre OpenGL ES 2.0 empleando un shader de tipo Cartoon.

En la tabla: 5.2 se encuentran las estadísticas de frames por segundo de esta prueba, Si los comparamos con los resultados de la tabla: 5.3, el rendimiento alcanzado en ambos casos es similar.

Modelo	HTC	Samsung Galaxy S I-9000	Archos
Cow	21.3 fps	53.8 fps	65.5 fps
Cessna	4.7 fps	54.0 fps	60.5 fps
Cessna Fire	18.6 fps	53.5 fps	54.7 fps
Dumptruck	4.7 fps	55.3 fps	67.1 fps
Fountain	47.3 fps	53.7 fps	41.0 fps
Lz	48.3 fps	53.7 fps	72.7 fps
Morphing	5.0 fps	21.1 fps	16.8 fps

Tabla 5.3: Estadísticas de frames por segundo de los modelos básicos sobre Android con OpenGL ES 2.0

5.3. Resultados modelos de alta resolución

Ha resultado imposible hacer un estudio de rendimiento con la versión 1.X de OpenGL ES. La representación de modelos con una geometría monolítica sin el uso de VertexBufferObjects(VBO) únicamente ha llegado a funcionar con el modelo de alta resolución más pequeño, "Bunny". La tasa de frames obtenida se presenta en la tabla: 5.4. Como se puede ver, el HTC ha sido incapaz de ejecutarlo lanzando una excepción del driver gráfico. La tableta Archos y el móvil Samsung han sido capaces de representarlo con una tasa excepcionalmente baja y el resto de modelos han provocado, de la misma manera una excepción en los driver gráficos de ambos.

Modelo	HTC Nexus	Archos 70i	Samsung Galaxy S I-9000
Bunny	Error driver	2fps	3fps

Tabla 5.4: Tasa de frames por segundo de los modelos de alta resolución sobre OpenGL ES 1.X

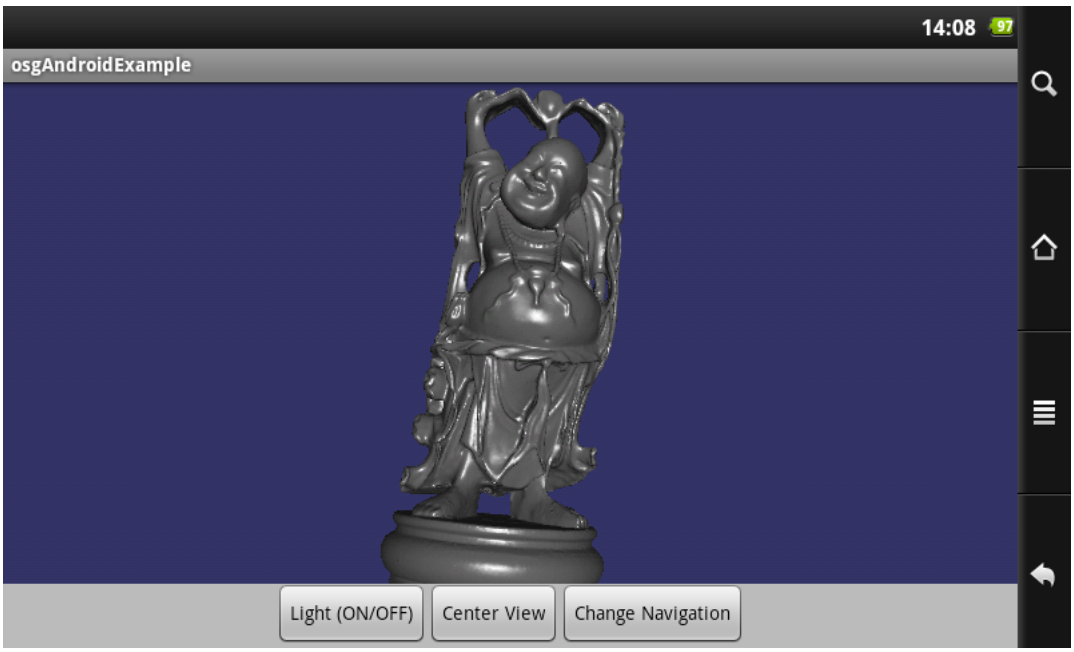

Figura 5.3: Aplicación OSG en Androi representando el modelo de alta resolución "El Budha feliz", que está formado por más de un millón de polígonos, de forma monolítica sin optimizaciones.

