Diseño e implementación de un sistema GIS 3D para plataformas móviles

José Miguel Santana Núñez

Tutores:

Agustín Trujillo Pino

Modesto Castrillón Santana

Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería - ULPGC

Tabla de contenido

1. Introducción

Ante un panorama de clara proliferación de aplicaciones para móviles, el software de representación de terreno pretenden ser una apuesta fuerte, dentro de los Sistemas de Información Geográfica 3D de código abierto, que abarque variadas utilidades sectoriales.

Este tipo de aplicaciones, también llamadas globos virtuales permiten tener a nuestra disposición un modelo virtual del planeta Tierra. Los usuarios pueden interactuar con el globo virtual rotándolo y ampliando zonas. De esta forma, se puede "volar" alrededor del modelo del planeta y acercarnos a cualquier zona para ver el terreno en más detalle. El terreno mostrará en su superficie imágenes que pueden corresponderse tanto con la textura real del mismo, como otro tipo de datos más específicos como mapas o callejeros, tal como se puede observar en la figura 1.

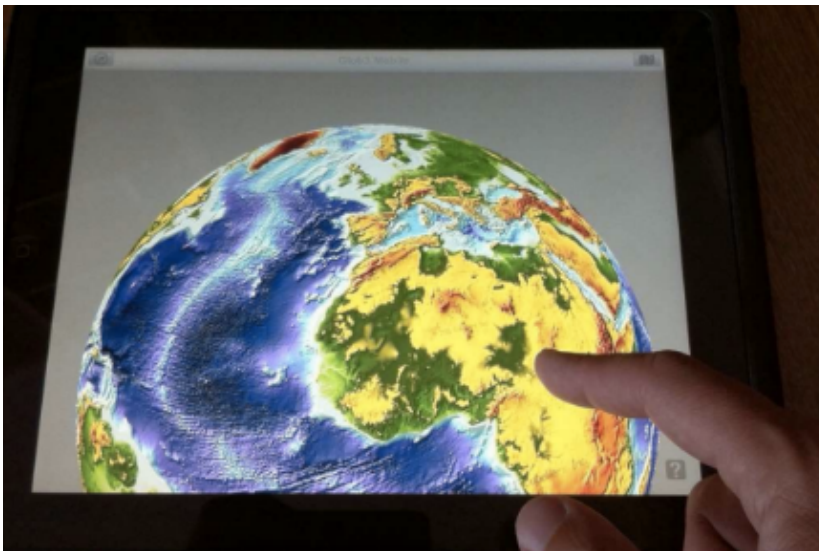

Figura 1 - Representación 3D del planeta en un tablet

Además se pueden superponer topónimos y fronteras, entre otros datos geográficos, a las imágenes. Se debe permitir visualizar imágenes de otras fuentes en Internet que usen protocolos y estándares conocidos como el protocolo del Open Geospatial Consortium Web Map Service (WMS). Adicionalmente existe multitud de ampliaciones que aumentan su funcionalidad, como por ejemplo, poder medir distancias u obtener datos de posición desde un GPS.

En los dispositivos móviles de hoy en día, ya las posibilidades tecnológicas de los mismos no representan una excusa. Se disponen incluso de dos o más procesadores junto con tarjetas gráficas de gran capacidad, con lo cual es perfectamente viable poder ejecutar una aplicación que haga un uso intensivo del hardware gráfico. En la figura 2 vemos varios ejemplos de globos virtuales funcionando sobre teléfonos inteligentes (smartphones) y tablets.

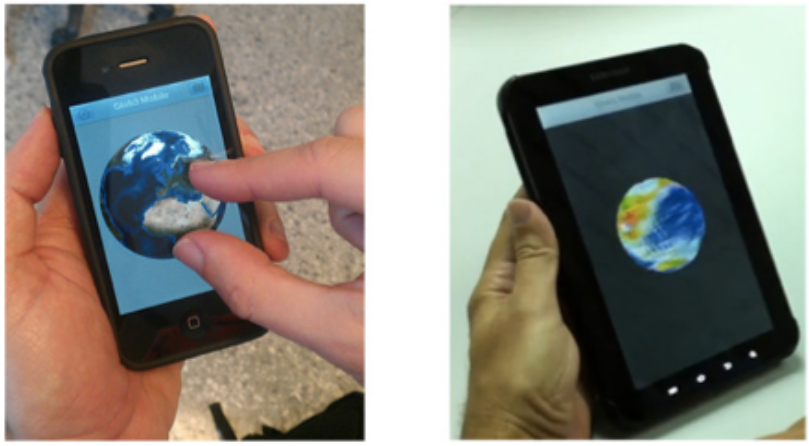

Figura 2 - Uso de dispositivos móviles con globos virtuales

Sin embargo, el modelo o arquitectura de trabajo para diseñar nuevas aplicaciones, programar y aumentar las funcionalidades de estos dispositivos es esencialmente distinta a la de ordenadores de escritorio. Entre ellos encontramos algoritmos y métodos que funcionan para la arquitectura clásica de un ordenador personal, pero dejan de ser portables a estos dispositivos móviles. Como ejemplo, todo lo relativo al manejo de la memoria, de la tarjeta gráfica, de la visualización, de la interacción con el dispositivo, etc. Basta con mencionar que las pantallas de los móviles, la mayoría de tipo táctil, son ya casi, el único elemento de interacción con estos dispositivos.

Este proyecto trata del desarrollo de un globo virtual de código abierto para dispositivos móviles (principalmente con sistema operativo Apple-iOS y Android). Por tanto, nos centraremos en el diseño e implementación del globo virtual multiplataforma, como elemento esencial que da soporte al SIG 3D, potenciando una arquitectura que permita la programación de nuevas funcionalidades al globo.

2. Estado del arte

Los globos virtuales han tenido un impacto positivo en muchas empresas tecnológicas de España y el resto del mundo, tanto en la organización económica, política y en relación con la imagen corporativa. También ha tenido un impacto positivo en las industrias relacionadas con la gestión de datos geográficos. Se han percibido cambios en las estrategias de negocio en la industria nacional sobre las indicaciones geográficas como resultado de los globos virtuales. Además, la tecnología de las aplicaciones geográficas también ha tenido un impacto positivo en la educación.

Dentro del terreno del software de visualización 3D en formato de software libre podemos encontrar muchas aplicaciones disponibles que implementan diferentes algoritmos de nivel de detalle (Level of Detail o LOD), gestión del terreno y otros aspectos de este tipo de navegadores. Tal y como ya hemos comentado, el objetivo de estas aplicaciones ha sido siempre la navegación en 3D por terrenos geográficos virtuales. Si bien sus comienzos estuvieron orientados a los juegos de ordenador, muchos de ellos encontraron una gran utilidad en la navegación virtual por zonas geográficas concretas. Entre estas aplicaciones podríamos destacar las siguientes:

- OssimPlanet [1] es un módulo desarrollado en C++, construido sobre el software Ossim, que permite la visualización geoespacial realista en 3D, a partir de un modelo digital de terreno en formato SRTM, y la inclusión de capas remotas a través del protocolo WMS.
- WorldWind [2] es un globo virtual desarrollado por la agencia espacial NASA, originalmente escrito en Java, que ofrece fotografía satélite y aérea y mapas topográficos, así como inclusión de elementos 3D accesibles desde servidores remotos.
- Virtual Terrain Project (VTP) [3] es un proyecto software que pretende promover la creación de herramientas para la construcción de cualquier parte del mundo real en forma digital, interactiva y tridimensional, relacionando tecnologías de áreas diversas como diseño asistidos (CAD), sistemas de información geográficos (SIG), simulación visual y teledetección.

3. Contexto y motivación del proyecto

3.1. Situación actual del mercado de aplicaciones móviles

A la hora de comenzar este proyecto nos encontramos ante un panorama de cambio de paradigma en lo que a la computación personal se refiere. Los tradicionales equipos de sobremesa y portátiles están siendo remplazados para diversas tareas por equipos móviles del tipo teléfono, PDA y tablets.

Un dispositivos móvil inteligente, el más común un teléfono inteligente (smartphone en inglés) es un dispositivo móvil construido sobre una plataforma de informática móvil, más la capacidad de computación avanzada y posiblemente con conectividad de un teléfono móvil, con la posibilidad de instalar aplicaciones para cualquier uso. El término «inteligente» hace referencia a la capacidad de usarse como un computador de bolsillo, llegando incluso a remplazar a un computador personal en algunos casos. En la figura 3 vemos algunos ejemplos.

Figura 3 - Ejemplos de smartphones

Entre otras características comunes está la función multitarea, el acceso a Internet vía WiFi o 3G, a los programas de agenda, a una cámara digital integrada, administración de contactos, acelerómetros, GPS y algunos programas de navegación así como ocasionalmente la habilidad de leer documentos de negocios en variedad de formatos como PDF y Microsoft Office.

El segundo dispositivo móvil inteligente más común es la tableta. Una tableta (del inglés: tablet o tablet computer) es un tipo de computadora portátil, de mayor tamaño que un smartphone o una PDA, integrado en una pantalla táctil (sencilla o multitáctil) con la que se interactúa primariamente con los dedos o una pluma stylus (pasiva o activa), sin necesidad de teclado físico ni ratón. Estos últimos se ven reemplazados por un teclado virtual y, en determinados modelos, por una

mini-trackball integrada en uno de los bordes de la pantalla. En la figura 4 podemos ver un tablet con S.O. Android.

Figura 4 - Tablet

El término puede aplicarse a una variedad de formatos que difieren en la posición de la pantalla con respecto a un teclado. El formato estándar se llama pizarra (slate) y carece de teclado integrado aunque puede conectarse a uno inalámbrico (por ejemplo Bluetooth) o mediante un cable USB (muchos sistemas operativos reconocen directamente teclados y ratones USB). Otro formato es el portátil convertible, que dispone de un teclado físico que gira sobre una bisagra o se desliza debajo de la pantalla. Un tercer formato, denominado híbrido (como el HP Compaq TC1100), dispone de un teclado físico, pero puede separarse de él para comportarse como una pizarra. Por último los Booklets incluyen dos pantallas, al menos una de ellas táctil, mostrando en ella un teclado virtual.

Hoy en día las tabletas utilizan mayoritariamente un sistema operativo diseñado con la movilidad en mente. Por ejemplo, iOS, Android y el minoritario Symbian provienen del campo smartphone, donde se reparten el mercado, MeeGo y HP webOS provienen del mundo PDA. Son un caso a parte los nuevos sistemas de Microsoft, pensados para funcionar tanto en ordenadores de escritorio como en dispositivos móviles. A continuación se muestra el mercado de negocio que generan las aplicaciones para móviles en los años 2009 y 2010, y el impresionante incremento que sufrieron en el año 2010, como se ve en la figura 5.

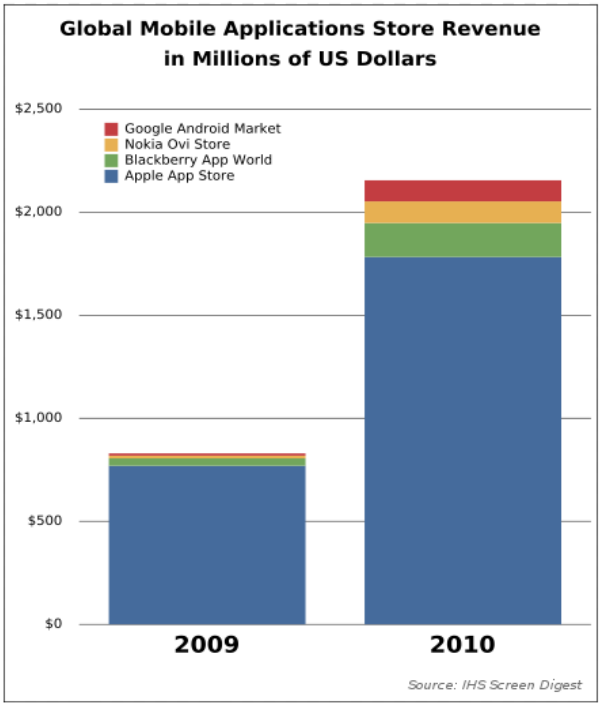

Figura 5 - Mercado de aplicaciones móviles

Esta tendencia parece mantenerse en el tiempo hasta el día de hoy y el mercado de las aplicaciones móviles se encuentra en continuo movimiento, mientras cada vez, más y más personas integran en su vida diaria nuevos dispositivos móviles. Este mercado emergente, que crece conforme la tecnología se desarrolla y se hace más accesible al gran público, es la motivación principal para embarcarse en un proyecto de este tipo desde el punto de vista social y económico.

Una vez hemos establecido la existencia de un mercado en el que situar el producto que queremos desarrollar, conviene examinar las diferentes plataformas que existen en él. Así como existe un gran número de terminales en el mercado, también existe un gran "ecosistema" de sistemas operativos funcionando en ellos, y la distribución de cuota de mercado para cada uno de ellos ha variado mucho en los últimos años.

En la figura 6 se puede observar a través del tiempo cual ha sido la aceptación de mercado de cada uno de los principales sistemas operativos móviles, y da a entender el porqué de un interés especial en los dispositivos Android e iOS.

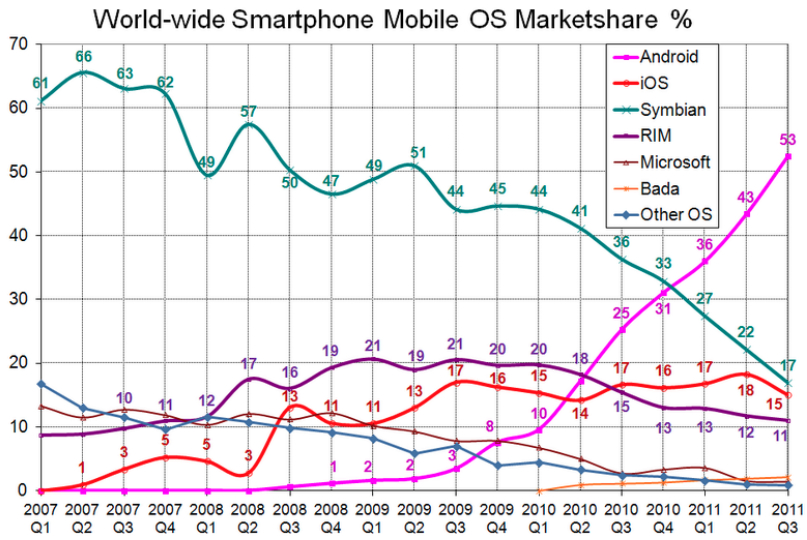

Figura 6 - Distribución de mercado de S.O. móviles

El primero representa el sistema más extendido en todo tipo de ámbitos a partir del 2010. El segundo es un sistema que siempre ha representado la vanguardia del sector y que sigue en constante crecimiento. Además tenemos que añadir a estos datos la aparición de nuevos sistemas móviles como son Windows 8 (diseñado para sobremesa y móvil a la vez) que enriquecen este mercado. Tomando como referencia la distribución de mercado actual, hemos decidido que las plataformas sobre las que centraremos nuestra atención serán iOS de Apple y Android de Google, sin descuidar la portabilidad del software hacia otros sistemas en un futuro.

Estos nuevos dispositivos al carecer de más interfaz de introducción de datos que la pantalla táctil son poco adecuados para la generación de contenido. Sin embargo, por su propia configuración y características técnicas, son muy útiles y cómodos para consumir contenidos.

3.2. Importancia de los Sistemas de Información Geográfica

Una vez hemos determinado la importancia del sector emergente que son las aplicaciones móviles, se hace necesario explicar como encajan los sistemas GIS en este ecosistema.

La geolocalización se ha convertido en un concepto familiar para todos en los últimos años. Hoy día, a nadie le es extraño consultar distintas fuentes de información geolocalizadas. La web 3.0 ha permitido que sean los propios dispositivos los que controlen su ubicación en el planeta y tomen decisiones de forma autónoma en base a ella.

Conocidos son los ejemplos de aplicaciones turísticas de asistencia al viajero que presentan su ruta en pantalla de forma tridimensional. Estos sistemas se integran tanto en dispositivos móviles como en grandes instalaciones públicas (pantallas de información táctiles, monitores...). Existen ya en explotación en Canarias sistemas

GIS integrados en tablet que ofrecen a los usuarios de ciertos hoteles información acerca de la oferta que tienen en su entorno.

De la misma forma, los servicios como índices de servicios (callejeros, páginas amarillas...) dan cada día más información geográfica de sus anunciantes [24]. El mundo del deporte se beneficia también de estos visores que permiten visualizar rutas en diferentes actividades [25].

Más allá de las aplicaciones para el gran público existen un gran conjunto de aplicaciones dedicadas a la empresa que utilizan GIS como servicio de soporte y apoyo a su logísticas (entrega de mercancía, análisis de recorridos, tráfico marítimo, estudios de marketing...).

En el mundo científico técnico también se están convirtiendo los GIS en aplicaciones indispensables. Hoy en día, no se conciben ciencias como la meteorología sin una adecuada presentación de los datos que manejan. De igual forma son indispensables para otras ramas como la exploración espacial o la la climatología. El uso de GIS no es solo indispensable en el análisis de datos, sino en la interpretación y presentación de resultados.

Como ejemplo de aplicaciones en el mundo científico contamos con el ejemplo de WorldWind5. Esta plataforma contiene multitud de ejemplos de GIS aplicado a varios sectores, fundamentalmente el científico. Entre estos se encuentran:

- GLIDER [4]: Globally leveraged integrated data explorer for research: Estudios de elevaciones del terreno.
- Gaea+ [5]: Planificación urbanística.
- Geoscience Australia's World Wind Viewer [26]: Visualización de conjuntos de datos del continente autraliano.
- JSatTrack [6]: Tracking de satélites terrestres.

Para la administración pública que mantiene un gran conjunto de datos también es muy interesante contar con una visualización adecuada de los mismos. Más aún, los sistemas GIS son a día de hoy una pieza clave en despliegue de medios en situaciones de emergencia como incendios, inundaciones, terremotos, etc.

Debido al gran número de aplicaciones en las que un sistema GIS puede aplicarse para la visualización de datos geolocalizados, es de interés contar con sistemas multipropósito que puedan utilizarse en diferentes ámbitos. Esta es la finalidad última de este proyecto, la de generar un motor genérico que permita la visualización tridimensional de un modelo de la superficie terrestre, sobre la que desplegar aplicaciones concretas para usos más específicos.

Ejemplos de software con este propósito existen (como el World Wind, antes mencionado), pero ninguno orientado a coexistir entre las plataformas móviles de las que hablamos anteriormente. El objetivo de este proyecto es generar un modelo de sistema único, manejable y modificable con el que se puedan desarrollar aplicaciones específicas en los sistemas de Apple y Google, sin un coste excesivo.

4. Requisitos funcionales

El sistema que pretendemos construir en el presente proyecto se trata de un visualizador de terreno genérico con el que se pueda trabajar en las plataformas móviles Android e iOS.

Como tal, no se pretende implementar funcionalidad para aplicaciones de un campo concreto, como podrían ser mediciones topométricas, geotagging o visualización de rutas o modelos complejos. Por otro lado, lo que sí se pretende es que el modelo de software generado sea lo suficientemente abierto y genérico para poder albergar en un futuro todas estas funcionalidades.

Nos centraremos por tanto en la construcción y visualización de una escena 3D que represente nuestro planeta, sin funcionalidades específicas. Enumeraremos, por tanto, los requisitos funcionales más obvios que hemos decidido incorporar al diseño.

4.1. Uso del sistema

En el campo de la usabilidad, encontramos que la interfaz de acceso al sistema se ve restringida a una pantalla táctil que permite interacción multipuntero con la superficie de renderizado.

Este tipo de sistemas establece eventos de Down, Move y Up de 1 hasta N punteros simultáneos. Mediante distintas configuraciones se desea poder reconocer gestos de pinching, spreading, dragging, tapping y double-tapping sobre la interfaz [7]. Estos gestos se pueden observar gráficamente en la figura 7.

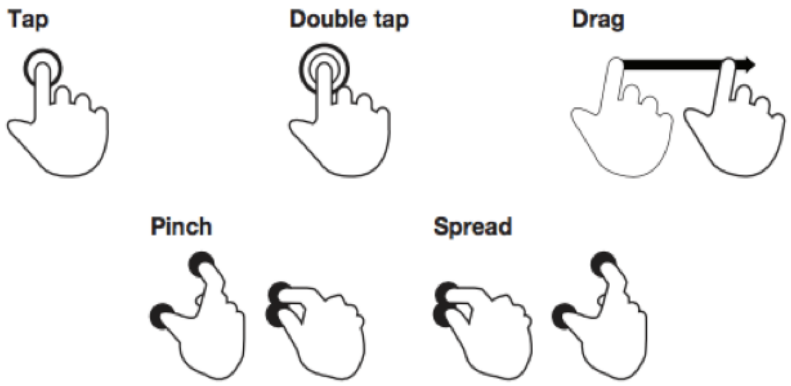

Figura 7 - Gestos multitáctiles [7]

Con estos métodos se debe poder variar la posición y centro de vista de una cámara que rote alrededor del modelo que representa al planeta. La cámara debe poder:

- Hacer zoom, para poder acercarse y alejarse de una determinada zona del terreno.

- Rotar con respecto a su eje de vista.
- Rotar con respecto a la normal de la superficie del planeta de forma que se pueda hacer vistas rotatorias alrededor de montañas y otros elementos geográficos.