La prueba realizada sobre OpenGL ES 2.0 ha obtenido unos mejores resultados. Se han conseguido representar todos los modelos. Esto supone en el caso del modelo "Budha feliz" la ocupación de 40Mbytes de memoria, una ocupación muy alta de memoria que, en muchos modelos, situaría a la aplicación en el borde de ser cerrada por el sistema operativo por consumo excesivo de memoria. Aun así, como se puede ver en la figura: 5.3 es posible mover un modelo monolítico, sin optimizaciones geométricas o de nivel de detalle, que supera el millón de polígonos empleando únicamente la potencia pura de los dispositivos actuales. Sin embargo, como se puede observar en la tabla: 5.5 el rendimiento resultante todavía es muy bajo, por ello es aconsejable el uso de las optimizaciones y particiones geométricas para obtener una tasa de dibujado buena en las escenas complejas.

Modelo	HTC	Samsung Galaxy S I-9000	Archos
Bunny	12.1 fps	34.7 fps	16.7 fps
Horse	6.9 fps	19.5 fps	11.4 fps
Hand	1.5 fps	9.5 fps	7.7 fps
Dragon	1.0 fps	11.7 fps	7.1 fps
Happy	0.7 fps	8.7 fps	6.7 fps

Tabla 5.5: Estadísticas de frames por segundo de los modelos de alta resolución sobre Android con OpenGL ES 2.0

5.4. Resultados de la representación de terrenos precalculada

Los test sobre terrenos, empleando bases de datos pregeneradas, han resultado contundentes. Cuando no empleábamos el regulador de memoria, los programas de prueba aumentaban rápidamente su consumo de memoria hasta llegar a los límites que ofrecían los dispositivos. Esto obligaba al sistema operativo a cerrarlas por consumo excesivo de memoria aunque las escenas empleaban los nodos PagedLod para realizar las cargas de secciones por demanda, el consumo de memoria crecía exponencialmente.

Para solucionar este problema se había creado un regulador de la carga del grafo de escena. El regulador permite configurarse con una serie de parámetros de funcionamiento. Período de muestreo de la memoria, tamaño máximo de nodo y los límites superior inferior de la ocupación deseada. Esto permite escalar la optimización según el modelo exacto que lo ejecuta.

Parámetro	Valor
Período de muestreo de memoria	5 fps
Tamaño máximo de nodo	0.5 Mbytes
Ocupación mínima	20 Mbytes
Ocupación máxima	30 Mbytes

Tabla 5.6: Parámetros del regulador de nodos para terrenos pregenerados.

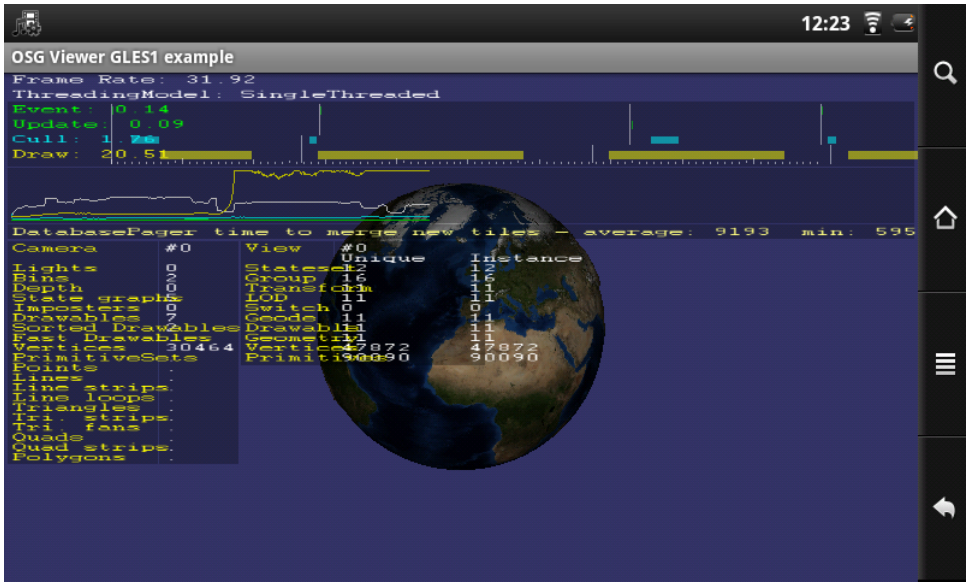

Figura 5.4: Imagen del programa de representación de terrenos pregenerados. En la imagen se puede ver una imagen a distancia de la tierra mientras se representan las estadísticas de la aplicación

Para emplear el regulador, se han fijado los parámetros tal y como se muestran en la tabla: 5.6. Los resultados en el funcionamiento y la estabilidad del programa han sido satisfactorios y no se ha generado ningún error por falta de memoria durante los vuelos de prueba. En la figura: 5.4 se puede ver representado el consumo de memoria durante un vuelo de un minuto en el programa de representación de terreno planetario. Como se puede observar, cuando el punto de vista se acerca lo suficiente y expande los nodos descendientes, se produce un repunte de uso de memoria por la carga de nuevos nodos a representar y la no eliminación de los padre que no se representan en ese momento. Esta explosión en el gasto de memoria termina sobrepasando el límite superior fijado durante la prueba, esto hace que reaccione el regulador de memoria y reajuste las distancias de visualización de cada nodo Lod. Esto obliga al grafo de escena a revisar el nuevo estado de la escena y eliminar aquellos nodos que no se emplean en el dibujado y no tienen descendientes liberando, en este proceso, memoria ocupada. Como consecuencia de este método, **la expansión de nodos será más profunda en la zona cercana al punto de visión del usuario, dejando el resto de zonas con un menor nivel de detalle**.

Como se puede ver en la tabla: 5.7 el rendimiento del programa con un terreno reducido de alta calidad es superior al programa con un terreno de grandes dimensiones y baja calidad. Aún así, el rendimiento en ambos es aceptable para llegar a funcionar

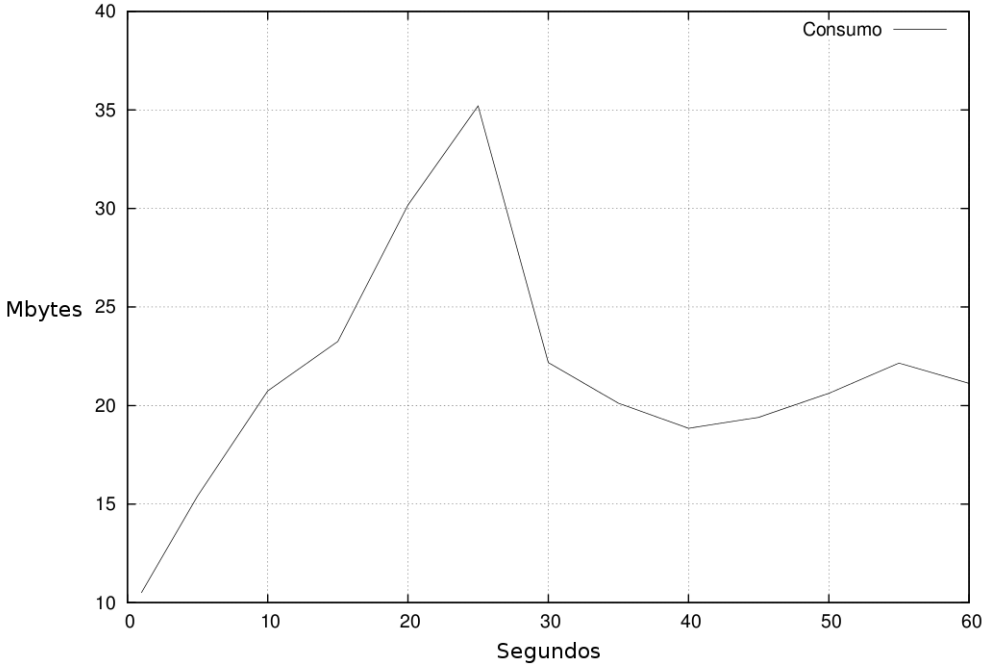

Figura 5.5: Estadísticas de consumo de memoria de nuestro programa con base de datos pregenerada. Las gráfica muestra la variación de la ocupación durante un minuto.

en tiempo interactivo, aunque con el modelo HTC se puede ver que el rendimiento termina siendo bajo. La imagen: 5.4 ha sido capturada sobre el programa que representa el modelo del planeta tierra. La imagen: 5.6 viene del programa que representa el área cercana al pueblo Anna, situado en La canal de Navarrés.