Todos estos gestos y eventos deben ser recibidos por los elementos dibujables de la escena que responderán a ellos. De esta forma se podrá interactuar con cualquier elemento que el usuario vea en pantalla.

4.2. Visualización de terreno multiresolución

La recreación de modelos de representación de grandes terrenos es una tarea compleja debido a la enorme cantidad de información a procesar. Estos van desde texturas, a elevación del terreno y mallado.

Se requiere de nuestra aplicación que sea capaz de lidiar con estos elementos de una forma inteligente para que, gestionando estos recursos, se pueda mantener un frame-rate constante y que permita visualizar el modelo sin saltos. Esta problemática es bien conocida en el campo de los videojuegos.

La cantidad de datos presentes en los modelos de terreno actuales hace imposible cargar un modelo tridimensional que represente el planeta Tierra en su conjunto, y menos en un dispositivo de recursos muy limitados, como con los que estamos trabajando.

Es por ello que nos vemos obligados a utilizar un sistema de Level Of Detail (LOD) [8] que nos permita ver con alto detalle las zonas de terreno únicamente las zonas cercanas a la cámara.

Así mismo el proceso de renderizado ha de ser lo bastante óptimo para permitir presentar muchos elementos en pantalla sin aumentar en exceso el tiempo de frame.

4.3. Conectividad

Nuestra aplicación ha de considerarse un visor de datos geográfico. Como tal, su funcionamiento depende de los datos que le son suministrados por fuentes que son ajenas al propio sistema.

El software a desarrollar debe contar con las herramientas para poder comunicarse de forma asincrónica con sus fuentes de datos a través de la red.

En nuestro caso, estas fuentes son servidores de datos que se encuentran distribuidos por la red y que se comunicarán con nosotros para ofrecernos datos geolocalizados que representen imágenes raster, información vectorial, marcadores, o información sobre elevación del terreno.

Para comunicarnos con estos servidores y pedirles datos en la localización en la que los necesitamos y en el formato en el que queremos existe una variedad de protocolos que podemos utilizar dependiendo del servidor y los datos en cuestión.

El más importante quizás, y en el que basaremos este proyecto, es Web Map Service (WMS) [9], un estándar proporcionado por la Open Geospatial Consortium [27], cuya definición puede encontrarse en el portal web de la organización.

Este protocolo permite solicitar imágenes raster que se correspondan con un determinado sector definido en un sistema de coordenadas terrestre. Con él pretendemos obtener las imágenes que plasmar sobre el terreno en nuestro modelo terrestre.

El acceso a red y a servidores cuya eficiencia a la hora de darnos los datos desconocemos, nos obliga a hacer un uso adecuado de este recurso. Es por esto, que otro requisito será que el sistema sea capaz de almacenar los datos de forma local. Nuestro motor ha de contar con una caché de los recursos obetenidos de la red con un doble objetivo. El primero es hacer un uso más razonable de los recursos de red. El segundo es habilitar el desarrollo de aplicaciones que sean capaces de funcionar en entornos offline con datos pregrabados.

5. Diseño

En el siguiente apartado, se pretende describir los aspectos fundamentales del diseño de nuestro sistema GIS. Entre ellos cabrá destacar lo relativo a la composición de la escena, la división en subsistemas independientes, así como la forma en la que hemos afrontado el problema de desarrollar un software único para distintas plataformas.

5.1. Diseño arquitectónico

Lo primero que hemos de hacer es definir cuales son los diferentes módulos que existen en el software y el tipo de relaciones que existen entre ellos. Al tratarse de un sistema diseñado para visualizar datos (imágenes de terreno, alturas, marcas…) que no se encuentran necesariamente en la máquina local, es relativamente directo plantearlo como una arquitectura de tipo cliente-servidor.

5.1.1. Los servidores de datos

La parte servidor se encuentra ya operativa y disgregada a través de internet en la forma de hosts que contienen los datos a los que se va a acceder. En principio el protocolo que utilizaremos para acceder a esas imágenes y mapas de elevación será el protocolo WMS, definido por el Open Geospatial Consortium dentro de OpenGIS. Este es un protocolo abierto y de libre uso, que nos permitirá pedir imágenes geolocalizadas de mapas en múltiples proyecciones y con infinidad de parámetros.

El estándar produce mapas de datos referenciados espacialmente, de forma dinámica a partir de información geográfica. Este estándar internacional define un "mapa" como una representación de la información geográfica en forma de un archivo de imagen digital conveniente para la exhibición en una pantalla de ordenador. Un mapa no consiste en los propios datos. Los mapas producidos por WMS se generan normalmente en un formato de imagen como PNG, GIF o JPEG, y opcionalmente como gráficos vectoriales en formato SVG (Scalable Vector Graphics) o WebCGM (Web Computer Graphics Metafile).

Este protocolo se encuentra actualmente en constante actualización, siendo las versiones 1.1 y 1.3 las más extendidas y por ello las que interesa implementar en nuestro agente.

El protocolo WMS es por tanto, un protocolo de red versátil y complejo que nos permite solicitar imágenes que se generarán durante la petición. Esto nos permite solicitarlas con el estilo de visualización y transparencias que queramos y lo que es más importante, del sector de la superficie terrestre que queramos definir.

Existen por el contrario, otros servidores que utilizan protocolos que permiten solicitar imágenes ya precacheadas. Normalmente la organización de dichas imágenes se corresponde con uno o varios niveles de detalles, que definen cuadrículas sobre la superficie terrestre.

Ejemplos de estos servicios son los mapas de Virtual Earth (Bing) [10] o algunos servidores de Open Street Map [11]. Estos servidores se caracterizan por dar respuestas más rápidas que las ofrecidas por WMS.

En términos generales la arquitectura de implementación del servidor de datos es ajena a este proyecto y por tanto solo nos ha de interesar que versiones del protocolo aceptan los servidores con los que vayamos a interactuar y, en términos de rendimiento, otros parámetros como el tiempo de respuesta promedio, el número de peticiones concurrentes que atienden, número límite de peticiones etc.

5.1.2. El cliente

Como decíamos anteriormente, el fruto de este proyecto es la generación de un cliente que permita visualizar datos geográficos, obtenidos de una o varias fuentes, en plataformas móviles.

Estas plataformas son principalmente smartphones y tablets que soportan sistemas operativos propios de estas nuevas arquitecturas. En el caso que nos ocupa, estás plataformas serán iOS y Android por ser los S.O. más ampliamente extendidos.

Para más información acerca de las arquitecturas de estos sistemas operativos y del papel que conforma nuestro sistema en ellas, véase el apartado Arquitecturas de los sistemas operativos móviles actuales.

En lo que respecta al diseño de la aplicación en si misma, se ha optado por un diseño modular que permita abordar las diferentes fases de implementación y prueba de forma separada.

Este diseño modular nos permite diseñar las clases de forma local, impidiendo el flujo descontrolado de datos dentro del sistema y de esta forma aislar los posibles errores que surjan durante el desarrollo. Se potencia a la vez la inteligibilidad del software y su reutilización y facilidad modificación.

En el siguiente apartado se detallarán cuales son los subsistemas base identificados y los aspectos que se han tenido en cuenta en el diseño de los mismos.

5.2. Subsistemas

Una vez hemos descrito las funcionalidades que esperamos de nuestro cliente, es hora de definir los módulos que permitirán desarrollar elementos dibujables en la escena. Este sistema base, proporcionará clases y servicios necesarios para la implementación de los demás elementos que se requieran para dibujar la superficie del planeta y para desarrollar módulos que añadan contenido a la aplicación final.

El esquema general de funcionamiento de la aplicación puede describirse en términos de módulos, subsistemas y paquetes como se observa en la figura 8.

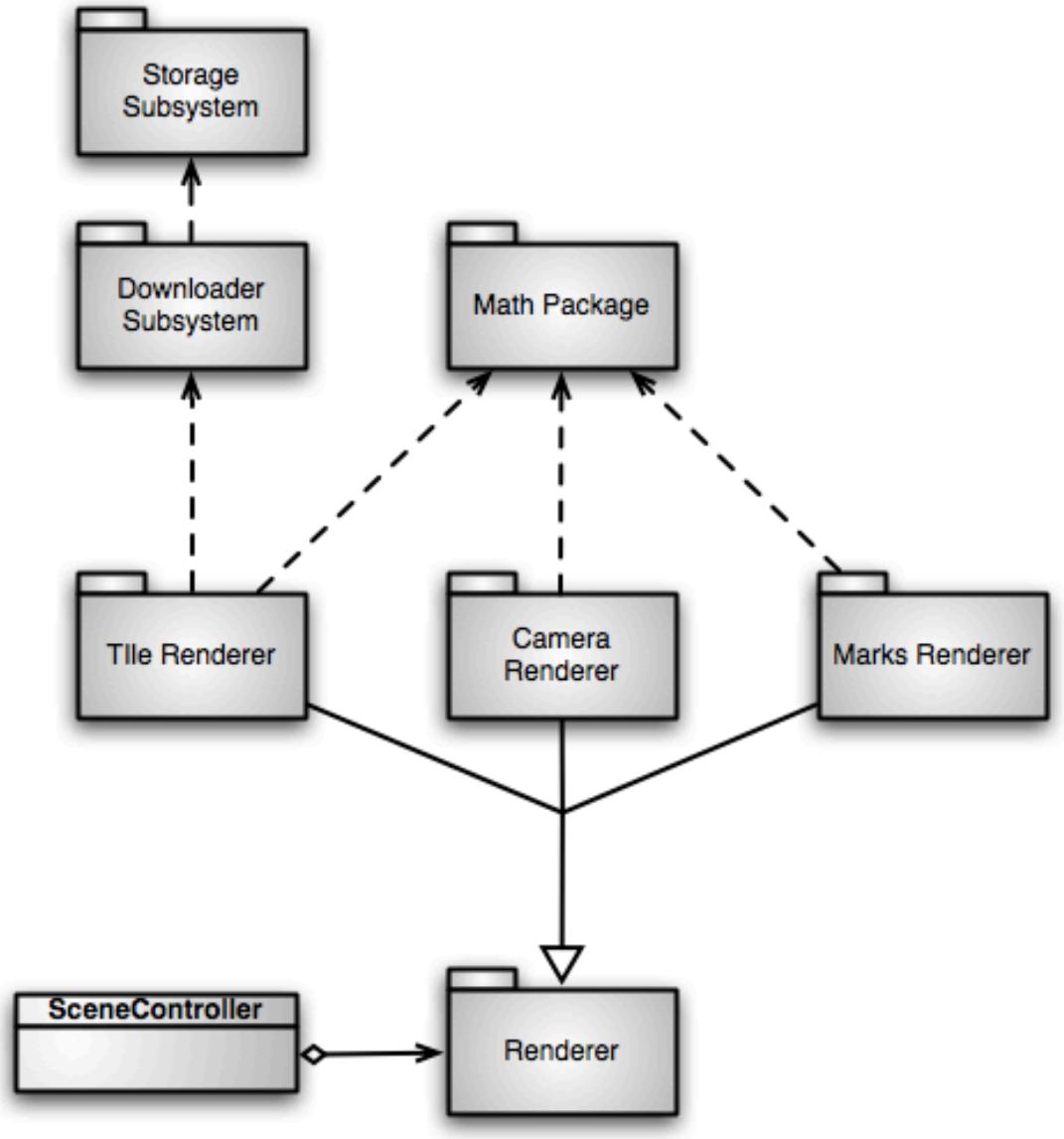

Figura 8 - Esquema general del proyecto

La definición y utilidad de cada uno de estos módulos se tratará con más detalle en secciones posteriores, así como ciertos detalles que habrá que tener en cuenta en la etapa de implementación.

5.2.1. Paquete matemático

Nuestro motor, como cualquier motor de gráficos 3D precisa de la ejecución de muchos cálculos. Por tanto, se ha decidido implementar un conjunto de clases que abstraigan las principales funciones matemáticas y geométricas que se necesitarán a la hora de implementar el resto de módulos del sistema.

Se persigue con esto un doble objetivo. Por un lado, simplificar y hacer más legible el código que haga uso de la interfaz matemática, generando métodos que sean lo más reutilizables y semánticamente ricos posible. De la misma forma, tener los

métodos matemáticos centralizados y reutilizados nos permite centrarnos en que su implementación sea lo más eficiente posible.

Por otro lado, se pretende que este paquete de funciones sea independiente de otros paquete del sistema, sin existir dependencias hacia otros módulos. Con esto conseguimos mejor mantenibilidad y reusabilidad en otros proyectos del mismo código.

Se beneficiarán de este módulo especialmente los renderers de escena, puesto que los cálculos en los que se fundamenta la generación del terreno son puramente geométricos. De la misma forma, los cálculos matriciales necesarios para la implementación de la cámara y su desplazamiento por la escena también se fundamentarán en métodos de este paquete.

Por otro lado, no podemos suponer que las funciones matemáticas básicas (seno, coseno, logaritmo..) vayan a existir con la misma interfaz en ambas plataformas. A su vez, resulta de interés tener ciertos valores como NaN o Infinito localizados dentro de una interfaz común. Es por ello que lo primero que vamos a implementar de este módulo es una interfaz llamada IMath, que utilizará las librerías existentes en cada plataforma para implementar las funciones matemáticas simples**.**

La implementación de IMath se hará dependiendo de la plataforma, utilizando las API's disponibles y procurando maximizar la eficiencia. El resto de clases utilizarán únicamente las funciones matemáticas que ofrece IMath, a la que accederán mediante un patrón singleton.

A la hora de diseñar este módulo, hemos identificado las siguientes entidades que deberá contener este módulo. A continuación se listan las clases principales y se definen sus funcionalidades más generales, aunque no por ello las únicas.

- Ángulo: Abstracción que contendrá un valor angular. Deberá generarse indiferentemente a partir de dos métodos fromDegrees() y fromRadians(). Contendrá métodos que devuelvan sus razones trigonométricas (seno, coseno, tangente, secante, etc.) así como métodos de interpolación y de traslación al rango 0...360.
- Vector 2D y Vector 3D: Representan conjuntos de pares XY y ternas XYZ respectivamente que simbolizan puntos (o vectores libres) en el espacio cartesiano. Aportan funcionalidades clásicas como cómputo del módulo, normalización, cálculo del ángulo interno, producto vectorial y escalar, etc.
- Matriz 4x4: Conjunto de 16 valores que conforman una matriz de 4x4. Clase básica en todo lo referente al cálculo de geometrías en la escena, así como de la creación de las matrices de vista y proyección. Una matriz podrá definirse a partir de sus elementos o de constructores específicos que generen las matrices de traslación, rotación y escalado. También habrá constructores para las matrices de proyección (3D a 2D), proyección inversa (2D a 3D) y de vista (a partir de los vectores Up, Center y Position).

También es de vital importancia que las matrices tengan implementadas la aritmética de matrices especialmente la multiplicación y la función inversa.

- Coordenadas 2D y Coordenadas 3D: Sistema de posicionamiento angular sobre la superficie del planeta, formado por los ángulos latitud y longitud y acompañado por la altura en el caso 3D. Estas clases nos ayudan a definir un punto en el espacio elipsoidal de la superficie terrestre. Deben contener funcionalidades como la interpolación, comprobaciones de rango y funciones aritméticas. En la figura 9 se puede ver como está distribuido el sistema de coordenadas geográficas.

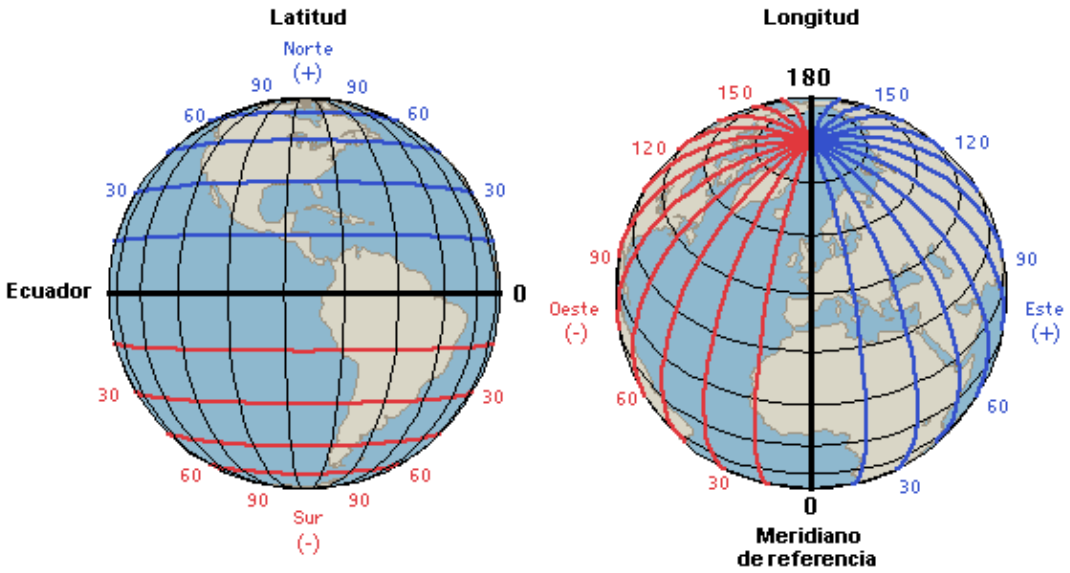

Figura 9 - Coordenadas geográficas

- Sector: Representa una porción de la superficie terrestre contenida entre unas latitudes y longitudes máximas y mínimas. Nos servirá para representar la extensión de las capas que queramos dibujar sobre el globo y la de los diferentes pedazos en los que hayamos descompuesto su superficie para un determinado nivel de detalle.
- Elipsoide: La Tierra es modelada como un elipsoide en los campos de astronomía y geociencias. Se han utilizado varios modelos pero el más aceptado hoy en día se denomina WGS84 [12], utilizado, entre otros, por el sistema GPS y que podemos ver en la figura 10.

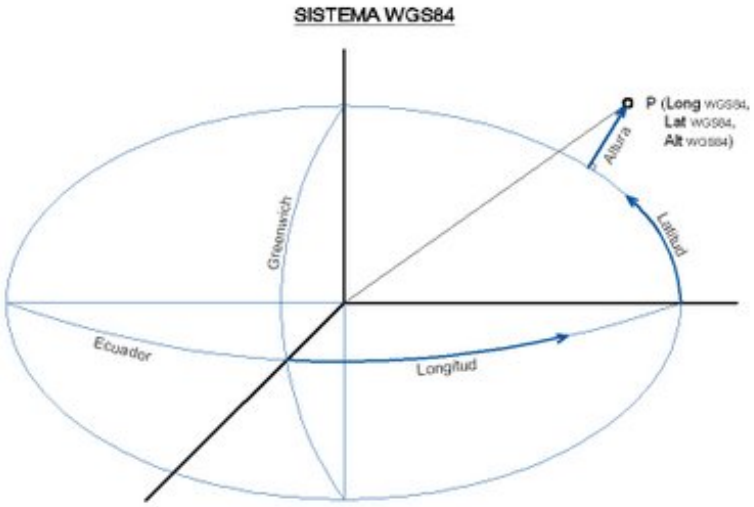

Figura 10 - Elipsoide

Debido a que estamos modelando un planeta, cuya definición matemática más aceptada es una elipsoide, tiene sentido que nuestro software tenga una entidad que aúne los cálculos referidos a elipsoides.

Entre estos cálculos se encuentran corte del elipsoide con una recta, el cálculo de normales a partir de un punto de la superficie, el cálculo de normales y las conversiones entre espacio cartesiano y el elipsoidal (Vector a Coordenadas y viceversa).

5.2.2. Módulo de descargas

El volumen de datos necesario para modelar el planeta Tierra es muy elevado. Se trata de mapas que cubren toda la superficie terrestre en multiresolución y nos permiten mostrar el terreno con elevaciones e imágenes. El problema se complica si en vez de una sola imagen para el terreno tenemos varias que se han de combinar. Es por esto que uno de los prerrequisitos que establecimos para el funcionamiento de la aplicación es tener acceso vía red a los servidores que le proveerán de esos datos.

Este acceso ha de ser manejado de forma que se puedan realizar descargas asíncronas de datos que no ralenticen o bloqueen el hilo de renderizado. Al mismo tiempo han de ser rápidas, para que cuando la cámara se desplace a una zona nueva del terreno los recursos necesarios para mostrarlo lleguen a tiempo y el usuario no observe el terreno demasiado tiempo con una resolución poco adecuada.

La solución que finalmente se ha optado para la implementación de este módulo de descargas, también llamado en el documento Downloader, de forma que cumpla los requisitos antes mencionados es la siguiente:

Al recibir una petición de descarga, el módulo la almacena como descarga pendiente. Dichas descargas estarán ordenadas por tiempo en el que fueron solicitadas y por un valor de prioridad.

Se contará con un pool de hilos en permanente ejecución. Dichos hilos de trabajo (working thread) estarán ejecutando un bucle infinito hasta que se les ordene cerrarse.

En cada iteración, cada hilo comprueba una pila de descargas común. En ella se encuentran las descargas que han sido encargadas pero que no han comenzado aún. El hilo de trabajo selecciona uno en función de su prioridad y antigüedad.

Al tomar una descarga la marca como en proceso y la ejecuta sincrónicamente utilizando las API's de la plataforma.

Al finalizar, pasa los datos obtenidos al hilo principal, el cual al recibirlos llama al callback que corresponde a dicha descarga. Una vez finalizado el callback la tarea de descarga se borra de la pila.

Ni que decir tiene que el acceso a la pila de descargas en espera requiere de mecanismos de sincronización que garanticen la exclusión mutua de los hilos que tengan intención de usarla. Tanto Objetive-C como Java contienen métodos de sincronismo como semáforos u objetos monitor convenientes para esta tarea. Sin embargo, tal como se verá en la etapa de implementación, es más cómodo para el desarrollo el apoyarse en las API's propias de la plataforma que trabajan a más alto nivel.

A modo de gráfico aclaratorio se muestra en la figura 11 el esquema general de funcionamiento de este módulo.

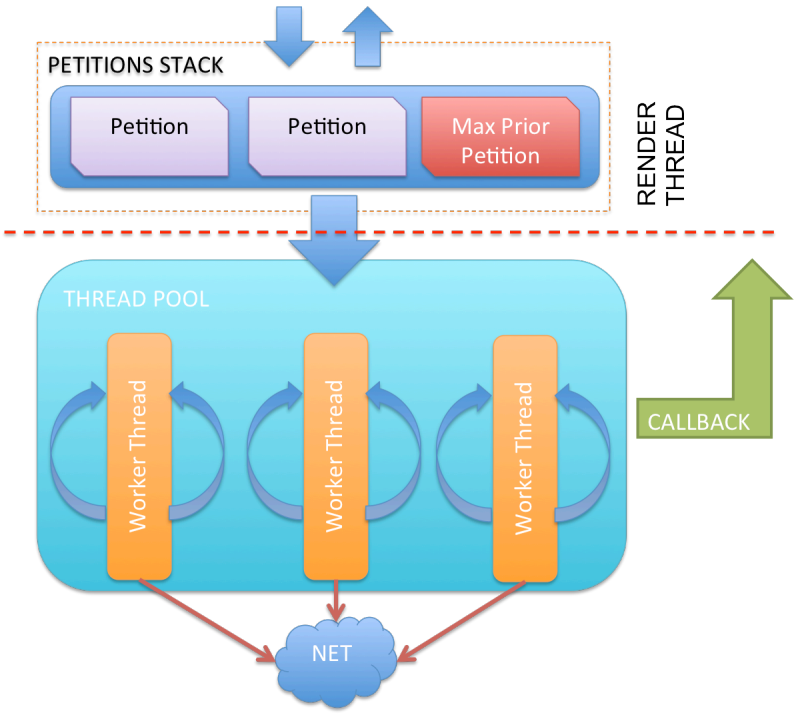

Figura 11 - Modelo de funcionamiento del Módulo de Descargas

Desde el punto de vista de diseño funcional este módulo será un subsistema con un único punto de entrada (patrón Facade [14]). Desde esta interfaz se tendrá acceso a los siguientes métodos:

- Iniciar Descarga Asíncrona: Esta es la funcionalidad principal. Al subsistema Downloader se le pasa la URL relativa al recurso que se desea descargar y un valor numérico que indique la prioridad de la descarga. Junto con ella se le pasa una referencia al objeto que está esperando el resultado de dicha descarga. Este objeto deberá contener una función de callback que será a la que se le pasará el resultado de la descarga. Dicho resultado podrá ser bien los datos descargados en forma de array de bytes o bien un código de error que indique por qué no se han podido recibir dichos datos.

- Detener Descarga: En ocasiones el usuario puede pasar brevemente sobre franjas de terreno, de forma que al salir de ellas los datos necesarios todavía no han sido descargados. En este caso, resulta interesante para maximizar la eficiencia de la aplicación eliminar las descargas no comenzadas de datos que ya no son relevantes, de forma que las que si lo son se puedan ejecutar lo antes posible. Si la descarga que se pretende descargar ya ha sido iniciada, se le permite completarse, pero no se llama a su callback asignado ya que el objeto que la requería podría haber dejado de existir.

5.2.3. Módulo de almacenamiento

Como se ha comentado anteriormente el acceso a red es fundamental en este tipo de aplicaciones. El acceso a los datos contenidos en el servidor puede ser lento o incluso puede haber un límite máximo de peticiones que este nos permite hacer en un periodo de tiempo. Es por eso que es necesario optimizar el acceso a este recurso, impidiendo que el sistema pida recursos de forma repetida.

Con este fin, se ha diseñado un módulo que recoja y almacene las respuestas recibidas por el encargado de realizar las descargas de red. De esta forma cuando se reciba una petición en el módulo de descargas que ya ha sido almacenada con anterioridad, esta puede ser atendida automáticamente de forma síncrona. Este módulo lo llamarémos módulo de almacenamiento o Storage en inglés.

El proceso sería el siguiente:

1. El módulo de descargas recibe una petición con su URL y objeto callback.

2. Dicha URL se envía al módulo Storage para comprobar si el recurso existe en la caché.

    a. Si la respuesta es positiva, el almacenamiento devuelve los bytes asociados al recurso y se llama al callback convenientemente, terminando todo el proceso.

    b. Si la respuesta es negativa el proceso sigue como es descrito en el apartado que describe el módulo de descargas.

3. Una vez finalizada la descarga y tras haber llamado al callback que esperaba los datos de respuesta, estos se almacenan en el módulo de almacenamiento asociándolos a la URL que los identifica.

El esquema de funcionamiento queda reflejado en la figura 12.

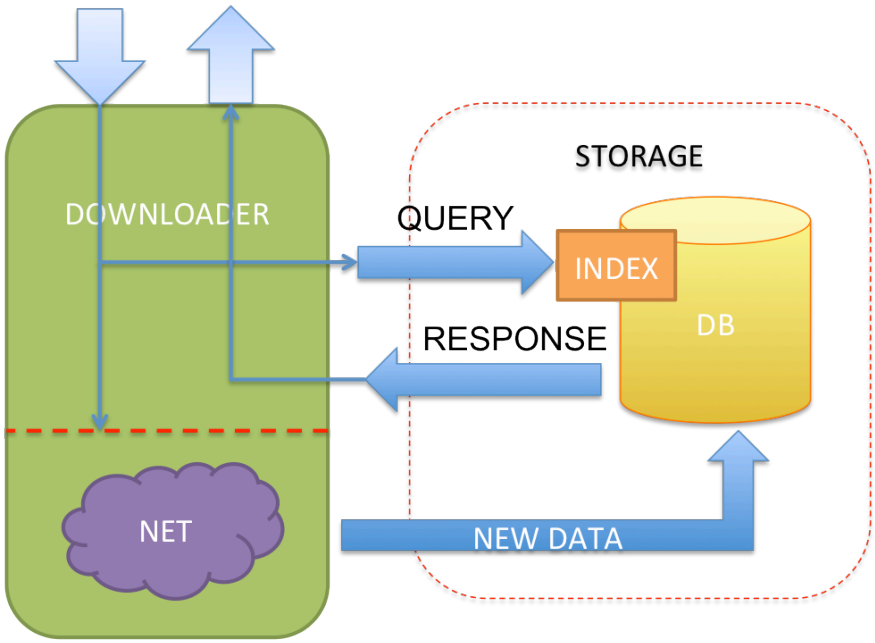

Figura 12 - Modelo de funcionamiento del Módulo de Almacenamiento

El módulo Storage por tanto ha de poseer una interfaz fachada, accedida principalmente por Downloader, que contenga los métodos necesarios para comprobar si contiene un recurso, obtener los datos y escribir un nuevo recurso a registrar. También es pertinente métodos que permitan vaciar los datos contenidos en el almacenamiento si, por ejemplo, los datos del servidor han sido actualizados o si la cuota de disco de la aplicación se ha alcanzado.

Desde el punto de vista de la implementación los datos que se van a almacenar son una relación entre una cadena y una ristra de bytes (Blob). Como tal, a fin de maximizar la rapidez del sistema, se utilizará una base de datos relacional que nos permita hacer búsquedas dentro de la misma de la forma más rápida posible.

En las plataformas con las que estamos trabajando existen API's sencillas que permiten la creación y acceso a bases de datos SQLite [15]. SQLite es una librería que implementa bases de datos transaccionales sin necesidad servidores o configuración, con buen rendimiento en las búsquedas.

Utilizando este motor generaremos la tabla que contenga los datos descargados, utilizando como clave primaria la URL de los mismos. De la misma forma generaremos un indice para acelerar las búsquedas con esa clave primaria. En futuras extensiones del sistema podrían añadirse otros parámetros como tiempo de expiración de los recursos, para darle más versatilidad al sistema.

5.2.4 Elementos de la escena

El núcleo del framework que estamos diseñando es una escena tridimensional en la que, en principio, se muestra un modelo del planeta Tierra. Sin embargo, el modelo software ha de ser extensible para albergar cuantos elementos de escena creamos convenientes.

Para esto se ha ideado un sistema en el que la aplicación contiene un conjunto de módulos "dibujables", que a partir de ahora llamaremos Renderers, en el que se pueden añadir cuantos objetos se desee y activarlos o desactivarlos a voluntad durante la ejecución.

Por ejemplo, el framework podría ser utilizado para mostrar mapas urbanos. Para ello podría ser interesante un renderer que dibuje edificios tridimensionales sobre la superficie del globo. En este esquema el globo es simplemente un objeto dibujable más que se integra en la escena.

Cada renderer que definamos deberá tener métodos para:

- Determinar si es posible dibujarlo. Por ejemplo, el renderer que dibuje la superficie del globo no se podrá pintar hasta que hayan llegado texturas al menos para cubrir toda su superficie con el mínimo nivel de detalle.
- Ejecutar tareas de pre-pintado. Carga de recursos, por ejemplo.
- Rutina de pintado. Método previsiblemente más pesado que contenga las llamadas al motor gráfico.
- Rutina de post-pintado. Método ejecutado tras la ejecución del pintado en un frame.

El hilo principal de nuestra aplicación se dedica por tanto, a estar ejecutando un bucle en la que cada iteración se corresponde con la generación de un frame. Para cada imagen se comprueba cada renderer registrado se comprueba su viabilidad y en caso de poderse se ejecutan las tareas de pre-pintado, pintado y post-pintado.

En apartados posteriores, se explica con más detalle el funcionamiento de los principales renderers.

5.3 Manejo de Cámara

Este módulo es el encargado del manejar de la cámara virtual que circunda el globo en nuestra aplicación, siendo el encargado de la gestión de las matrices de vista y de la interacción con el usuario.

5.3.1. Operaciones de pintado relacionadas con la cámara

Este módulo está integrado por defecto en el sistema, puesto que es fundamental en el pintado de la escena. Además ha de ser ejecutado antes del pintado de los renderers de la escena, puesto que es el encargado de generar las matrices de pintado que utilizará el motor gráfico para convertir los vértices (almacenadas en coordenadas de escena) a coordenadas de pantalla.

Matriz de modelo

La primera de estas matrices es la matriz de modelo (model matrix) [16], que convierte los vértices de la escena al espacio de vista del espectador. Esta matriz lógicamente variará conforme el espectador navegue por la escena o dirija el centro de vista en diferentes lugares.

La matriz de modelo está definida por un sistema de tres vectores:

- Posición (Position): Indica la posición de la cámara en coordenadas del mundo.
- Centro (Center): Lugar a donde se dirige el centro de vista.
- Arriba (Up): Vector perpendicular al definido entre la posición y el centro y que coincide con la perpendicular a la vista.

Si tomamos Y como el vector de Up e Z como la dirección de vista, la cámara define unos planos de corte como en la figura 13.

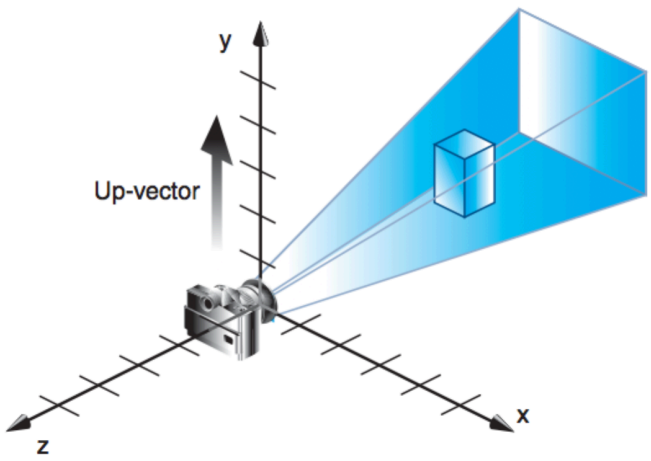

Figura 13 - La cámara

La matriz de modelo se construye siguiendo una formulación matemática preestablecida en función de los parámetros antes mencionados. De esta forma, cada vez que se cambie la posición de la cámara o su orientación, esta matriz ha de ser recalculada.

Multiplicando los vértices expresados en coordenadas 3D de la escena por esta matriz, da como resultado los mismos vértices en coordenadas de cámara, o relativas al espectador.

Matriz de proyección

La proyección 3D es el método por el cual se mapean puntos tridimensionales sobre un plano bidimensional, en este caso, la superficie de pintado [16]. Existen diversos métodos para ello, pero los dos más importantes y utilizados en los gráficos por computador son:

- Proyección ortográfica: No tiene en cuenta la distancia entre el espectador y el objeto, de forma que los objetos lejanos se ven del mismo tamaño que los grandes. Es muy utilizada en diseño e ingeniería para mostrar las proporciones de los elementos de la escena con detalle.
- Proyección en perspectiva: La vista en perspectiva muestra los objetos lejanos de menor tamaño, tal como lo haría el ojo humano. De esta forma se consigue dibujar escenas con aspecto realista. El sistema que estamos diseñando utilizará esta perspectiva, pues el objetivo es mostrar el modelo del planeta de la forma lo más realista posible.

Esta matriz se computa a partir de los siguientes parámetros:

- Tamaño horizontal: Definen el tamaño del plano de proyección a la izquierda y derecha respectivamente.
- Tamaño vertical: Definen el tamaño del plano de proyección hacia arriba y abajo respectivamente.
- Corte en Z cercano y lejano (zFar y zNear): El primero define a que distancia de la posición de la cámara está el plano de proyección. El segundo define el plano de corte a partir del cual los objetos ya no serán pintados al estar demasiado lejos.

Estos valores determinan un espacio de proyección en forma de pirámide truncada. Los vértices contenidos dentro de este espacio son los que aparecerán dibujados en pantalla. Este espacio se denomina frustum y se puede observar en la figura 14.

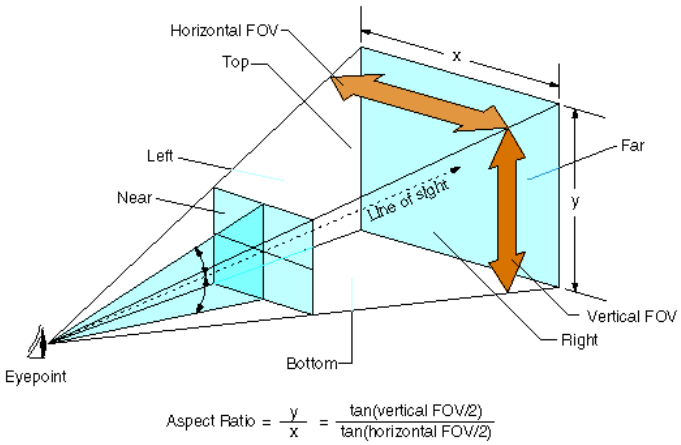

Figura 14 - Parámetros del Frustum

Esta pirámide simula el espacio de vista del ojo o de una cámara convencional. Además, si este espacio es multiplicado por la matriz de proyección, se convertirá en un cubo perfecto entre -1 y 1, en el que los objetos más cercanos al plano de proyección aparecen más grandes. Esta deformación del espacio de la escena se puede observar en la figura 15.

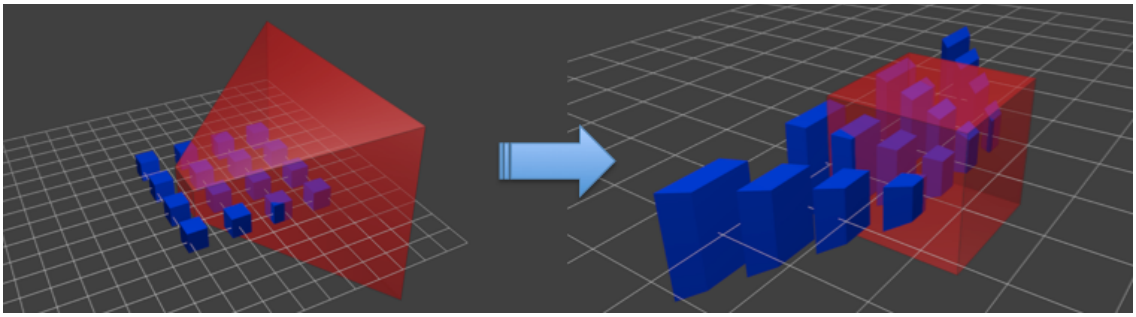

Figura 15 - Deformación del espacio producida por la matriz de proyección

De esta forma, la matriz de proyección transformará los puntos que hemos obtenido en el paso anterior en el espacio de coordenada del observador al espacio de coordenadas de pantalla. En general y para todos los vértices de la escena, se realiza el siguiente cálculo que permite pasar de la representación del mundo a coordenadas de pantalla.

$$V_{pantalla} = V_{mundo} * M_{modelo} * M_{perspectiva}$$

Para acelerar cálculos, las matrices de modelo y perspectiva se combinan previamente, obteniendo la matriz vista/modelo o modelview, que se multiplicará por todos los vértices de la escena.

Frustum

Los valores que definen el frustum [16] de la escena han de ser recalculados en función de la cercanía de la cámara al centro del planeta.

En todo momento, dada la naturaleza de la escena que estamos pintando, podemos asegurar que no estamos dibujando nada más lejano que el centro del planeta. Es por esto recomendable que el plano definido en zFar no llegue más allá del mismo.

El valor de profundidad de un pixel de la escena viene representado por un conjunto de bits en un buffer del sistema (zbuffer). Si al acercarnos a la superficie del planeta mantenemos una distancia entre zNear y zFar muy amplia, la representación de la profundidad de dos vértices cercanos puede colisionar.

Esto es debido a que los valores de Z_{eye} (distancia a cámara) se almacenan en coma flotante, lo que les otorga mucha precisión cerca del 0 pero poca para valores grandes. En proyecciones de perspectiva como la que nos ocupa esto puede ser un problema. El problema se agrava en casos como el nuestro, en que, a diferencia de otras aplicaciones como videojuegos, las escenas son muy grandes y objetos lejanos pueden fácilmente colisionar en el buffer.

Este efecto conocido como Z-Fighting [12], es evitable ajustando en cada momento la profundidad del frustum a las necesidades reales de la escena. Este proceso modifica la matriz de proyección de la cámara como se muestra en la figura.

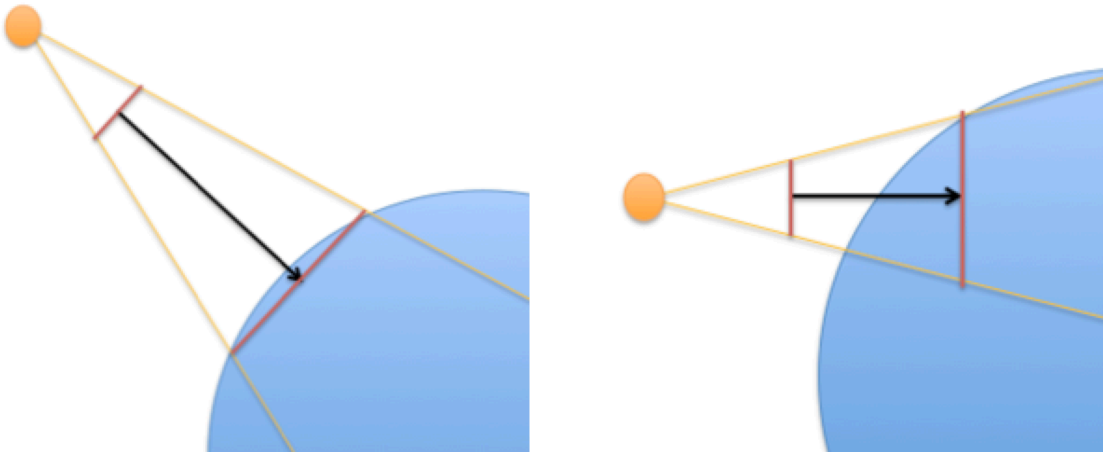

Figura 16 - Ajuste del plano ZFar

La existencia del frustum además, nos permite acelerar en gran medida el proceso de pintado. Una de las técnicas con las que hacerlo es la llamada Frustum Culling.

Este procedimiento permite no pintar los objetos de la escena que caen fuera del volumen de vista. De esta forma se acelera el proceso de renderizado, lo que redunda en poder crear escenas más complejas (con mayor número de vértices) que se renderizarán a mayores FPS.

La forma de implementación que hemos utilizado en este proyecto es añadir a cada figura de la escena un cálculo automático de una caja contenedora (o Bounding Box en inglés) [17]. Esta es una caja alineada a los ejes de la escena que contiene todos los vértices del objeto, con el volumen mínimo, parecida a la mostrada en la figura 17, en la que se muestra el contenedor de una esfera.

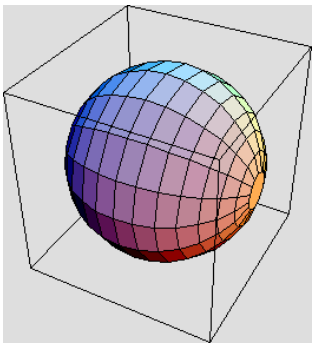

Figura 17 - Bounding Box

Comprobando que ninguno de los vértices de la caja está contenido por el frustum, podemos estar seguros de que ninguno de los vértices del objeto es visible y por tanto podemos ignorar su pintado.