Parámetro	HTC	Samsung Galaxy S I-9000	Archos
Modelo planetario tierra	8fps	35.2fps	17.3fps
Sección terreno Anna	12fps	42.4fps	23.7fps

Tabla 5.7: Estadísticas de los ejemplos de representación de terrenos.

Figura 5.6: Imagen del programa de representación de terrenos pregenerados. En la imagen se puede ver la zona de Anna en La canal de Navarrés

5.5. Resultados de la representación de terrenos generados en tiempo de dibujado

Los tests de terrenos cuya geometría se ha generado en tiempo de renderizado nos muestran claramente la posibilidad de emplearlo para programas reales. Como se ve en las estadísticas de todos los modelos, siempre y cuando el tamaño de la tile de terreno es reducido es posible tener cifras superiores a 24fps. Las gráficas siguientes muestran los resultados obtenidos, de ellas, se infieren las siguientes conclusiones:

- El uso de VBO para la representación de terreno genera un aumento de rendimiento importante
- El uso de las normales precalculadas es más veloz que el calculo en tiempo real.
- En los actuales dispositivos existe un cuello de botella muy importante en el acceso a la memoria, dependiendo de la cantidad de lecturas sobre la memoria, será o no aconsejable el uso de normales precalculadas.

Como se puede ver claramente en las gráficas, el uso de VBO mejora el rendimiento. En el caso del Archos el rendimiento se ve beneficiado por su uso. En el caso de

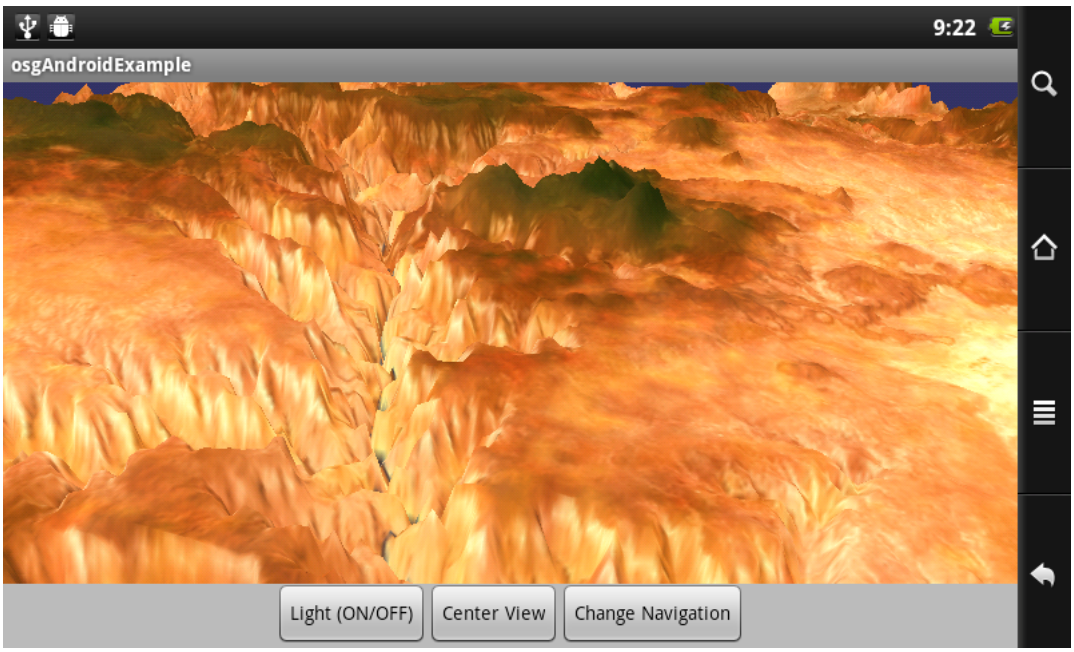

Figura 5.7: Imagen del programa de prueba de renderizado de terreno generado en tiempo de dibujado representando "El Gran Cañón".