5.3.2. Interacción multitáctil de manejo de la cámara

Un sistema GIS como el que estamos plateando requiere de un sistema de navegación que permita al usuario desplazarse por la escena a voluntad para poder observar la parte del globo que desee y desde el punto de vista que quiera.

Los sistemas de pantallas táctiles como con los que estamos trabajando permiten que los usuarios interactúen con los mismos a través de gestos realizados con los dedos sobre la pantalla. De esta forma, el usuario cuenta con un número virtualmente ilimitado de punteros con los que interactuar con el sistema y en nuestro caso mover la cámara de la escena.

Las plataformas Android e iOS disponen de reconocedores de eventos propios que podrían ayudarnos a distinguir unos como otros [18] [19]. Sin embargo, al ser implementaciones propias de las API's no constituyen una solución transversal al problema. Es por esto, que se ha decidido implementar reconocedores de eventos usando únicamente los eventos de bajo nivel Pulsar, Arrastrar y Levantar (Down, Move y Up) que generan los punteros al desplazarse por la pantalla. De esta forma aseguramos el mismo comportamiento en todas las plataformas [7].

Nuestra escena a parte, tiene la particularidad de centrarse en una elipsoide. Debemos, por tanto, reconocer gestos y asociarlos a movimientos de traslación alrededor del planeta, así como de rotación de la cámara en distintos grados de libertad.

Tapping

Es el gesto más sencillo. El usuario toca la pantalla con un solo dedo y rápidamente lo levanta. Este gesto no modifica la cámara en esta implementación, pero permite indicar al sistema una posición concreta sobre el globo.

Para calcular la posición del globo que el usuario está tocando con el dedo, se toman las coordenadas de pantalla en las que sucedió el evento. Estas coordenadas se convierten al espacio de escena, usando la inversa de la matriz de proyección,

localizándose en un punto dentro del plano de proyección de la cámara. Este proceso puede verse en la figura 18.

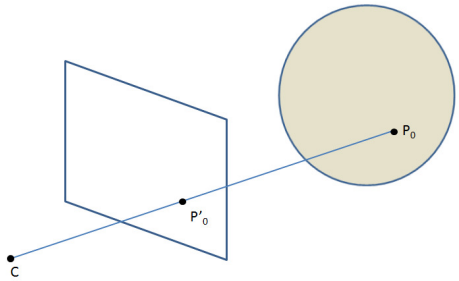

Figura 18 - Tapping

Si trazamos el rayo que conecta la posición de la cámara (supuesto "ojo" del espectador) con el punto obtenido, encontraremos que este corta al elipsoide en un punto que se corresponde con la posición señalada por el usuario. Concretamente, el rayo tendrá dos intersecciones con el elipsoide, con lo que tenemos que quedarnos con la más cercana a la posición de la cámara.

Generalmente el tapping es el primer paso para la consecución de otros gestos más complejos, y en cada caso es necesario calcular el punto contraproyectado en la superficie del planeta.

Double tapping

Este gesto es utilizado normalmente en los SIG más populares como método para acercar la cámara una cierta distancia a la posición marcada con el puntero. Este gesto, en plataformas controladas por ratón o trackpad es asociado normalmente al doble-click.

Cuando realizamos este gesto, lo que se obtiene es una animación de dos segundos, en la que la cámara se acerca a la superficie de globo. Este acercamiento es porcentual, de forma que cuanto más cerca estemos, menos se desplazará la cámara hacia delante. Además para conseguir una mayor suavidad, el movimiento de la cámara será acelerado al principio y desacelerará conforme se llegue a la posición final.

Panning

En este gesto, el usuario arrastra un dedo por la pantalla sin perder el contacto. De esta forma el globo es rotado, dando la sensación de que es el propio usuario el que gira el globo.

Para cada frame se calcula los puntos proyectados para la posición inicial del dedo (P0) y la posición en la que se encuentra en ese momento (Pk). Siendo O el centro de coordenadas, el eje de rotación vendrá dado por el producto cruzado de los vectores de OP0 y OPk. El ángulo de rotación a aplicar se calculará como el producto escalar de los mismos vectores, tal como se muestra en la figura 19.

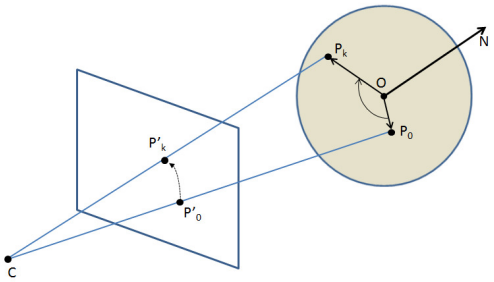

Figura 19 - Panning

Una vez el dedo se levanta de la pantalla, se considera el gesto como terminado. Aún así y para dotar de cierta naturalidad al movimiento, se continua con el movimiento produciendo una ligera rotación residual que va disminuyendo con el tiempo. El resultado es un efecto de inercia del globo, agradable e intuitivo para el usuario.

Pinching

En este evento el usuario arrastra dos dedos sobre la pantalla, acercándolos o alejándolos el uno del otro. Se produce convenientemente un efecto de "zoom", que se corresponde con un alejamiento o acercamiento de la cámara a lo largo del vector de dirección de vista.

Intentamos que durante el desplazamiento de los dedos, estos mantengan su punto de proyección sobre el planeta. De esta forma, el efecto obtenido es el mismo que se espera al realizar un gesto de zoom sobre un mapa 2D.

Si consideramos P0 y Q0 los puntos proyectados iniciales y Pk y Qk los puntos proyectados en el momento k, se busca cual sería la distancia que habría que desplazar el centro del planeta para convertir las posiciones en k a la posición 0. Esta proyección trigonométrica para mantener la posición de los dedos sobre el globo se ve en la figura 20.

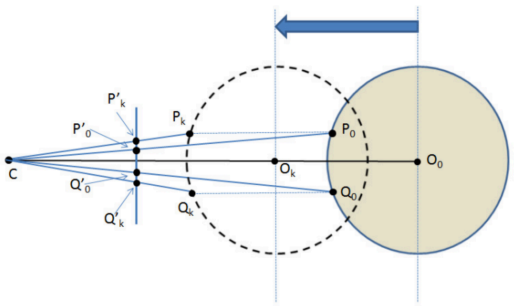

Figura 20 - Pinching

Estos cálculos trigonométricos son exactos cuando la posición central de los dedos es el centro del plano de proyección y, por tanto, el vector CO_0 son perpendiculares.

Sin embargo, en la práctica, este supuesto nos permite calcular con bastante precisión la distancia que la cámara ha de avanzar para cualquier posición de los dedos. Nótese que para realizar el zoom, es prerrequisito que ambos dedos se hayan posado sobre la superficie del planeta.

Rotating

Para rotar el globo alrededor del centro de vista, el usuario ha de arrastrar dos dedos sobre la pantalla en un movimiento circular. El eje de rotación de la cámara en este caso vendrá dado por el vector de dirección de la cámara.

Siendo P0 y Q0 las posiciones iniciales de los dedos y Pk y Qk las posiciones actuales, el ángulo se puede calcular mediante la siguiente fórmula y se observa en la figura 21.

$$\frac{|\overrightarrow{P'_0Q'_0} \cdot \overrightarrow{P'_kQ'_k}|}{|\overrightarrow{P'_0Q'_0}| \cdot |\overrightarrow{P'_kQ'_k}|}$$

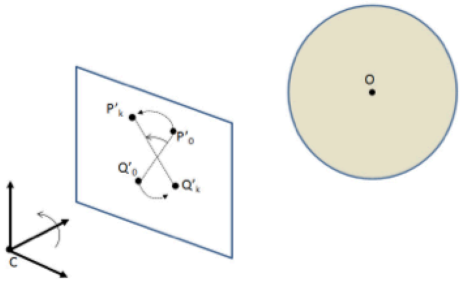

Figura 21 - Rotating

Swiping

Este gesto se consigue arrastrando dos dedos por la pantalla paralelamente por la pantalla, de forma vertical u horizontal con respecto a la posición de la misma.

Si el desplazamiento de los dedos es prioritariamente vertical, lo que se pretende es abatir la cámara, de forma que el centro de vista pase de estar enfocado al centro del elipsoide, hasta ser tangente al mismo. De una forma más explícita, lo que conseguimos es pasar de una vista cenital del terreno a una vista de paisaje, en la que se aprecia mejor las irregularidades del terreno.

En este caso, el vector de rotación de la cámara será el producto vectorial de la normal del planeta en la proyección del centro de vista por el vector dirección de la cámara. El ángulo a rotar vendrá dado por la cantidad de movimiento vertical realizado por los dedos, como se ve en la figura 22.

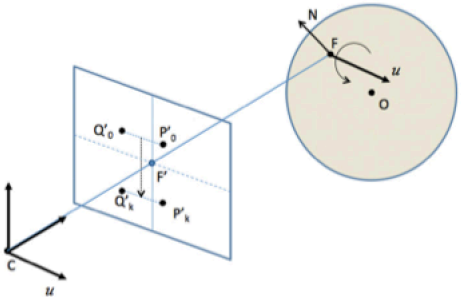

Figura 22 – Swipping vertical

En el caso de que el desplazamiento sea horizontal, lo que se pretende es que la cámara pivote alrededor de la posición en la que está situada el centro de vista. Si por ejemplo, el usuario tiene la vista centrada en una montaña, este gesto le permitirá realizar un travelling alrededor de la misma para ver todas sus caras.

El eje de rotación es por tanto el vector normal a la superficie del elipsoide en el punto proyectado del centro de la vista. El grado de movimiento será directamente proporcional al desplazamiento horizontal promedio de los mismos y a su dirección, tal como en la figura 23.

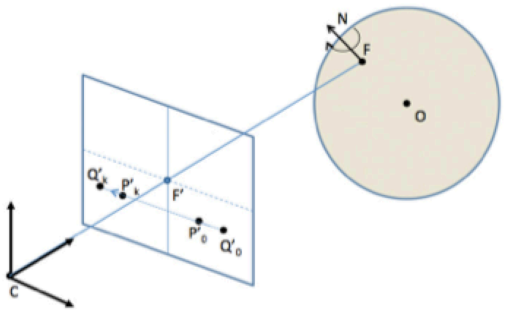

Figura 23 - Swipping horizontal

5.4. Tile Renderer

El Tile Renderer es el módulo de nuestro sistema encargado de dibujar la superficie del planeta y de mostrar las capas de imágenes que necesitemos sobre el mismo. Es por tanto, junto con el Camera Renderer, uno de los renderers necesarios para construir la escena y es posiblemente el más complejo y configurable de todos ellos.

5.4.1. El Tile

El tile (o baldosa/azulejo en español) [12], es el componente básico de representación del globo. Este representa una zona de la superficie terrestre comprendida en un sector, es decir, en un rango de latitud y longitud. Nuestra representación de la superficie terrestre estará formada por un conjunto de tiles en memoria que completarán el rango de -180º a 180º en longitud y de -90º a 90º en latitud.

Como se verá más adelante, los tiles están ordenados en función del tamaño de superficie que representan, asignándosele a cada uno un nivel de detalle o LOD (Level of Detail) con el cual representan el terreno que tienen asignados. Cada tile, solicitará recursos online al módulo de descargas para poder construirse, tales como las imágenes que debe mostrar y los mapas de elevaciones.

El tile debe contener por tanto la información referente tanto a la representación geométrica del terreno, a su localización jerárquica con respecto a otros tiles, y a las imágenes que se van a mostrar sobre ellos. Un tile pintado en solitario se vería tal como se muestra en la figura 24.

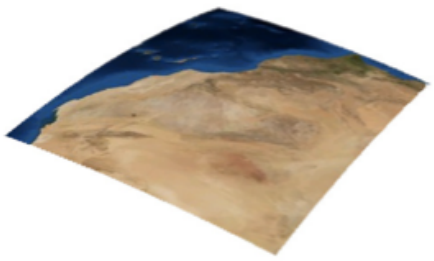

Figura 24 - Tile

En este caso y dada la naturaleza de nuestra aplicación estos tiles serán dibujados como superficies de forma elipsoidal. Sin embargo, ha de hacerse hincapié, en que el concepto de tile es igualmente utilizado en aplicaciones que representan mapas 2D, con la salvedad de la utilización de elevaciones.

Representación Geométrica

El tile de nuestro globo 3D es una malla de triángulos que representa un “parche” rectangular sobre la superficie de un globo. La malla cubre y representa un sector

de coordenadas del elipsoide que es la aproximación matemática de la superficie del globo.

Mediante un conjunto de estos parches que cubra todo el espacio de coordenadas geográficas, obtenemos un modelo teselado del planeta Tierra para nuestra escena [12]. La figura 25 muestra el globo representado con 50 tiles iniciales de resolución 12x12, lo que supone unos 14400 triángulos en la superficie al cargar la aplicación.

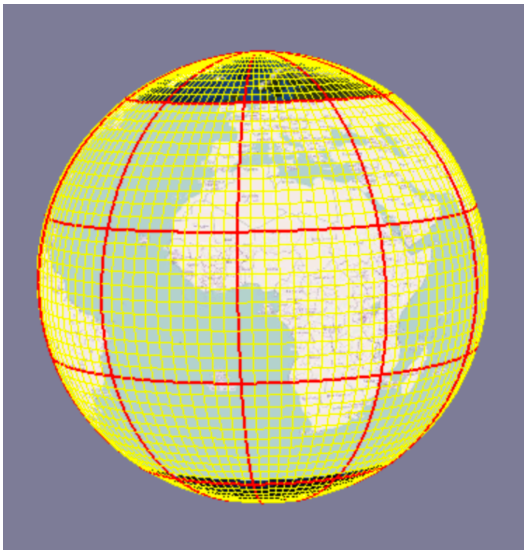

Figura 25 - Vista del globo mallada

Los vértices de las mallas se calcularán como interpolaciones lineales uniformemente repartidas en los rangos de latitud y longitud del tile. Utilizando los métodos matemáticos referidos al elipsoide pasaremos de este espacio en coordenadas geográficas a sus valores correspondientes en el espacio cartesiano. De esta forma obtenemos los valores XYZ de cada uno de los puntos que conforman la malla.

Cada una de estas mallas está definida en base a un valor de resolución, es decir, el número de triángulos verticales y horizontales que la conforman. De esta manera, una malla de resolución 16 estará formada por 16x16 cuadrángulos dibujados con 2 triángulos cada uno, lo que hace una cuenta de 512 triángulos a dibujar.

Este valor es ajustable en función de las capacidades del hardware gráfico del dispositivo. Mallas con mayor resolución implican una mayor fidelidad a la hora de representar el elipsoide pero un número mayor de triángulos para dibujarlos. Un pobre desempeño gráfico del dispositivo, podría obligarnos a rebajar el número de triángulos para no disminuir los FPS de renderizado.

Uno de los detalles a tener en cuenta a la hora de implementar estas mallas, es que las API's gráficas trabajan con valores de 32 bits. Esto no nos otorga mucha precisión en valores cercanos al cero, lo que no es recomendable en un sistema que trabaja con escalas que van de los pocos metros a los miles de kilómetros.

Para procurar que los valores de los vértices estén cercanos al cero, se almacenan siempre referenciados al punto central del tile. Este centro se almacena y se agrega a la matriz de modelo multiplicándola por una matriz de traslación, operación que se realizará en aritmética de 64 bits. La figura 26 muestra la posición del centro de un tile en relación al mallado exterior.

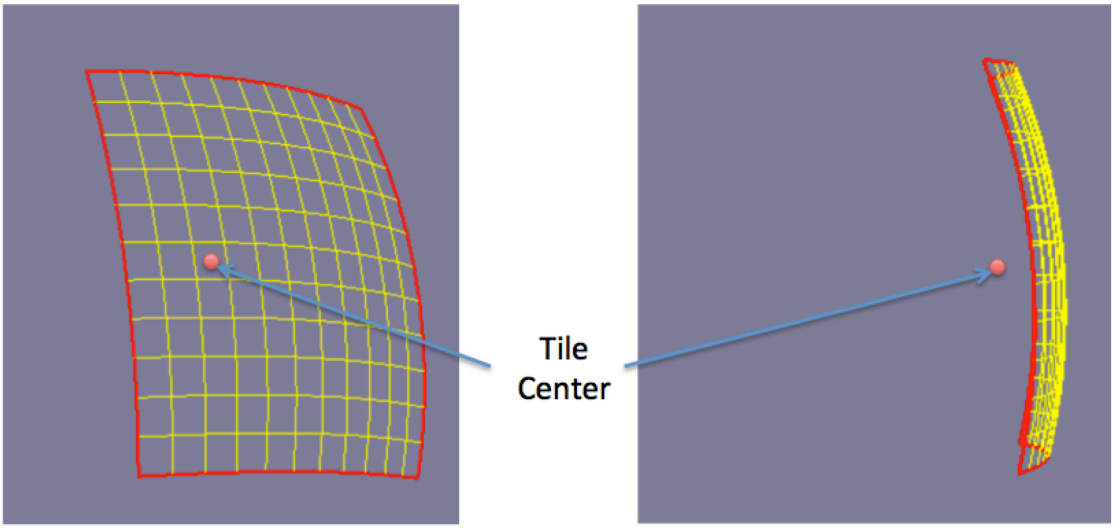

Figura 26 - Posición relativa del centro del tile

Cada triángulo de la malla comparte cada uno de sus vértices con los triángulos adyacentes, lo que en un vértice central se traduce en el mismo vértice replicado en 8 triángulos. Para ahorrar memoria en GPU, en vez de definir los triángulos de forma individual, pasamos independientemente un vector con las posiciones de los vértices que será almacenado en la memoria gráfica. Posteriormente indicaremos como pintar los triángulos utilizando un vector de índices, en el que cada 3 índices determinan los vértices de un triángulo.

En nuestro sistema cada tile cuenta con una única textura que se pintará sobre el cubriéndolo por completo. La forma en la que esta textura se distribuye entre los triángulos que conforman la malla viene dada según unas coordenadas de textura bidimensionales (en el rango 0 a 1) que se asignan a cada uno de los vértices de la malla.

Este vector de coordenadas de vectores, así como el vector de índices, es común para todos los tiles que pintamos. Lo cual implica que no han de recalcularse, y que pueden ser compartidos por todos los tiles, ahorrando memoria en el sistema.

También es necesario recalcar que, debido a problemas de precisión a la hora del cálculo de las coordenadas de tiles, estos pueden no "encajar" de forma perfecta. Este salto entre un tile y su vecino puede ser molesto en la visualización. Para enmascarar estas pequeñas irregularidades se suelen añadir a los tiles unas "faldas" o mayas triangulares que tienen la misma textura y bordean el tile por debajo, como se ve en la figura 27.

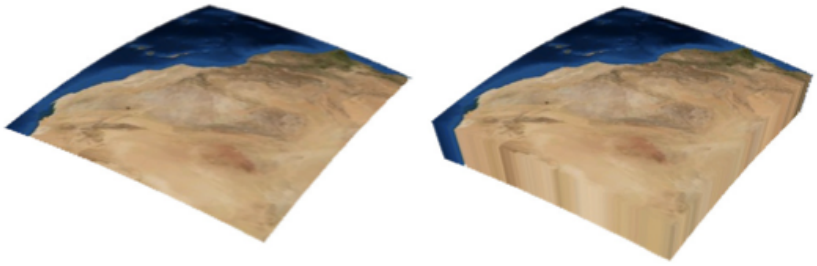

Figura 27 - Tile sin falda y con falda

Mapas multitextura

Como se ha dicho anteriormente, nuestro globo virtual presentará una serie de capas superpuestas, o mapas, que se pintarán sobre su superficie. De esta forma, se pueden representar todo tipo de contenido geolocalizado como mapas satélite, representaciones de alturas, callejeros, campos vectoriales, etc.

Cada tile puede en principio mostrar múltiples imágenes, por lo que parece que la solución del problema está orientada al uso de multitexturas. Sin embargo, hay que recordar que estamos trabajando con dispositivos móviles que cuentan con una memoria gráfica que puede ser muy limitada y que el número de texturas puede ser igualmente muy elevado. Es por esto, que en la implementación se ha decidido otorgar a cada tile una sola textura, que será combinación de las capas que debe mostrar.

Estas se almacenan en el sistema como una lista de capas alojadas en distintos servidores WMS. Cada capa tendrá asignada, además de su ubicación en la red, una serie de parámetros con la que se definirá su estilo, transparencias, sector, etc. Siendo así, es tarea de cada tile determinar que conjunto de imágenes ha de pedir a la hora de construirse.

Para esto se recorre la lista de capas WMS buscando capas cuyo bounding box tenga una intersección con el sector que representa el tile, que serán los sectores que se soliciten al servidor de imágenes. Se sigue el proceso de búsqueda de capas activas, hasta que se llega al final de la lista de capas, o se alcanza una capa opaca que cubre todo el tile. Las imágenes así conseguidas se combinan de abajo hacia arriba, teniendo presente sus respectivos sectores.

En principio los mapas que dibujaremos están proyectados en la llamada proyección Merkator [20]. Esta proyección asegura que la distancia entre dos pixeles de las imágenes es directamente proporcional a la distancia angular existente entre ellos. Otras proyecciones lineales, no garantizan esto, observándose una deformación de los mapas en latitudes altas.