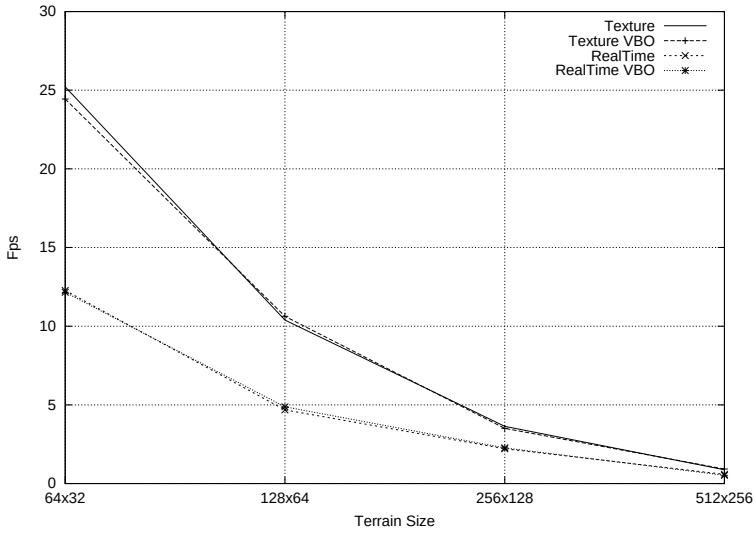

Figura 5.8: Comparativa de rendimiento de las diferentes combinaciones de uso de los VBO y texturas con cálculos de normales precalculadas dependiendo del tamaño del terreno en el móvil Htc

los Samsung, se ve claramente que cuando la geometría aumenta, se crea una clara diferencia entre el uso o no de VBO. En la gráfica: 5.8 el móvil HTC se desmarca de esa tendencia, sin embargo es posible que esto sea debido a un problema del driver gráfico.

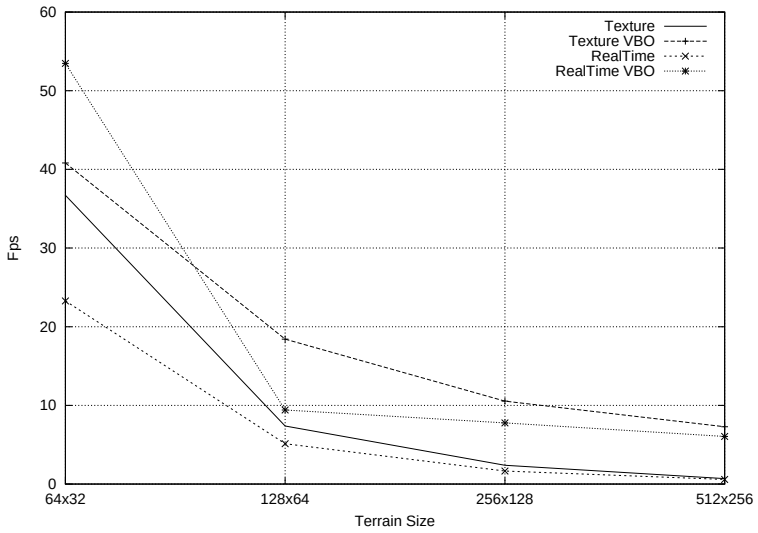

Figura 5.9: Comparativa de rendimiento de las diferentes combinaciones de uso de los VBO y texturas con cálculos de normales precalculadas dependiendo del tamaño del terreno en la tableta Archos

Con respecto al uso de texturas con las normales precalculadas se puede ver en todos los modelos un claro aumento en el rendimiento, sin embargo, como se puede notar en la gráfica: 5.9 la tableta Archos, dependiendo de la cantidad de accesos a textura que se realicen termina siendo contraproducente y más óptimo el cálculo puro de las normales. Esto pone de manifiesto el punto tres, algunas de las memorias empleadas en estos dispositivos tienen unos tiempos de latencia alta que, al emplear técnicas que requieran lecturas de textura, provocan una disminución importante en el rendimiento. Así pues, habrá situaciones en las que debido al alto número de accesos a memoria, será conveniente realizar versiones que empleen el cálculo matemático. Si bien, este tipo de limitaciones están muy ligadas a determinados dispositivos de hardware y aunque no pueden preveerse previamente, se pueden realizar versiones de un programa para los dispositivos que tengan este problema. Esto se podrá realizar de forma transparente al usuario gracias a los mecanismos actuales de versiones específi-

cas en Android, el desarrollador puede publicar varias versiones especificas bajo un mismo nombre y la aplicación de descarga selecciona automáticamente la versión más ajustada según los parámetros que establezca el desarrollador en el manifiesto.