Esto nos permite determinar que las coordenadas de textura, serán en este caso comunes a todos los tiles y se calcularán como una interpolación en el espacio (0,0) – (1,1) de los puntos definidos por los NxN vértices de la malla.

Elevaciones

El tile tal como lo hemos definido hasta ahora, responde geométricamente a la superficie de un elipsoide perfecto. Esto funciona bien para tiles que representen el globo con poco nivel de detalle. Sin embargo, al acercarnos al terreno es deseable que la superficie de los tiles presente "rugosidades" que se correspondan con la orografía real del terreno. De esta forma podemos presentar un verdadero modelo 3D de la Tierra con valles, montañas, llanuras, etc.

El último recurso que un tile necesita antes de pasar poder pintarse en pantalla es información acerca de la altura que presenta cada uno de los vértices del mallado que lo conforma [12]. Esta información vendrá dada en la forma de un mapa de alturas en el que cada pixel del mapa tendrá un valor que será la altura en un punto determinado.

En nuestro caso, hemos decidido utilizar como fuente de esos datos un servidor de alturas de la NASA que devolverá imágenes en formato BIL. En ellas cada pixel se corresponde con uno de nuestros vértices, y contendrá un valor de 16 bits indicando la altura.

A cada pixel de la malla habrá que sumarle por tanto la normal de la superficie del elipsoide en ese punto con módulo la altura.

Tras varias pruebas, hemos determinado que la precisión de los mapas BIL otorgados por la NASA, se degrada en los bordes de las imágenes. Es por eso que nos vemos obligados a pedir imágenes de un tamaño ligeramente superior y recortarlas a posteriori.

5.4.2. Jerarquía de tiles

Ya hemos determinado que el tile es el componente que utilizaremos para representar una parcela comprendida en un rango de coordenadas. Sin embargo, falta por definir cuántos de ellos serán necesarios y de que forma se organizarán para poder representar todo el globo a distinto nivel de detalle.

A la hora de cargar la aplicación, el globo ha de dividirse en un grid de tiles [12]. Al tener que cubrir, 360 grados de longitud y 180 de latitud, generalmente se producen el doble de tiles horizontales que verticales, para que sean igual de anchos que altos en coordenadas geográficas. Una configuración típica es 2x4. Estos tiles son llamados tiles iniciales, existen siempre en el sistema y tienen nivel de detalle 0.

Cada uno de estos tiles iniciales representa una gran parte de la superficie terrestre con un bajo nivel de detalle. Para dibujar el globo 3D por primera vez es necesario contar con los recursos que requieran los tiles iniciales, hasta entonces se ha de mostrar al usuario algún dialogo que le pida que espere a que se finalicen las descargas.

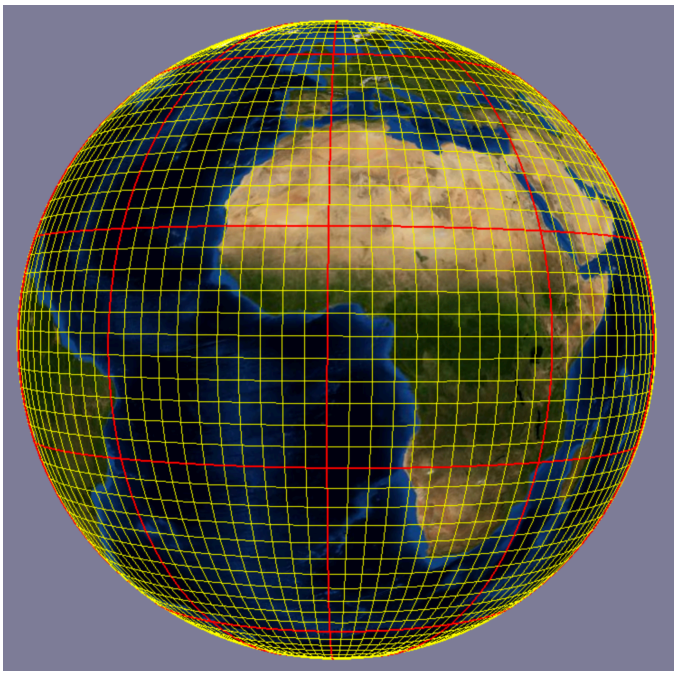

Figura 28 - Imágenes WMS para LOD 0

Para representar la superficie terrestre con más detalle, el usuario ha de acercarse a la zona que desea ver mejor. Esto provocará que el tile que representa esta zona sea sustituido por cuatro tiles, que representan los cuatro sectores resultantes de subdividir el tile centralmente en latitud y longitud.

Podemos considerar al tile mayor como padre de los cuatro tiles menores que lo sustituyen para representar el terreno con más precisión. Si entendemos pues que cada tile que definamos tiene un padre y potencialmente puede tener cuatro hijos, podemos encajar todos los tiles desplegados en una estructura arbórea llamada quad-tree, en el que cada nodo tiene cero o cuatro nodos hijos.

El sustituir un tile por sus hijos, equivale a representar un sector de terreno con cuatro veces más vértices y cuatro veces más resolución de textura. Los tiles hijos representan, por tanto, con mayor fidelidad el terreno y es por eso que tienen un nivel de LOD superior a su padre. En la figura se muestra como al acercarnos a la superficie del planeta se forman nuevos tiles.

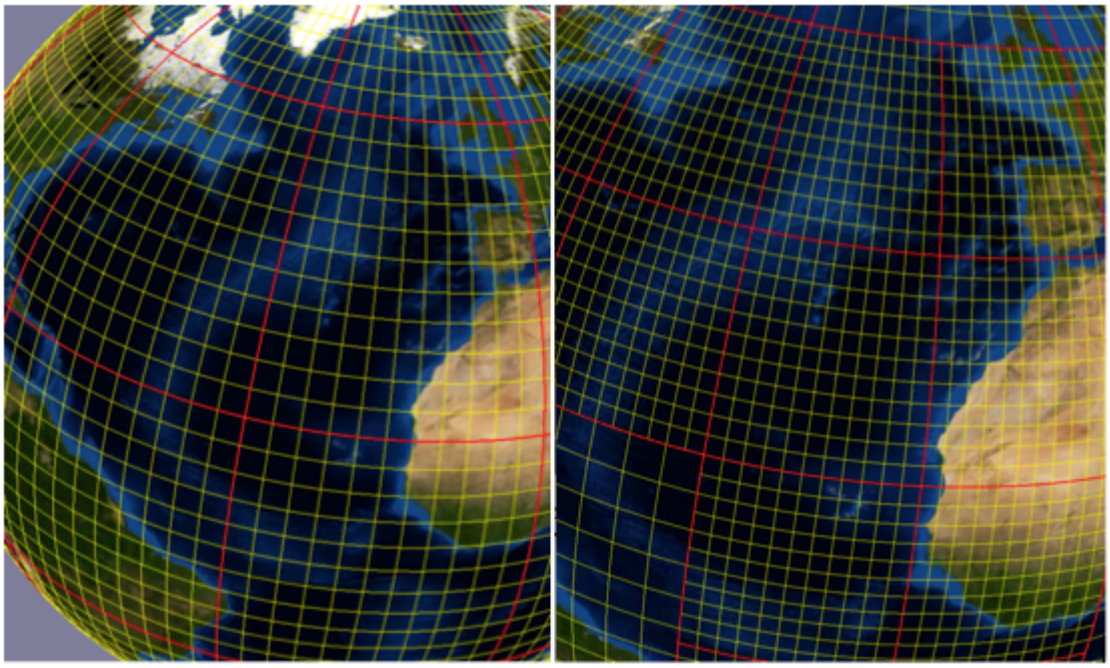

Figura 29 - Subdivisión de tiles LOD 0 y LOD 1

Se debe recalcar en este momento, que cuando hablamos de "eliminar" o "añadir" un tile nos referimos a la inclusión de ese nodo en el dibujado de la escena. Los tiles, una vez creados, existen en memoria y sus recursos, como vértices y textura en GPU, pueden ser liberados si se necesita espacio y no están siendo pintados.

De la misma forma, cuando el usuario aleja la cámara de una zona o dirige su atención hacia otra parte del mapa, los tiles hijos son eliminados y sustituidos por el padre.

Así, conforme nos vamos acercando a la superficie del modelo, se van formando varios quad-tree que contiene todos los tiles necesarios para su representación. En este árbol, el nodo raíz se corresponde con el tile inicial y del mismo solo son visibles los nodos hoja.

Política de crecimiento del quad-tree escalonada

Es el primer algoritmo que diseñamos para la creación y destrucción de los tiles. En él se mantiene una lista de los nodos visibles del quad-tree (nodos hojas) y se procesa su visibilidad en cada frame.

Los tiles que necesiten mostrarse en más detalle crearán sus nodos hojas, los colgarán del árbol y serán sustituidos por ellos en cuanto lleguen sus recursos de red. Estos nodos entran en la lista de tiles visibles y su padre sale.

De forma análoga, cuando se determina que todos los tiles visibles de un mismo padre representan el terreno con demasiada precisión, son eliminados del árbol y el padre vuelve a ser mostrado. Para que estos saltos hacia atrás sean lo más

rápidos posible, no eliminamos de GPU las texturas de los tiles cuyos hijos sean todavía visibles.

En el siguiente gráfico se muestra la evolución del árbol de tiles en un frame. Los nodos hojas (marcados en rojo) son los únicos a los que se aplica el test de LOD, pudiendo tener que agruparse (naranja) o dividirse (violeta). Se puede observar en la figura 30, que el árbol puede replegarse o expandirse a razón de un nivel cada vez y conteniendo todos los nodos en memoria.

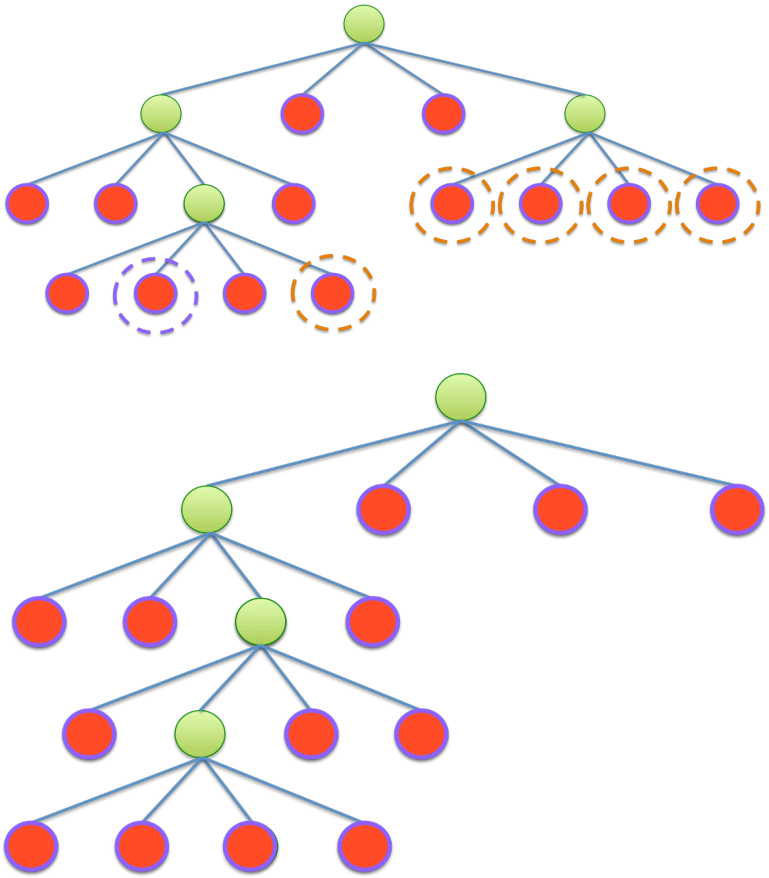

Figura 30 - Modificación del Quad-Tree en un frame

Este modo de funcionamiento asegura que para visualizar un nivel de detalle de una zona, previamente se han mostrado todos los niveles de detalle inferiores, y por tanto, sus recursos han sido cacheados de la red. Asegura a su vez, que en un mismo frame se han de generar un número limitado de nuevos tiles, puesto que el árbol solo crece un nivel por frame. Además, este ritmo de crecimiento asegura que las peticiones de recursos sean más homogéneas, lo cual ayuda a no colapsar el acceso a la red.

Sin embargo, este enfoque obliga a un correcto despliegue de todos los niveles de detalle anteriores al actual. Por ejemplo, si una imagen de nivel 2 no llega, será imposible visualizar ningún tile de nivel 3 que esté contenido en está. Además el despliegue frame a frame del quad-tree hace que los niveles de detalle alto tarden bastante en mostrarse en caso de movimientos rápidos de cámara.

Estas desventajas nos han llevado a plantear otra estrategia.

Política de despliegue total del quad-tree

Este nuevo método nos lleva a la construcción total del quad-tree en cada frame. De esta forma, partiendo de la base de cada uno de los tiles iniciales se van subdividiendo y creando los niveles de detalle necesarios para la representación correcta del modelo. A cada nuevo tile del árbol se le aplica el criterio de visibilidad y si es necesario se subdividirá en sus cuatro hijos de nivel superior a los que se le aplicará de nuevo. De esta forma el árbol en cada frame crece hasta el nivel de detalle que requiere la posición de la cámara.

Para asegurar que en cada frame hay tiempo para la generación de cada uno de los tiles, se utiliza un sistema de caché de tiles, de forma que cuando se va a agregar un tile al árbol primero si ya está creado. De esta forma solo se generan los nuevos tiles que se van a insertar en la escena.

Estos nuevos tiles se mostrarán en el mismo frame de su creación, pidiéndose sus recursos necesarios a la red o extrayéndolos de caché si fuera posible. En el caso de que no se haya recibido su textura aún desde red, los tiles utilizarán una porción de la textura que utilice el tile de nivel de detalle inferior más cercano con una textura cargada. En la figura 31 se aprecia como los tiles que no cuentan aún con datos de textura (marcados en rojo) utilizan la textura de su antecesor texturizado más cercano.

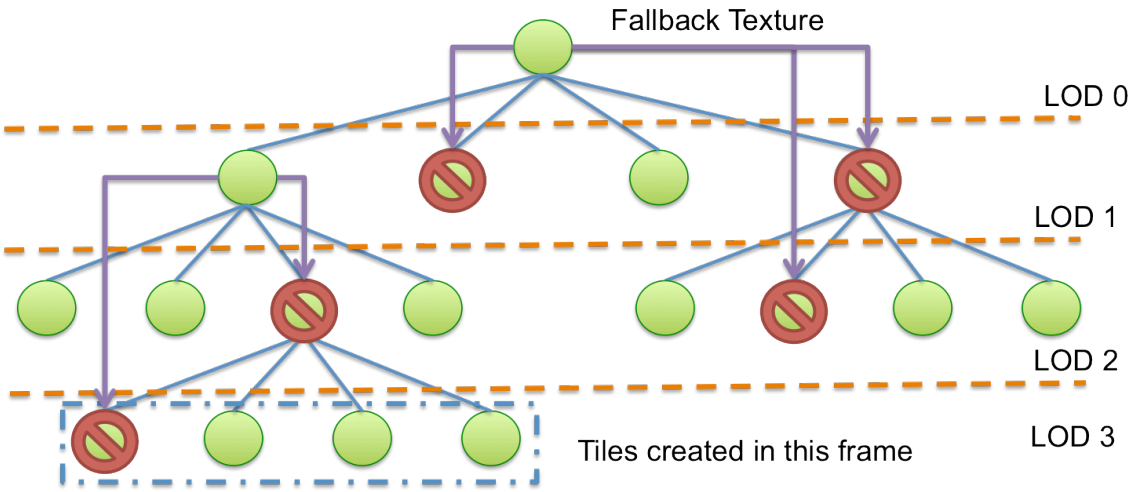

Figura 31 - Árbol construido en un frame al que le faltan recursos

Esto se consigue aplicando un factor de traslación y escalado a las coordenadas de textura originales. Estos factores se calcularán a partir del sector del tile al que pertenece la textura y del sector al que se va a aplicar, tal como se ve en la figura 32.

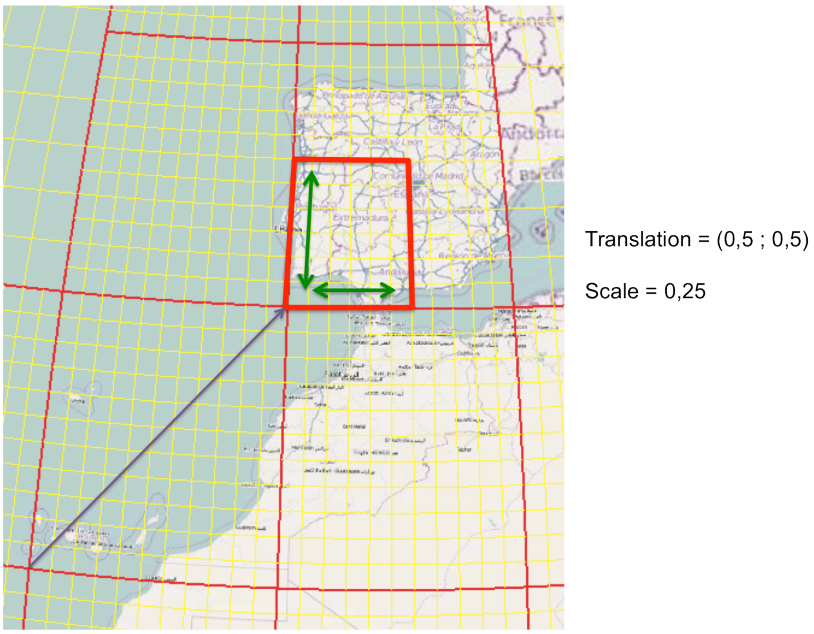

Figura 32 - Traslación y escalado de las coordenadas de textura

Este sistema implica que en la misma escena pueden llegar a observarse tiles con texturas de gran nivel de detalle, con otros que todavía están esperando la llegada de su textura y muestran una de menos detalle.

Sin embargo, y como contrapartida, los movimientos rápidos de cámara permiten reflejar de forma rápida los incrementos en nivel de detalle, puesto que en un solo frame se puede saltar de un nivel de LOD a cualquier otro. Este enfoque también permite liberar el acceso a red, puesto que, al saltar en un frame de un nivel de LOD a otro superior, no debemos preocuparnos en pedir texturas para los tiles de niveles intermedios.

5.4.3. Test de LOD

Para entender finalmente como es el despliegue de los tiles sobre el terreno falta explicar el funcionamiento de una pieza fundamental: el test de LOD [12]. Este algoritmo determina en función de la geometría de un tile y de la posición actual de la cámara, si este representa con un nivel de detalle adecuado la superficie del planeta.

En nuestro modelo de funcionanmiento del tile renderer, este test es un método que puede devolver tres estados para un tile determinado:

- KeepLOD: El nivel de detalle es ideal para la representación del terreno que el tile abarca a esa distancia del espectador.
- IncLOD: El nivel de detalle se ha de incrementar. El tile ha de ser sustituido por sus hijos.
- DecLOD: El tile representa con demasiado detalle la superficie del planeta y puede ser sustituido por su padre, sin que el usuario de la aplicación aprecie una perdida de calidad visual.

Durante el desarrollo de este proyecto hemos generado dos tipos de test LOD que se explican a continuación.

Test LOD por posición de cámara

Este algoritmo comienza estimando cual es el punto central de la vista en el terreno (CPV). Cuando la dirección de vista es normal a la superficie, este se encuentra en el centro de la pantalla y conforme la cámara se abate, este punto se desplaza hacia la parte baja de la pantalla, tal como se muestra en la siguiente figura. La proyección de ese punto en el terreno será la posición que deba mostrarse a mayor detalle. La evolución del CPV en la pantalla se muestra en la figura 33.

Figura 33 - Punto central de la vista en diferentes configuraciones

Después de esto, hemos de calcular la distancia geodésica que separa el centro del tile (T) del punto CPV. Siendo d_T dicha distancia y w_T el ancho del tile en cuestión, D la distancia de la cámara al CPV y R el radio de la Tierra, podemos establecer la siguiente formulación:

$$\varepsilon(T) = \varepsilon_{\max} + (\varepsilon_{\min} - \varepsilon_{\max}) \frac{w_T - D/4}{\pi R - D/4}$$

De esta forma $\varepsilon(T)$ es un valor acotado entre ε_{min} y ε_{max} que depende del ancho del tile. $\varepsilon(T)$ es la relación existente entre el ancho del tile y su distancia al centro de vista, siendo máxima cuando el tile es pequeño. Experimentalmente, se ha decidido fijar esos umbrales en 0.9 y 1.1 respectivamente.

El tile T habrá de subdividirse en el caso de que se cumplan dos condiciones:

- La primera establece que la relación entre wT (el ancho del tile) y dT (la distancia del tile al CPV) no debe superar el valor de ε.

$$\frac{d_T}{w_T} < \varepsilon$$

- La segunda no permite que el tile que contiene el CPV se divida indefinidamente, estableciendo un nivel máximo de LOD.

$$w_T > D/4$$

Este algoritmo para determinar si un tile ha de subdividirse o no se basa únicamente en su relación con el punto CPV. Esto repercute en que la disposición de los tiles sea simétrica con respecto a la posición de la cámara. De esta forma se muestra el mismo nivel de detalle delante o detrás de la cámara, lo cual es bueno en escenarios donde la cámara pueda rotar con rapidez.