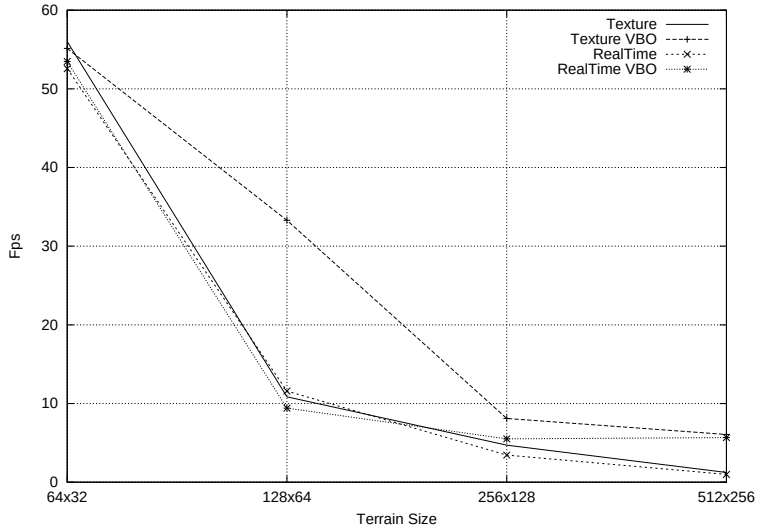

Figura 5.10: Comparativa de rendimiento de las diferentes combinaciones de uso de los VBO y texturas con cálculos de normales precalculadas dependiendo del tamaño del terreno en el móvil Samsung Galaxy S I-9000

6

Conclusiones y trabajo futuro

En esta sección repasaremos los puntos más importantes para llegar a las conclusiones finales y las posibles líneas de trabajo futuro.

El objetivo fundamental de este proyecto ha sido la adaptación de la librería OpenSceneGraph (OSG) a Android. Se ha llevado a cabo para gestionar y optimizar la representación de escenas tridimensionales, ya que, como grafo de escena que es, permitirá escalar la complejidad de una forma más sencilla para obtener tasas interactivas de renderizado.

Durante todo el proyecto, se ha buscado encontrar la manera de integrar los cambios para la nueva plataforma del modo menos intrusivo posible. Para ello se ha realizado un estudio exhaustivo del sistema operativo Android y de las diferentes dependencias de la librería OSG.

Como resultado se han realizado una serie de cambios a los archivos de código fuente de la librería que arreglaban los problemas provocados por las diferencias entre una distribución Linux y Android. Estos cambios se han presentado a la comunidad OSG y han sido integrados en la rama principal de desarrollo.

Además de los cambios para eliminar incompatibilidades, se ha tenido que generar una serie de scripts de compilación NDK Android para OSG. Debido a que la librería emplea scripts de CMake para gestionar la cadena de compilación y con el objetivo de no añadir complejidad al mantenimiento de los scripts de compilación, se

han implementado una serie de funciones y macros en CMake que permiten generar los archivos de compilación NDK Android desde los scripts CMake originales. Estos cambios sobre los ficheros CMake también han sido integrados en la rama principal de desarrollo.

OSG es una librería que, opcionalmente, depende de una serie de librerías de terceros para gestionar la apertura y el guardado de determinados tipos de fichero. Para poder probar todas las funcionalidades, se ha realizado la compatibilización de una serie de librerías básicas (libJpeg, libPng, libtiff, Freetype, Curl, Gdal, etc) creando, con ello, un paquete de librerías externas para emplear en Android que ha sido publicado en la página de la librería OSG.

Finalmente se han presentado una serie de programas de prueba, así como su estructura, que permiten comprobar las distintas características del grafo de escena. Estos programas han servido para estudiar las posibilidades de optimización y escalado de escenas empleando OSG.

También se ha presentado un algoritmo de balanceado del grafo de escena. Este algoritmo carga y descarga nodos dependiendo de la ocupación actual de la memoria por parte del programa.

De acuerdo a los datos obtenidos en los resultados, todas las características soportadas por la API funcionan correctamente. Las únicas que no están disponibles son aquellas que no están soportadas actualmente en las API de OpenGL ES. Las pruebas de representación de terreno, muestran la idoneidad de emplear OSG para optimizar y escalar la representación de la escena dentro de los límites de nuestra plataforma objetivo.

Adicionalmente, se ha presentado a la comunidad OSG una serie de aplicaciones Android que sirven de ejemplo para integrar OSG en el desarrollo de una aplicación, así como la documentación oportuna para realizar una compilación de OSG para Android.

Las líneas de trabajo futuro pasan por mejorar la integración de OSG con la plataforma Android. Oficialmente, en la última versión todavía no existe una implementación dinámica de la librería STL estándar. Esto supone un coste espacial prohibitivo, cada componente de OSG tendría que contener parte de la STL y muchas partes de código quedarían replicadas dentro de la librería con el incremento de tamaño que supondría para los ejecutables.