Test LOD basado en área proyectada

A parte del algoritmo antes comentado, existen otras aproximaciones al problema del test LOD. En este proyecto, hemos decidido implementar una de las estrategias más clásicas, basada en realizar una estimación del área proyectada del tile en el visor.

El primer paso de nuestro algoritmo es determinar que el tile es visible para la configuración actual de cámara. La forma más rápida de descartar la mayoría de los tiles es realizar el test de inclusión en el frustum tal como se describe en la sección dedicada al Camera Renderer.

En el segundo paso determinaremos si el tile no está oculto tras otros tiles que se encuentren más cerca. Una superficie visible será aquella que tenga en algún punto un vector normal que apunte en dirección al espectador. Tomando como aproximación a la superficie del tile una sección de elipsoide, podemos determinar que un tile está orientado hacia nosotros si el vector normal de alguna de sus esquinas lo está. Estos test iniciales se pueden ver en la figura 34.

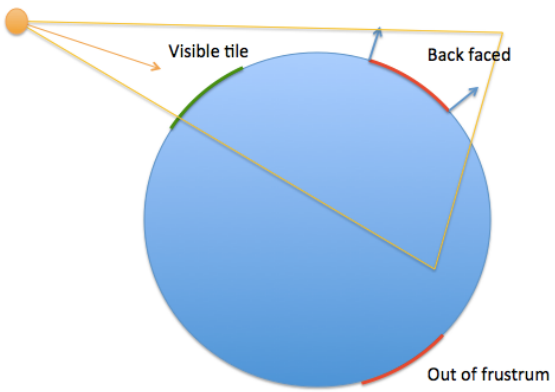

Figura 34 - Tests previos al cálculo de LOD

Una vez hemos determinado que el tile cumple ambos requisitos de visibilidad, queda determinar si la superficie que proyecta en pantalla tiene un tamaño proporcional a su nivel de detalle. Para esto proyectamos los vértices de las esquinas del tile sobre la superficie de renderizado.

De esta forma tenemos en coordenadas de pantalla los vértices que vemos como limites del tile. Posteriormente calculamos la circunferencia de tamaño mínimo que contiene todos los vértices. Esta circunferencia estará centrada en el punto medio del área encerrada por los cuatro vértices y tendrá como radio la distancia de este punto al vértice más lejano. En la figura 35 se puede observar la construcción de esas circunferencias contenedoras dependiendo de la posición del tile.

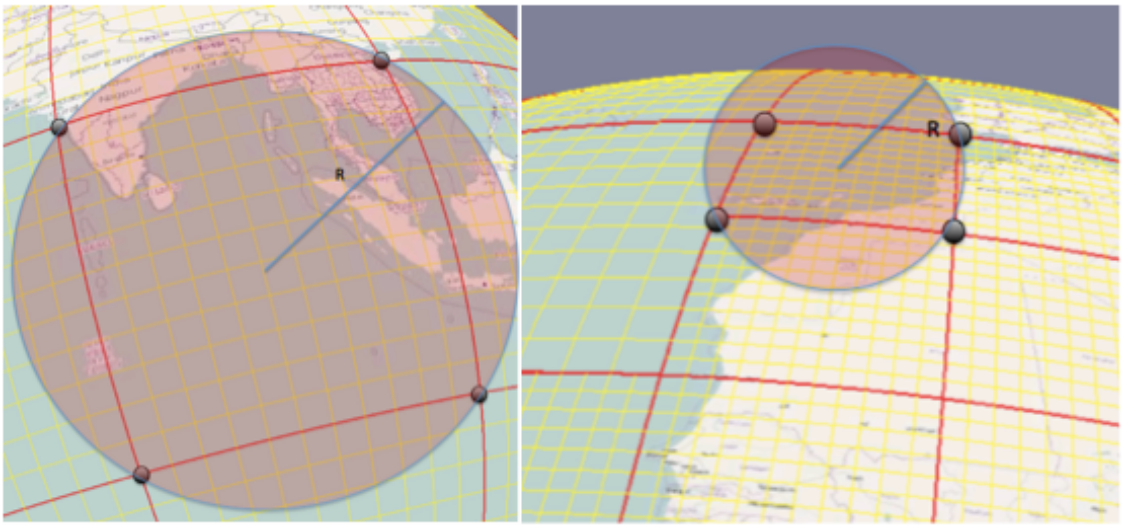

Figura 35 -Circunferencia de area proyectada

El círculo encerrado en esta circunferencia representa una aproximación al alza de la superficie proyectada en pantalla por el tile. Al estar en coordenadas de pantalla (de 0 a 1) podemos calcular fácilmente cual es el número de pixeles que encierra, sabiendo que el viewport tiene una resolución de w x h.

$$P(T) = \pi R^2 * w * h$$

Si estamos utilizando texturas cuadradas de w_T x h_T píxeles podemos determinar que el tile ha de subdividirse siempre que se cumpla la condición:

$$P(T) > w_T * h_T * \lambda$$

Donde lambda es un valor superior a 1. Cuanto mayor sea lambda, menor será la calidad gráfica obtenida del modelo.

Existen otras técnicas similares, basadas en la proyección del tile. En unas se calcula la diagonal mayor o se toma la proyección del cuadrilatero formado por las puntas del tile, otras consisten en construir un contenedor (cubico, elipsoidal…) que contenga el tile, y calcular el área de su proyección. Sin embargo, tras las pruebas nos hemos decantado por la circunferencia, pues ofrece resultados similares con un coste computacional menor.

Este sistema, a diferencia al Test de LOD por posición de la cámara, no tiene en cuenta los tiles que se despliegan por detrás de la misma, que pasan al nivel mínimo de LOD.

Esto genera árboles de tiles más desequilibrados pero con un número menor de tiles. Para rotaciones rápidas de cámara es, por tanto, preferible utilizar algoritmos de crecimiento rápido del quad-tree, como el de despliegue total, que permitan que ante movimientos bruscos se alcancen rápidamente los niveles más altos de LOD.

5.5. Marks Renderer

Hasta ahora se han tratado los renderers indispensables para la correcta visualización del modelo de terreno. Sin embargo, la idea de generar renderers independientes para dibujarse conjuntamente permite añadir elementos extra que se integren en la escena.

Como ejemplo de renderer integrable en el motor, hemos desarrollado un pequeño complemento que permite mostrar marcas en el terreno. Estas marcas pueden ser útiles en un sin fin de aplicaciones, pudiendo señalar la posición de sitios de interés, waypoints, señalar la posición actual del usuario, etc.

El Marks Renderer contiene una lista de marcas en las que el usuario o el programador pueden añadir más dinámicamente. De forma genérica, cada marca contiene:

- Una posición. Indicada en coordenadas geográficas y una altitud.
- Un nombre.
- Una descripción.
- Una textura con la que mostrarse. Suele tratarse de una imagen pequeña y con transparencias.

La marca así definida, se ha de visualizar como un icono que mantenga su orientación hacia el observador (visualización BillBoard), mostrándose en el mismo punto fijo sobre la superficie del planeta, y manteniendo su tamaño y orientación en el viewport.

5.5.1. Visualización de las marcas

Antes de efectuar el primer renderizado de las marcas sobre el terreno, el renderer ha de asegurarse de que todas las texturas que va a utilizar están cargadas en GPU. Es común que muchas marcas se visualicen con el mismo dibujo en pantalla. Es por eso que el renderer ha de asegurarse de que no carga varias veces el mismo archivo a GPU, sino que reutiliza el identificador de textura para las marcas iguales.

Una vez los recursos a utilizar están disponibles para su uso, pasamos al proceso de pintado. Para ello es importante hay que tener en cuenta que las marcas están pegadas a la superficie del suelo pero siempre han de estar completamente visibles sobre el mismo. La técnica ha utilizar para ello es ejecutar el renderizado de las mismas después de la ejecución del Tile Renderer y desactivando el test de

profundidad en el pipeline gráfico. De esta forma las marcas siempre son completamente visibles.

En la figura 36 se muestra por qué es importante desactivar el test de profundidad a la hora de pintar las marcas, y de esa forma, evitar que el terreno pueda solaparse con las mismas.

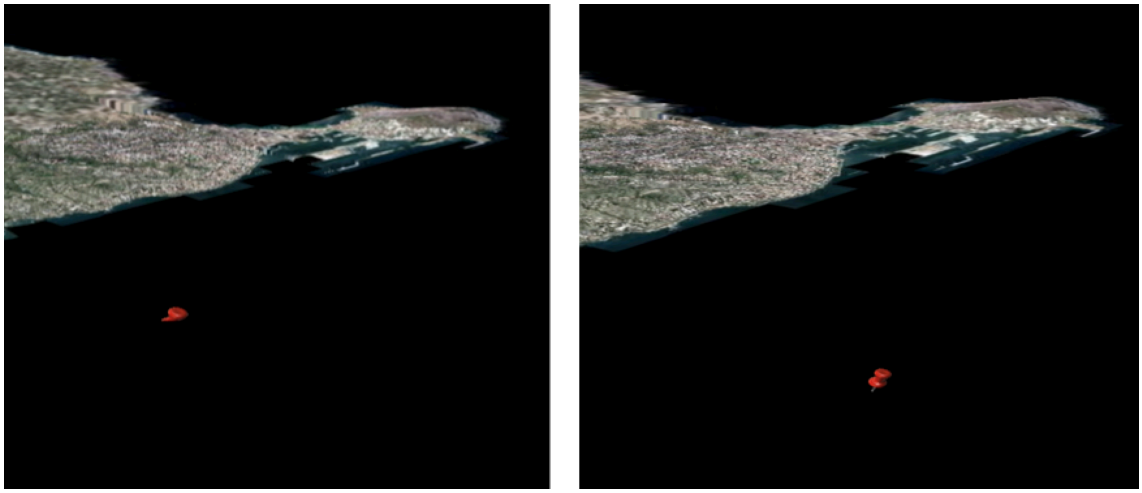

Figura 36 - Marca renderizada con y sin test de profundidad

Como se ha comentado antes la marca ha de mantener en todo momento su proporción con respecto al área de pantalla y debe de estar orientada hacia el espectador (billboarding [17]). Podría plantearse un modelo de visualizado en el que las coordenadas de los vértices de las esquinas de la marca se calculan en función de la posición de la marca y de la posición de la cámara. Sin embargo, hemos preferido utilizar un sistema mucho más simple y computacionalmente eficiente.

Para ello, únicamente calculamos la posición central de la marca, como la traslación al espacio cartesiano de su latitud, longitud y altura (P_M). Como coordenadas de textura utilizaremos siempre las esquinas de la textura asignada a la marca, es decir T={(0,0), (0,1), (1,0) y (1,1)}. La figura 37 muestra una marca, la posición en la que se encuentra y sus esquinas.

Figura 37 - Esquinas y punto central de una marca

A la hora de renderizar la marca, se pasará al pipeline gráfico el vector de las cuatro esquinas de la marca como si todas estuvieran en el punto central, es decir, $\{P_M, P_M, P_M, P_M\}$.

El pipeline, de la forma habitual, trasformará estás coordenadas al espacio de pantalla, generando $\{P_M', P_M', P_M', P_M'\}$. La diferencia radica ahora, en que indicaremos al pipeline que posteriormente al cálculo de estas coordenadas ha de trasladarlas en el espacio de pantalla utilizando las coordenadas de textura que le hemos indicado.

$$P'_{Mi} = \alpha * T_i * P'_M$$

Donde P'_M es el punto central en pantalla de la marca, T_i es coordenada de textura para la esquina i, y P'_{Mi} es la coordenada en pantalla final para esa esquina. Finalmente α es un valor pequeño (por ejemplo 0.1) que indica el porcentaje de pantalla que ocupará la marca.

Finalmente hay que decir que, para viewports no cuadrados, las coordenadas finales de dibujo de la marca han de ser multiplicadas por la relación ancho/alto de la superficie de dibujado. De otra forma, la marca tendrá la relación de aspecto del viewport en vez de ser cuadrada.

6. Aspectos relevantes de la implementación

6.1. Arquitecturas de los sistemas operativos móviles actuales

6.1.1 Android

El sistema operativo móvil más ampliamente extendido a día de hoy es Android, el sistema desarrollado por Google que cubre casi la mitad de la distribución de mercado según algunos analistas. Esta importancia, mostrada en el gráfico de la figura 38 [21], está en aumento, conforme el sistema Android y su hardware asociado mejora comparativamente a otras marcas.

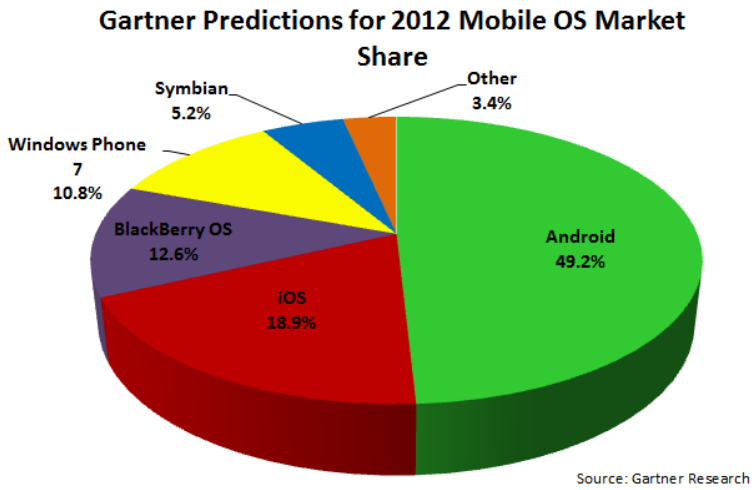

Figura 38 - Importancia de la plataforma Android dentro del mercado móvil

Esto se debe principalmente a que es un sistema operativo abierto a ser utilizado en numerosas plataformas de diversos fabricantes, que de esta forma, obtienen un sistema operativo compatible con un amplio espectro de aplicaciones ya desarrolladas y acceso a un buen número de servicios en la nube que ofrece Google de manera gratuita.

Cada fabricante es responsable de que las diferentes distribuciones del sistema operativo y sus actualizaciones sean compatibles con sus terminales. Esto provoca el consecuente problema de fragmentación del mercado. Las diferentes versiones hasta la fecha de Android y su integración real en el mercado se puede ver en la figura 39.

Version	Codename	API	Distribution
1.5	Cupcake	3	0.1%
1.6	Donut	4	0.3%
2.1	Eclair	7	3.1%
2.2	Froyo	8	12%
2.3 - 2.3.2	Gingerbread	9	0.3%
2.3.3 - 2.3.7		10	53.9%
3.1	Honeycomb	12	0.4%
3.2		13	1.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	25.8%
4.1	Jelly Bean	16	2.7%

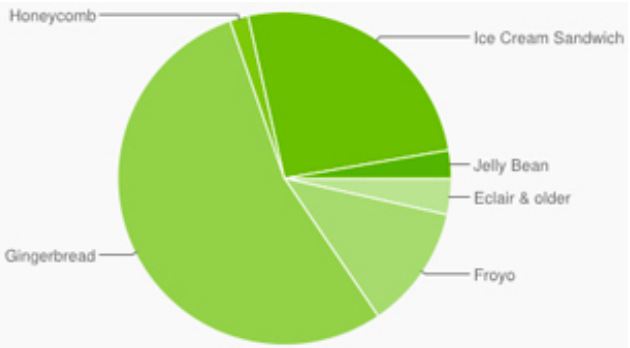

Figura 39 - Distribución de versiones de Android

Un desarrollador debe elegir a partir de cual nivel de API su aplicación será soportada y dejar fuera a los potenciales clientes que no la soporten. En nuestro caso, viendo que la Gingerbread (API 10) es la distro más utilizada y que soporta las funcionalidades que requerimos nos hemos decantado por esta.

La arquitectura de Android [18], vista en la figura 40, está compuesta de:

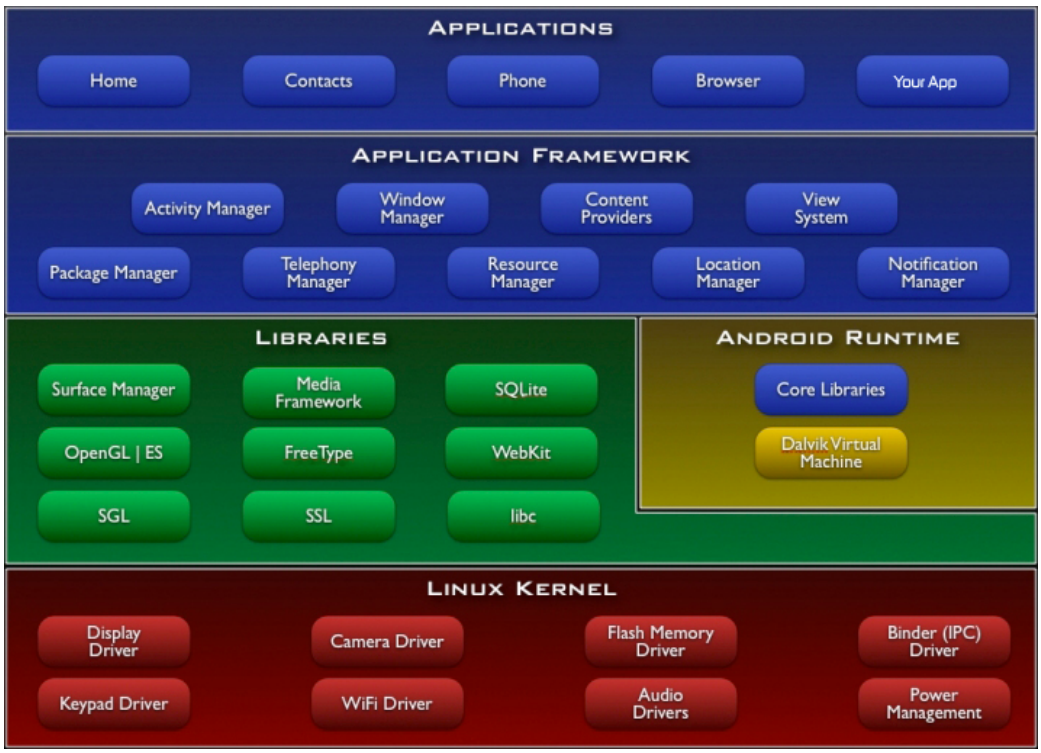

Figura 40 - Esquema del sistema operativo Android

- Las aplicaciones. Tanto las incluidas por defecto con el sistema como las instaladas a posteriori.

- Un conjunto de distintos frameworks que dan acceso a los servicios del terminal. (Contactos, teléfono, recursos,…).
- Un conjunto de librerías accesibles para el desarrollador. Muchas de ellas son comunes a otros entornos de desarrollo java.
- El Runtime, que incluye una máquina virtual Dalkiv. Dalvik está optimizada para requerir poca memoria y está diseñada para permitir ejecutar varias instancias de la máquina virtual simultáneamente, delegando en el sistema operativo subyacente el soporte de aislamiento de procesos, gestión de memoria e hilos. A menudo Dalvik es nombrada como una máquina virtual Java, pero esto no es estrictamente correcto, ya que el bytecode con el que opera no es Java bytecode. Sin embargo, la herramienta dx incluida en el SDK de Android permite transformar los archivos Class de Java compilados por un compilador Java al formato de archivos Dex.
- Finalmente un Kernel de Linux (actualmente el 2.6) que soporta la ejecución de las distintas máquinas virtuales que se ejecutan concurrentemente y contiene los drivers específicos de dispositivo que permiten el acceso a los recursos hardware del mismo.

Como se puede ver esta arquitectura está pensada explícitamente para que con solo alterar el núcleo Linux que trabaja por debajo el sistema sea portable a distintas plataformas.

6.1.2. iOS

El caso de iOS [19] es ligeramente diferente debido a que todo el producto (software y hardware) es controlado y diseñado por Apple. En este sistema un desarrollador solo puede generar aplicaciones que se colocan en la capa superior de todo el sistema software. La figura 41 muestra la estructura en capas de iOS.

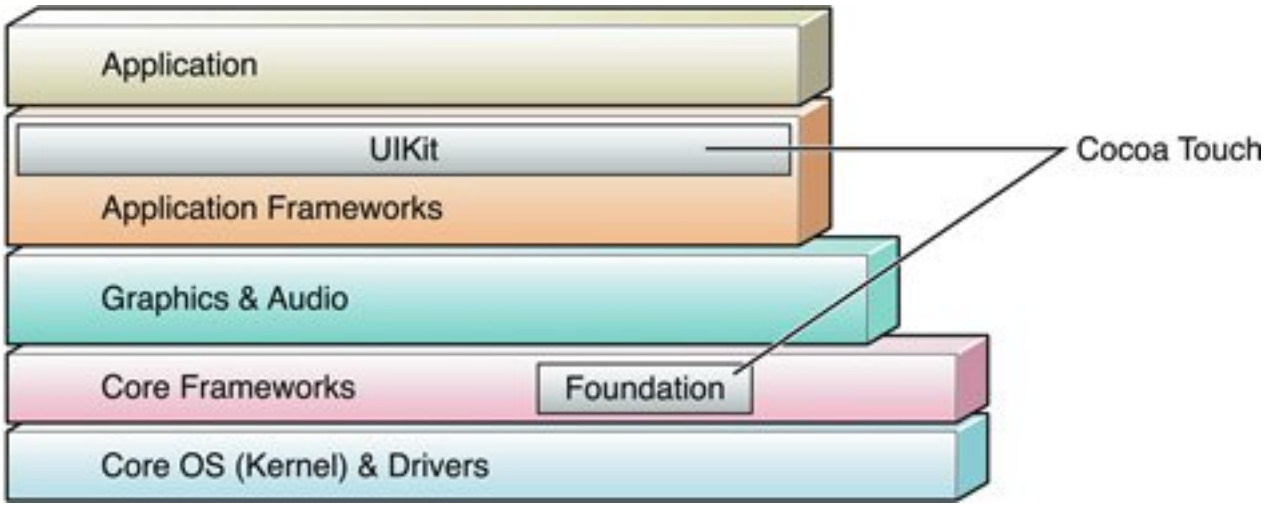

Figura 41 - Esquema del S.O. iOS

Toda la interfaz gráfica de nuestra aplicación, así como el procesamiento de eventos táctiles se realizará a través de las librerías UIKit (desarrolladas en Objetive-C) que forman parte de Cocoa Touch.