Es necesario también incluir un sistema de carga de librerías dinámicas siguiendo los estándares de Android. A diferencia de Linux, la instalación de las librerías no se

realiza sobre el sistema. Dado que algunas librerías basadas en OSG no permiten la compilación estática, avanzar en esta línea de trabajo supondría una mejora importante.

Otra vertiente de trabajo pasa por modificar algunas partes de código de la librería OSG para permitir la representación de algunos elementos gráficos sin emplear la tubería fija. Al contrario que las versiones de sobremesa de la API gráfica, las versiones para dispositivos embebidos o tienen tubería fija o tienen tubería programable. Por ejemplo, sería muy interesante trabajar en la representación de las estadísticas OSG. Aunque las estadísticas se pueden obtener mediante llamadas de código, no se pueden representar como si ocurre cuando se puede emplear la tubería fija.

Un punto que se ha quedado sin desarrollar en este trabajo ha sido la programación de Actividades totalmente nativas mediante la "NativeActivity". Aunque en el trabajo se ha incluido una estructura teórica de aplicación y se han comentado las posibilidades que ofrecen, no se ha publicado ninguno de los ejemplos, ya que no se ha podido disponer de un dispositivo compatible para las pruebas de funcionamiento.

Finalmente, entrando en la representación de terreno, este proyecto solo ha cubierto una parte muy reducida y que debería ser ampliada estudiando más en profundidad la representación de terrenos con las nuevas capacidades de estos dispositivos. En esta tesitura sería interesante estudiar la implementación de librerías basadas en OSG que se están empleando actualmente en aplicaciones GIS como osgVP y osgEarth.

7

Agradecimientos

Deseo agradecer a mi director de proyecto, Javier Lluch, por haber hecho posible este trabajo. También he de agradecer especialmente a mi codirector de proyecto, Jordi Torres, que me ha ayudado continuamente a centrar los objetivos de este proyecto. Quiero hacer una mención especial a Rafa Gaitán cuya ayuda y conocimientos sobre la librería OSG han sido muy importantes para agilizar algunas partes de este proyecto.

Por otra parte quiero agradecer a mis compañeros del Instituto AI2. Durante nueve meses se han convertido en un entorno familiar y profesional donde he podido trabajar y aprender junto a un grupo de trabajo con una gran cantidad de desarrollos a sus espaldas. Especialmente quiero agradecer la compañía, los consejos y las risas de Leo Salom, María Ten y Jesús Zarzoso.

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.

[Biz10] Andrea Bizzotto. *Touchscreen-Based Used Interaction*, 2010.

[Cam06] Javier Lluch Rafa Gaitán Miguel Escrivá Emilio Camahort. Multiresolution 3d rendering on mobile devices. 2006.

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

[Cat10] Ken Catterall. *Migration to OpenGL ES 2.0*, 2010.

[CG02] Chun-Fa Chang and Shyh-Haur Ger. Enhancing 3d graphics on mobile devices by image-based rendering. In *IEEE Third Pacific-Rim Conference on Multimedia (PCM 2002)*, 2002.

[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*, 2005.

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

[ESC00] Jihad El-Sana and Yi-Jen Chiang. External memory view-dependent simplification. In *EuroGraphics '2000*, volume 19, 2000.

[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*, 2009.

[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.

[goo11d] *Google Stadistics - http://developer.android.com/resources/dashboard/platform-versions.html*, 2011.

[GSM08] Dan Ginsburg, Dave Shreiner, and Aaftab Munshi. *OpenGL ES 2.0 Programming Guide*, 2008.

[HW09] Bin HU and Jingnong WENG. Component-based virtual globe visualization engine design. *Computational Intelligence and Software Engineering*, 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] Yury Kryarchko. *Using Vertex Texture Displacement for Realistic Water Rendering*, 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.

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

[Mik10] Morten S. Mikkelsen. *Bump Mapping Unparametrized Surfaces on the GPU*, 2010.

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

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

[SP09] Marjan Sterk and Mariano Agustín Cecowski Palacio. Virtual globe on the android – remote vs. local rendering. *Sixth International Conference on Information Technology: New Generations*, 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.

[USK06] Tamás Umenhoffer and László Szirmay-Kalos. *Displacement Mapping on the GPU - State of the Art*, 2006.