Cocoa es un framework de desarrollo de interfaces utilizado en los sistemas iOS y OS X. Las librerías Foundation funcionan a más bajo nivel y permiten realizar tareas como generación de gráficos bidimensionales.

Otro framework destinado a OpenGL (en nuestro caso OpenGL ES 2.0) nos permitirá acceder a los drivers de bajo nivel para generar gráficos 3D. En el caso de Apple, al controlar ellos el hardware, no hay necesidad de una máquina virtual intermedia.

Otra ventaja del modelo "jardín cerrado" proporcionado por Apple es que no existe una segmentación notoria del mercado en función de niveles de API como veíamos en Android. Un desarrollador de aplicaciones iOS puede empezar a trabajar a partir de la API actual, con seguridad de que un amplio porcentaje de los terminales serán compatibles.

6.2. API Gráfica

Para el desarrollo de cualquier aplicación gráfica, la primera decisión que ha de tomarse al respecto es cual va a ser el motor gráfico que se utilizará para el desarrollo de la misma. En el mercado existen numerosos motores que hacen más sencilla la labor del programador gráfico, que no tiene que trabajar a bajo nivel y permitiendo entre otras cosas soporte multiplataforma.

Sin embargo, en el sector móvil, esta oferta se ve mucho más limitada. Esto unido a los problemas de compatibilidad entre múltiples plataformas hacen preferible el trabajo a bajo nivel mediante llamadas a la API gráfica, lo que además, nos permite un control mayor sobre el rendimiento gráfico en cada hardware.

En la mayoría de los sistemas móviles actuales, el estándar de facto para la generación de gráficos 3D es OpenGL en su variante para sistemas empotrados.

OpenGL ES (OpenGL for Embedded Systems) [22] es un subconjunto de la API gráfica OpenGL diseñada para dispositivos integrados tales como teléfonos móviles, PDA's y consolas de videojuegos. Existen varias versiones de la especificación OpenGL ES. La versión 1.0 está basada en OpenGL 1.3; OpenGL ES 1.1 se define en relación a OpenGL 1.5 y OpenGL ES 2.0 parte de OpenGL 2.0.

En estos momentos, la mayoría de los dispositivos móviles en el mercado cuentan con soporte para OpenGL ES 2.0 [23]. Esto unido a su mayor flexibilidad y a que cuenta con todas las características necesarias para nuestro motor, son las principales razones que lo han convertido en la API gráfica que utilizaremos. Existe, a día de hoy, una especificación en desarrollo para OpenGL ES 3.0, pero es relativamente similar a la versión anterior y no cuenta con muchos ejemplos de uso aún ni dispositivos que la soporten.

6.2.1. OpenGL ES 2.0

OpenGL ES es una API multiplataforma con total funcionalidad para gráficos 2D y 3D en sistemas integrados como consolas, electrodomésticos y vehículos. Está compuesto por varios subconjuntos del OpenGL para sobremesa, formando una interfaz a bajo nivel entre el software y el hardware de aceleración gráfica.

OpenGL ES 2.0 permite los gráficos 3D completamente programables [16]. Cada versión de OpenGL define un graphic pipeline o rendering pipeline, que define etapa por etapa, como los datos de entrada de una escena tridimensional se convierten en una imagen raster bidimensional. En esta versión encontramos un pipeline gráfico programable con la habilidad de crear shaders y programas con las que definir Vertex Shaders y Fragment Shaders en el OpenGL ES Shading Language (GLSL).

Un shader es un programa que se ejecuta en el hardware gráfico y que generalmente se utiliza para el cómputo para niveles de iluminación en la imagen, aunque en los pipelines actuales puede producir cualquier clase de efecto o postprocesado.

El Vertex Shader es una herramienta capaz de trabajar con la estructura de vértices de figuras modeladas en 3D, y realizar operaciones matemáticas sobre ella para definir colores, texturas e incidencia de la luz. Esto da libertad a los programadores para realizar diferentes efectos, desde la deformación de un objeto hasta la recreación de las olas del mar.

El Fragment Shader es un programa ejecutado para cada pixel de la imagen y que genera el sombreado de la misma. En el caso de OpenGL este shader se utiliza también para el mapeo de las texturas sobre los polígonos de la escena.

Ha de decirse, que el pipeline gráfico de OpenGL ES 2.0 viene con unas funcionalidades de shading definidas por defecto. Sin embargo, en el caso del proyecto que nos ocupa nos conviene reescribir estos programas para que puedan realizar aplicaciones específicas, como en el caso de la renderización de BillBoards.

El pipeline de OpenGL ES 2.0 se representa gráficamente como en la figura 42.

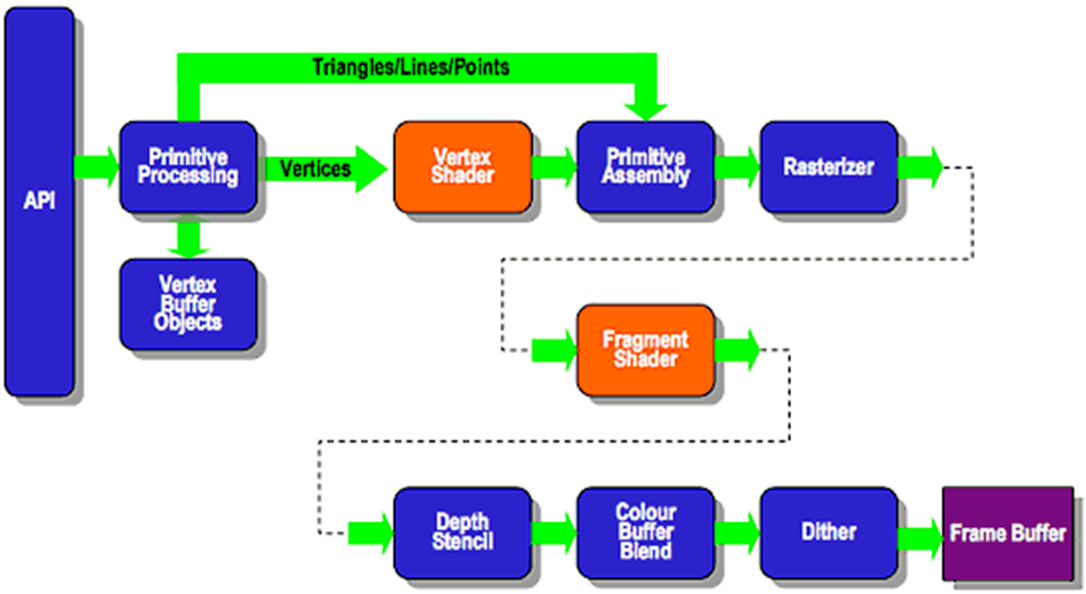

Figura 42 - Pipeline de OpenGL ES 2.0

Donde los módulos naranjas son compilados en tiempo de ejecución a partir de ficheros de texto fuente que serán parte de nuestro proyecto. Cada uno de los shaders tiene predefinidos unos valores de entrada y de salida que encajan con su funcionalidad dentro del pipeline.

El lenguaje para programar estos módulos es GLSL [16] que es el acrónimo de OpenGL Shading Language (Lenguaje de Sombreado de OpenGL), también conocido como GLslang, una tecnología parte del API estándar OpenGL, que permite especificar segmentos de programas gráficos que serán ejecutados sobre el GPU. Su contrapartida en DirectX es el HLSL. GLSL es un lenguaje de sombreado de alto nivel basado en el lenguaje de programación C.

6.3. Desarrollo multiplataforma

Uno de los pilares de este proyecto es que el software obtenido ha de ser multiplataforma. Esto implica que muchas de las funcionalidades de bajo nivel que requiere el motor (tratamiento de imagen, acceso a la red, sistema de ficheros…) sean implementadas con diferentes API's dependiendo de la plataforma en la que nos encontremos.

Para solventar este problema, hemos decidido dividir el software en dos grupos de clases bien diferenciados. Por un lado tenemos un Núcleo (Core) del software común a todas las plataformas. Este core será programado en C++ para la plataforma iOS y se traducirá a Java para Android utilizando aplicaciones de traducción automática (figura 43). Por otro lado, las funciones básicas que realiza el dispositivo a bajo nivel mediante sus API's específicas estarán encapsuladas en clases con una implementación propia por plataforma.

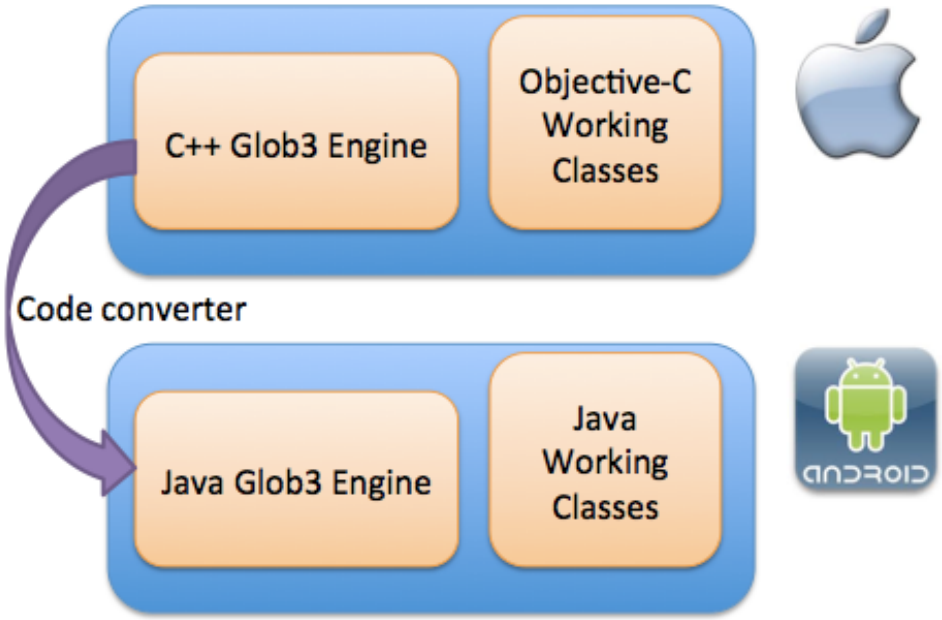

Figura 43 - Modelo de desarrollo para multiplataforma

De esta forma aseguramos tener la misma funcionalidad en todos los dispositivos, siendo gran parte del trabajo de diseño mantener la mayor parte posible en la zona de código común.

Para que el código común sea capaz de utilizar las clases específicas, se establecen interfaces comunes conocidas por el core. Dichas interfaces serán implementadas en clases específicas. Para ello a cada motor se le pasará como parámetro en su declaración un objeto factoría (Factory) que generará instancias específicas de los objetos correspondientes a cada interfaz.

Como cada plataforma cuenta con un objeto Factory propio, el core dispone a su vez de una interfaz para realizar el acceso a la misma. Es lo que se denomina el patrón Abstract Factory [14], mostrado en la figura 44.

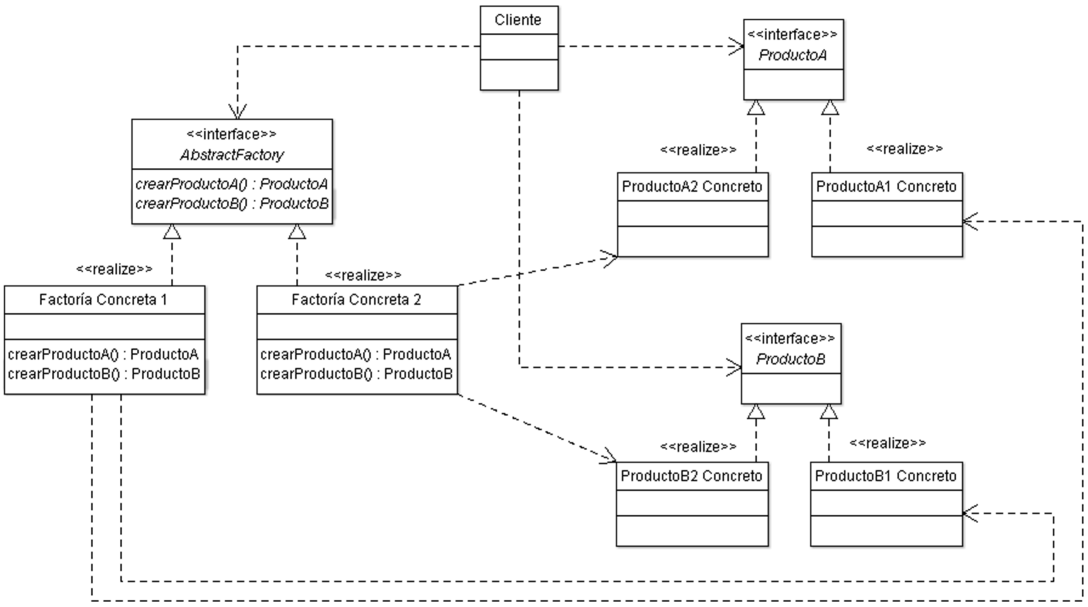

Figura 44- Patrón Abstract Factory

6.4. Implementación de las clases específicas

Como hemos dicho cada plataforma contará con objetos específicos que utilizarán las API's del sistema operativo para acceder a la funciones básicas. Para información completa acerca de las API's utilizadas en ambos sistemas operativos, conmino al lector a la lectura de la referencia online de ambos, que cuenta con numerosos ejemplos de uso [18] [19].

Queda ahora definir cuales serán esos objetos y como se van a implementar sus correspondientes instancias iOS y Android. Las clases específicas identificadas son las siguientes:

6.4.1. Clase Imagen

Entidad que encapsula una imagen. El tratamiento de imágenes es una de las Es una clase útil para el tratamiento de imágenes que serán usadas como textura, conteniendo funciones como:

- Descompresión de imágenes en formato comprimido JPEG, PNG, BIL, BMP…
- Recorte.
- Redimensionado.
- Combinación de imagen.
- Cambios de espacio de color.
- Representación y exportación a vector de píxeles.

En la plataforma iOS la implementación utiliza la clase UIImage, mientras que en Android la clase específica que cumple estas funciones es Bitmap. Ambas contienen funciones que permiten implementar estas funcionalidades, en muchos casos utilizando aceleración hardware.

6.4.2. Clase Network

Objeto utilizado para recibir datos de red (principalmente imágenes WMS en nuestro caso). La implementación requiere el poder realizar peticiones síncronas, ya que el asincronismo lo aportan los diferentes hilos con los que trabaja el módulo de descargas. Sus funciones son:

- Lanzar petición.
- Cancelar Petición.
- Conocer estado de la petición. Entre ellos se encuentra En Espera, Finalizada Correctamente, Finalizada Fallida y Cancelada.
- Obtener datos. La clase Network podrá devolver los objetos en formato RAW (array de bytes) o en formato Imagen por cuestiones de eficiencia.

En el caso de iOS las clases base utilizadas son varias, entre ellas NSURL y NSURLConnection, que permiten realizar peticiones HTTP. Para Android las clases homólogas serán ConnectivityManager, URL, URLConnection e InputStream.

6.4.3. Clase Timer

Permite medir intervalos de tiempo, así como conocer la hora del sistema. Sus métodos más importantes serán:

- Fijar tiempo de inicio.
- Obtener tiempo transcurrido desde inicio.
- Obtener tiempo actual.

En iOS la implementación se basa en la clase NSDate y NSTimeInterval, mientras que en Android se utiliza SystemClock.

6.4.4. Clase SQLiteStorage

La clase SQLiteStorage hace de interfaz al módulo de almacenamiento con la API específica de la plataforma referente a la creación de base de datos SQLite. SQLite es una librería de software autocontenida que permite crear bases de datos sin necesidad de configuración o servidores externos.

Las bases de datos son transaccionales utilizando, como es lógico, SQL como lenguaje de comunicación. SQLite es el motor SQL más extendido en el mundo y u código fuente es libre, lo que propicia que sea implementado en un gran número de plataformas entre las que se encuentran iOS y Android.

En iOS las clases utilizadas de la API son SQDatabase y SQResultSet que hacen referencia a la base de datos en si y al conjunto de resultados obtenido por una query respectivamente.

En Android la clase base que sirve para instanciar la base de datos es SQLiteDatabase, mientras que los resultados son accesibles mediante una instancia de Cursor.

6.4.5. Clases Gtask y ThreadUtils

Como hemos visto, el motor requiere de la ejecución de tareas en hilos diferentes al del renderizado para no disminuir los FPS cuando se requieran procesos pesados como la descompresión y procesamiento de imágenes. Estás tareas se encapsularán en la clase genérica Gtask, que contiene una función de execute.

La clase ThreadUtils nos permitirá ejecutar estas tareas en segundo plano o insertarlas de nuevo en el hilo de renderizado para que se ejecuten tras la ejecución del frame.

En Android para la ejecución asíncrona de tareas se utiliza la clase ThreadPoolExecutor que pone a nuestra disposición un conjunto de hilos secundarios que se pondrán a ejecutar las tareas conforme haya tiempo para ellas. Para la reinserción de tareas en el hilo de render la clase GLSurfaceView. Renderer (responsable del bucle de renderizado) dispone de métodos por defecto que encolan tareas a realizar tras el proceso de generación del frame.

De forma similar en iOS contamos con la clase NSOperationQueue encargada de la ejecución de tareas en segundo plano. Para la ejecución de métodos en el hilo principal recurriremos al framework dispatch y al método dispatch_async. Esté método integra en una cola el bloque de ejecución que queramos asociada al hilo de rendererizado.

6.4.6. Clase GL

Quizá la clase específica más extensa de todas. Supone un wrapping de todas las funciones de OpenGL ES 2.0 que vayamos a utilizar. Entre ellas se encuentran las funciones:

- Referidas a la compilación de los Shaders.
- Habilitar / deshabilitar funciones del pipeline (depth test, blending…).
- Asignar a los shaders variables de entrada como vértices, índices, coordenadas de textura…
- Asignar propiedades de textura y subirla a GPU.
- Pintar primitivas (triángulos, líneas…).

Estas funciones serán accesibles desde un módulo común que contenga la lógica para funciones de más alto nivel, como la carga del programa de Shading, mantenimiento de matrices de vista, carga de texturas en GPU…

A continuación se muestra como el módulo GL alcanza las funciones de la API OpenGL, en ambas plataformas.

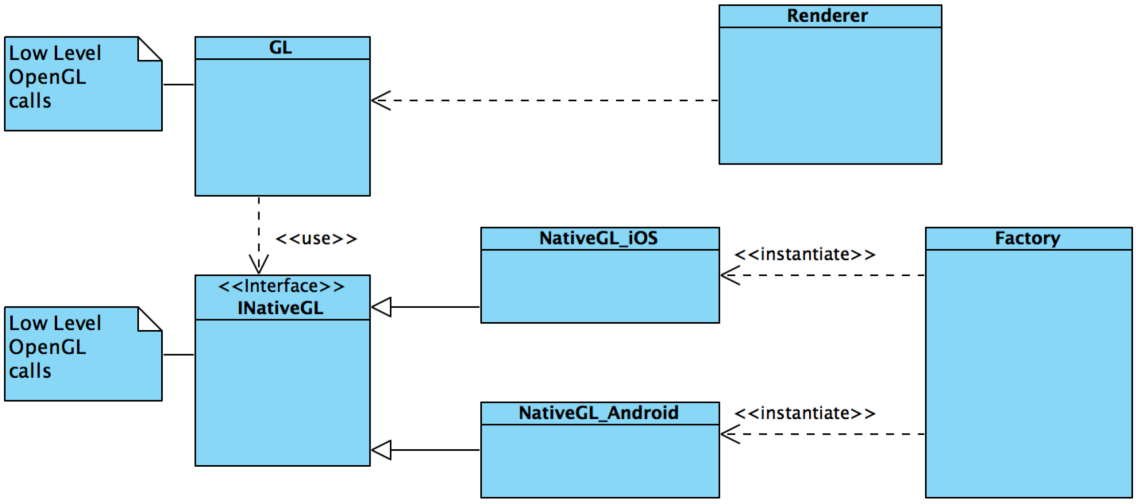

Figura 45 - Acceso a las API's gráficas

7. Estudio de usabilidad

Una vez llegado a este punto, en el que se ha definido tanto el diseño como los detalles de realización, es interesante presentar los resultados obtenidos a partir de la implementación final del proyecto.

Para ello, hemos decidido realizar pruebas generando una aplicación de demostración que utilice únicamente nuestro motor para mostrar el planeta Tierra, sin ninguna funcionalidad extra asociada. La configuración del sistema que mejores resultados ha ofrecido es utilizar una política de despliegue total del quad-tree y el test de LOD basado en área proyectada.

Esta aplicación de demostración constará de una única vista, la de la superficie de renderizado. En ella en el frame inicial se muestra el planeta Tierra sobre la que se mostrará una capa perteneciente a Open Street Map. La posición inicial de la cámara estará ubicada en el centro de coordenadas geográficas, con una altitud equivalente a 4 veces el radio de la Tierra.

Figura 46 - Primer frame producido por la demo

7.1. Definición de la prueba

En el estudio que estamos planteando pretendemos medir la eficiencia del Tile Renderer, puesto que, a parte de ser la pieza fundamental del sistema, es la que mayor carga computacional precisa y la que determinará en ultima instancia la velocidad a la que se producen los frames y la usabilidad general del sistema.

Debido a la arquitectura paralela que estamos utilizando, las tareas como procesado de imágenes o descargas de la red no afectan notablemente a la velocidad de generación de las imágenes. En el caso del Tile Renderer esta velocidad viene determinada mayoritariamente por el número de tiles visibles y por el tamaño del

Para testear la velocidad de renderizado que nos ofrece el motor, primero llevaremos la cámara hasta una distancia del globo en la que podamos observar el terreno con detalle. En este caso la llevaremos hasta mostrar tiles nivel de LOD 13. A continuación abatiremos la vista para producir una visión panorámica que muestre el mayor número de tiles posibles. A continuación la cámara volverá a alejarse del globo y después volverá a descender en otra área diferente llegando a mostrar LOD 16. De esta forma conseguiremos probar la eficiencia del motor de renderizado en diferentes configuraciones de jerarquía de tiles.

Para cada frame obtenido durante esta prueba se registrará el tiempo que se ha tardado en generarlo, el número de tiles que han sido dibujados y el número de tiles procesados dentro de los árboles de tiles.

La industria de los videojuegos y del cine han establecido como estándar de facto una tasa de 24 frames por segundo (FPS) para obtener una visualización sin saltos en el movimiento y una experiencia de usuario satisfactoria [28]. Una tasa de FPS menor provocaría incómodos saltos en la navegación para el usuario. Es por esto que consideraremos un tiempo adecuado para generación de un frame todo aquel que no rebase los 41.6 milisegundos.

En este punto es necesario decir que aunque en un determinado punto se puedan generar más cuadros por segundo de los necesarios, esto no se hará, debido a que un mayor uso del hardware gráfico implica un uso mayor de energía y a que la frecuencia de refresco de la pantalla provoca que esos frames se pierdan.

7.2. Plataformas para la prueba

Al ser este un proyecto destinado a la multiplaforma, es de rigor que las pruebas se realicen siempre sobre al menos un dispositivo con el sistema iOS y otro del mercado Android.

En este caso se cuenta con dos tablets de características técnicas similares y fecha de salida al mercado cercana:

- Tablet iPad de segunda generación.
- Tablet Samung Tab 10.1.

Las especificaciones técnicas (no siempre del todo transparentes al público) pueden ser consultadas en las referencias online que ofrecen Apple y Samsung respectivamente [29] [30].

7.3. Resultados

7.3.1. Resultados obtenidos con iPad

En las figuras 47 y 48 se muestran los datos obtenidos de la prueba realizada con el iPad. En ellos se observa una clara tendencia creciente de los frames a tardar más en completarse conforme se procesan más tiles. Sin embargo, la tendencia es aún más acusada cuando se aumenta el número de tiles a renderizar.

Esto nos indica que en el caso de este dispositivo en particular, es el hardware gráfico el que ofrece una mayor latencia ante crecimientos de la jerarquía de tiles.

Otro dato a mencionar es que en ningún momento se supera la barrera de un mínimo de 24 FPS, con lo que el control de la escena ha sido fluido y continuo en todo momento.

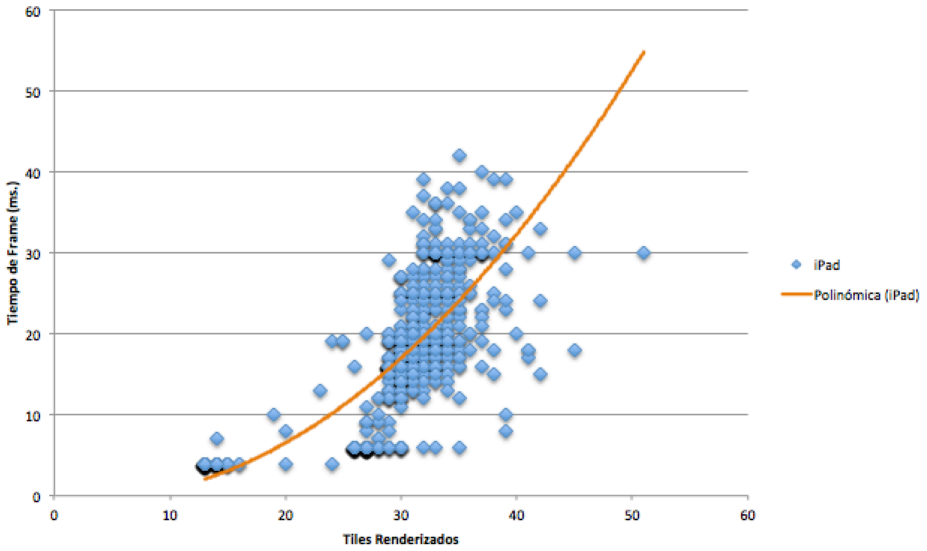

Figura 47 - iPad, Tiempo de Frame vs. Tiles Renderizados

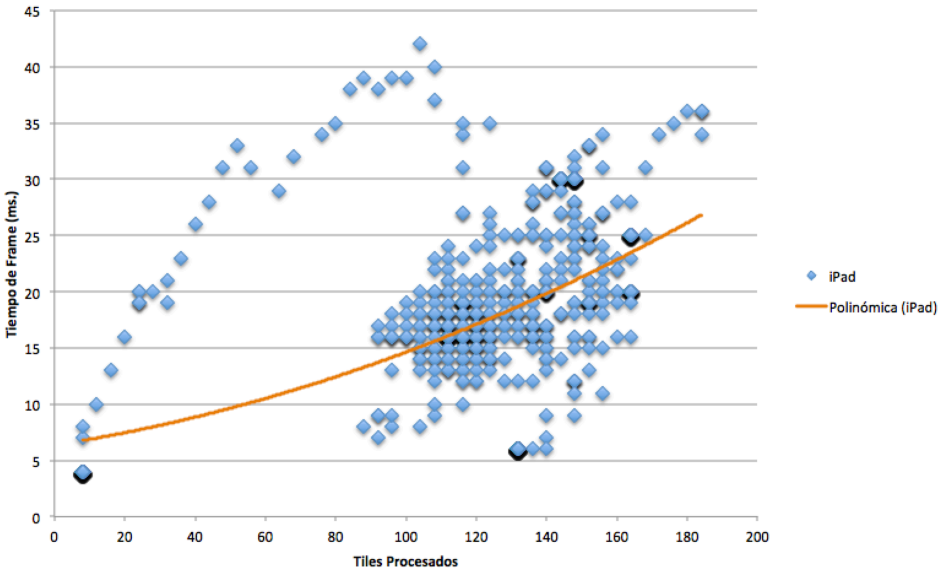

Figura 48 - iPad, Tiempo de Frame vs. Tiles Procesados

7.3.2. Resultados obtenidos con Samsung Tab 10.1

Las figuras 49 y 50, muestran los resultados del mismo test pero ejecutado sobre el tablet Samsung. Se observan las mismas tendencias crecientes que se observaban en el apartado anterior, aunque menos acusadas y con mayor dispersión de datos.

En este caso los resultados no son tan alentadores como en el caso anterior y si bien la inmensa mayoría de los tiles no superan los 60 milisegundos, existe un 20,96% de frames que superan la barrera de los 41.6 milisegundos que nos pusimos como límite superior para una navegación fluida.

Que los frames que tardan más se encuentren muy dispersos por el grafo, puede ser indicio de que los factores que estamos estudiando no son los que los producen, sino factores externos, que en el caso de Android pueden deberse a tareas en segundo plano del sistema operativo o de la máquina virtual, como puede ser el manejo de memoria.

Sin embargo, a nivel de experiencia personal, la navegación se realizó sin problemas durante todo el experimento. Y si bien, este dispositivo se notaba no tan desenvuelto como el de la prueba anterior, en ningún caso dio la impresión de malfuncionamiento. Sin embargo, es notoria la pérdida de rendimiento, probablemente debida a un hardware más genérico y el uso de máquinas virtuales de Java, como es la Dalvik [31], para la ejecución de la aplicación.

En resumen se puede decir que esta prueba, como la anterior, ha sido satisfactoria, demostrando el completo funcionamiento del motor y un alto nivel de usabilidad durante la demostración.

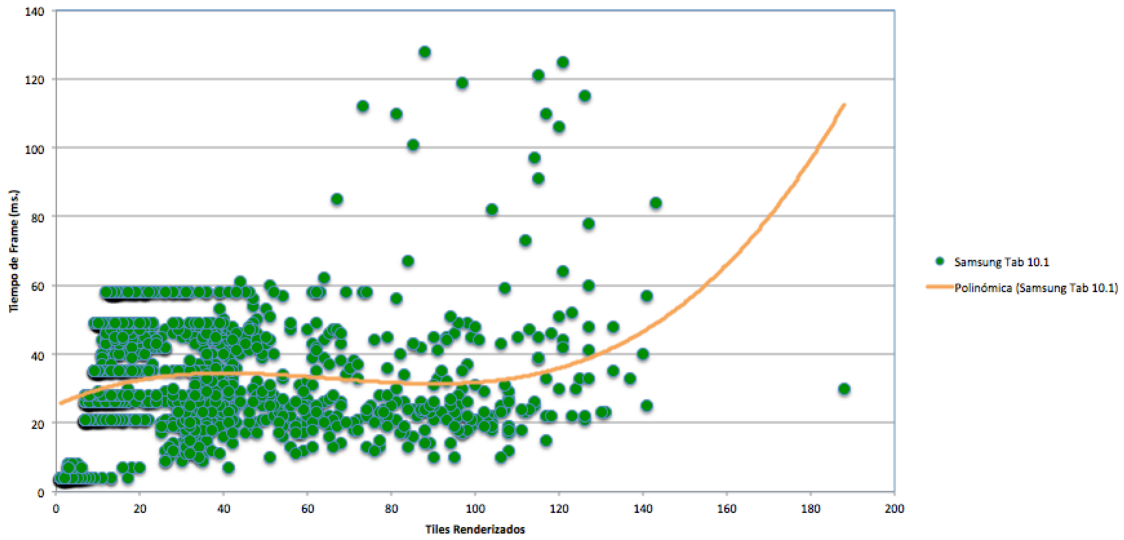

Figura 49 - Samsung Tab, Tiempo de Frame vs. Tiles Renderizados

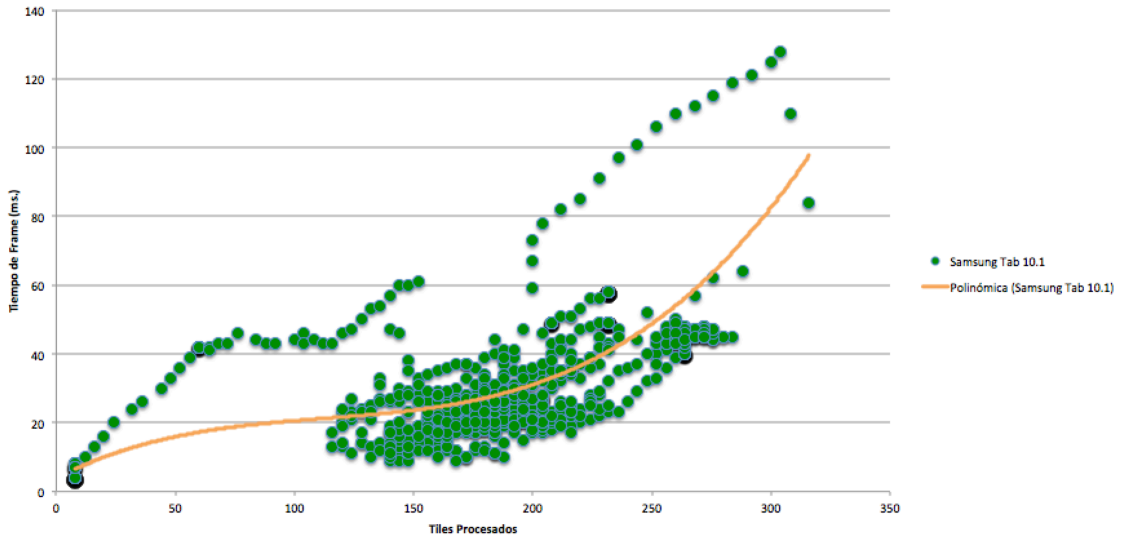

Figura 50 - Samsung Tab, Tiempo de Frame vs. Tiles Procesados

8. Conclusiones, mejoras y extensiones

Llegados a este punto se puede decir que se han completado las expectativas iniciales fijadas al comienzo de la vida del proyecto. Se ha conseguido desarrollar un motor gráfico con el que visualizar un globo terráqueo 3D, integrándolo dentro de un SDK con el que poder desarrollar aplicaciones.

A falta de añadir funcionalidades más avanzadas, nuestro motor puede compararse en términos de rendimiento y usabilidad a otros proyectos existentes en el mercado (muchos de ellos cerrados o de pago). Además, a partir de las pruebas que se han realizado durante el desarrollo, el producto final demuestra ser estable y eficiente durante la ejecución en ambas plataformas. Esto lo convierte en una alternativa a tener en cuenta a la hora de afrontar el desarrollo de una aplicación que utilice datos geolocalizados.

El proyecto está, por tanto, enfocado al desarrollo de extensiones que permitan el desarrollo de aplicaciones concretas que utilicen este motor gráfico. Las posibilidades en este sentido son ilimitadas. Se pueden desarrollar aplicaciones en campos como:

- Informática personal:
  - Localización personal o de ítems como vehículos.
  - Callejeros.
  - Directorios comerciales.
- Geomática y geología:
  - Visualización de movimientos sísmicos.
  - Catastro.
- Meteorología:
  - Visualización de campos de viento.
  - Mareas.
  - Simulación.
- Transporte:
  - Simulación y visualización de líneas de transporte.
  - Planificación.
- Aplicación militar:
  - Visualización de despliegue.
  - Visualización en balística o de vehículos.

El diseño del sistema, agrupando los objetos a visualizar en renderers ayuda a simplificar el desarrollo de estas extensiones. El modelo de implementación de las clases referidas a la representación geométrica del globo está pensado también para que, con pocos cambios, puedan representarse representaciones parciales del mismo o con diferentes proyecciones.

En este sentido, sería relativamente sencillo añadir al sistema otra vista en la que el planeta fuera representado como una superficie 2D. Esta proyección es bastante cómoda e intuitiva y por tanto utilizada frecuentemente en aplicaciones para mapas locales o callejeros.

Finalmente en el apartado de mejoras se ha de recalcar que gran parte del esfuerzo en este desarrollo está orientado a la eficiencia del renderizado, más teniendo en cuenta el hardware utilizado. Sin embargo, este puede ser continuamente mejora y actualizado ya sea bien por el uso de algoritmos y herramientas más optimizados, o por un uso más eficiente de las herramientas de cada sistema.

Una posible mejora a la eficiencia sería la carga de objetos fijos en memoria gráfica como la geometría de los tiles. De esta forma, no será necesario cargarlos en cada frame. Otro posible campo de estudio es la influencia de los distintos tipos de compresión de las texturas a la hora de la carga en GPU y rasterizado, así como su impacto visual.

También, y aunque se ha trabajado bastante en este sentido, se podría intentar mejorar los algoritmos de LOD y subdivisión de tiles, intentando conseguir representaciones más fieles de la superficie terrestre o que presenten un mejor rendimiento durante el renderizado.

Finalmente, es importante recalcar que existen otras plataformas a las que este motor y sistema de desarrollo podrían ser exportados. Sin ir más lejos, ya se está trabajando en su integración en entornos web utilizando estándares emergentes como es HTML5 y WebGL y técnologías de traducción de código como GWT.

Glosario

- OCG / Open Geospatial Consortium: Organización cuyo fin es la definición de estándares abiertos e interoperables dentro de los Sistemas de Información Geográfica y de la World Wide Web
- WMS / Web Map Services: Servicio que produce mapas de datos referenciados espacialmente, de forma dinámica a partir de información geográfica. Este estándar internacional define un "mapa" como una representación de la información geográfica en forma de un archivo de imagen digital conveniente para la exhibición en una pantalla de ordenador.
- Teléfono inteligente / smartphone: Teléfono móvil construido sobre unaplataforma informática móvil, con una mayor capacidad de computación y conectividad que unteléfono móvil convencional. El término «inteligente» hace referencia a la capacidad de usarse como un ordenador de bolsillo, llegando incluso a remplazar a un ordenador personal en algunos casos.
- Tableta / Tablet: Tipo de computadora portátil, de mayor tamaño que un teléfono inteligente o una PDA, integrado en una pantalla táctil (sencilla o multitáctil) con la que se interactúa primariamente con los dedos o una pluma stylus (pasiva o activa), sin necesidad de teclado físico ni ratón. Estos últimos se ven reemplazados por un teclado virtual y, en determinados modelos, por una mini-trackball integrada en uno de los bordes de la pantalla.
- Nivel de detalle / Level of detail / LOD: Característica que determina la precisión con la que un modelo representa el objeto al que hace referencia. En aplicaciones geográficas indica el grado de precesición en la representación del terreno.
- Misión topográfica de radar a bordo del transbordador / Shuttle Radar Topography Mission / SRTM: Misión para obtener un modelo digital de elevación de la zona del globo terráqueo entre 56 °S a 60 °N, de modo que genere una completa base de cartas topográficas digitales de alta resolución de la Tierra.
- Gráficos Vectoriales Redimensionables / Scalable Vector Graphics / SVG: Especificación para describir gráficos vectoriales bidimensionales, tanto estáticos como animados en formato XML.
- Computer Graphics Metafile / CGM: Formato abierto y estandar internacional para gráficos vectoriales 2D, imágenes ráster y texto, definido por ISO/IEC 8632.
- Interfaz de programación de aplicaciones / Application Programming Interface / API : Conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
- Baldosa / Tile: Superficie bidimensional de pequeño tamaño que forma un conjunto con otras similares para cubrir de forma completa una superficie mayor.
- FPS /Frames Per Second: Número de imágenes por segundo que produce un video o imagen en movimiento.

- GPU / Graphic Processing Unit: Coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y o aplicaciones 3D interactivas.
- Hilo de ejecución / Thread: Unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo. Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente. Un hilo es básicamente una tarea que puede ser ejecutada en paralelo con otra tarea.

Referencias

1. Ossim Planet - http://trac.osgeo.org/ossim/wiki/OssimPlanet
2. World Wind - http://worldwind.arc.nasa.gov/java/
3. Virtual Terrain Project - http://vterrain.org/
4. GLIDER - http://open.nasa.gov/plan/global-leveraged-integrated-data-explorer-for-research-glider/
5. GAEA+ - http://www.gaeaplus.eu/en/
6. JStaTrak - http://www.gano.name/shawn/JSatTrak/
7. Villamor C, Willis D, Wroblewski L (2010) Touch Gesture Reference Guide. - http://www.lukew.com/ff/entry.asp?1071
8. Ulrich T (2002) Rendering Massive Terrains using Chunked Level of Detail Control. Proceedings of SIGGRAPH2002. ACM Press
9. Web Map Services - http://www.opengeospatial.org/standards/wms/
10. Virtual Earth by Microsoft - http://www.microsoft.com/maps/
11. Open Street Map - http://www.openstreetmap.org/
12. Cozzi P., Ring K. (2011) 3D Engine Design for Virtual Globes. CRC Press.
13. Thread Pool Executor , Android API - http://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html
14. Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns Element of Reusable Object-Oriented Software
15. SQLite - http://www.sqlite.org/
16. Wright R., Lipchak B., Haemel N. (2007) OpenGL SuperBible 4th Edition, Addison-Wesley
17. Shreiner D. (2009) OpenGL Programming Guide 7th Edition, Addison-Wesley
18. Android Mobile Operating System (2012) - http://www.android.com
19. iOS Mobile Operating System (2012) - http://www.apple.com/iphone/ios
20. Proyección Mercator - http://es.wikipedia.org/wiki/Proyecci%C3%B3n_de_Mercator
21. Estudio de mercado de plataformas móviles - http://www.gartner.com/it/page.jsp?id=1622614
22. Khronos Group (2004) OpenGL ES: The Standard for Embedded Accelerated 3D Graphics. http://www.khronos.org/
23. OpenGL ES 2.0 - http://www.khronos.org/opengles/2_X/
24. Callejero Páginas Amarillas - http://callejero.paginasamarillas.es/home.asp
25. EndoMondo - http://www.endomondo.com/login
26. Geoscience Australia's World Wind Viewer - http://www.ga.gov.au/resources/multimedia/world-wind.jsp
27. Open Geospatial Consortium - http://www.opengeospatial.org/
28. FPS - http://en.wikipedia.org/wiki/Frame_rate
29. iPad 2 Specs - http://www.apple.com/ipad/ipad-2/specs.html
30. Samsung Tab 10.1 Features - http://www.samsung.com/global/microsite/galaxytab/10.1/feature.html
31. Dalvik VM - http://source.android.com/tech/dalvik/index.html