Glob3 Mobile: hacia un SIG 3D para entornos Apple-iOS, Android y WebGL

M. de la Calle[(1)], D. Gómez[(1)], A. Trujillo[(2)], J.M. Santana[(2)], K. Perdomo[(2)], J.P. Suárez[(3)], A. Pedriza[(4)]

[(1)] IGO SOFTWARE, Departamento de I+D - mdelacalle@igosoftware.es
[(2)] Universidad de Las Palmas de Gran Canaria. Departamento de Informática y Sistemas, atrujillo@dis.ulpgc.es
[(3)] Universidad de Las Palmas de Gran Canaria. Departamento de Cartografía y Expresión Gráfica en la Ingeniería, jsuarez@dcegi.ulpgc.es
[(4)] COTESA, Área de Sistemas de Información, alfonsopedriza@grupotecopy.es

RESUMEN

El trabajo describe el proyecto de desarrollo de un SIG 3D de código abierto para dispositivos móviles (Apple-iOS y Android) y para navegadores web con tecnología WebGL. En la fase actual, nos centraremos en el diseño e implementación del globo virtual, como elemento esencial que da soporte al SIG 3D y de una IDE que permite la programación de nuevas funcionalidades al globo. Dentro de los objetivos de diseño del globo virtual tenemos (i) simplicidad, con código estructurado que facilita la portabilidad y con una API de código abierto sencilla, (ii) eficiencia, tomando en cuenta los recursos hardware de los dispositivos móviles más extendidos en el mercado, (ii) usabilidad, implementando una navegación intuitiva mediante gestos para la interacción en pantalla y (iv) escalabilidad, gracias a una API desarrollada, se permite aumentar de las prestaciones mediante el desarrollo de scripts y podrán ser ejecutados tanto dentro del navegador web como de forma nativa en las plataformas móviles. Ante un panorama de clara proliferación de aplicaciones para móviles, Glob3 Mobile pretende ser una apuesta fuerte que llegue a convertirse en un SIG 3D de código abierto que abarque variadas aplicaciones sectoriales, algunas ya en marcha.

Palabras clave: *SIG 3D, dispositivo móvil, globo virtual*

ABSTRACT

The paper describes the project of developing an open source GIS 3D for mobile devices (Apple iOS and Android) and for web browsers with WebGL technology. At this stage, we focus on the design and implementation of the virtual globe as an essential element for the 3D GIS. An IDE is under development that allows the programming of new functionality to the globe. Within the design goals of the virtual globe we have (i) simplicity, with strcuture code enabling portability and with a simple open source API, (ii) efficiency, taking into account the hardware resources of mobile devices more widespread in the market, (ii) usability, implementing intuitive navigation through gestures for scree interaction and (iv) scalability, thanks to an API developed that permits new functionalities that enriches the globe
and implemented in both, within a web browser and as native on mobile devices. With a clear proliferation of mobile applications, Mobile Glob3 intends to be a truly open source 3D GIS covering various sectoral applications, some already underway.

INTRODUCCIÓN

El rápido avance en prestaciones y servicios de los dispositivos móviles ha dado un extraordinario impulso a las aplicaciones que sobre ellos se desarrollan. De esta manera es posible encontrar aplicaciones sectoriales muy diversas, en especial en el campo de los SIG's. Actualmente no existe ningún globo virtual 3D que funcione en las tres plataformas iOS, Android y Web. El único que existe es Google Earth, y no es de código abierto. Otros globos existentes actualmente que merece la pena destacar son:

- Nasa Worldwind[1]: escrito en Java, sólo funciona en ordenadores de sobremesa y portátiles:
- WebglEarth[2]: escrito en JavaScript y webgl. Funciona en cualquier navegador HTML5. Quizás funcione en móviles usando versiones recientes de Firefox mobile:
- ReadyMap[3]: escrito en JavaScript y webgl. Funciona en cualquier navegador HTML5. Quizás funcione en móviles usando versiones recientes de Firefox mobile:
- osgEarth[4]: escrito en C++. Funciona solo como aplicación de escritorio.

Para dispositivos móviles iOS y Android, nuestra propuesta está desarrollada directamente en el código nativo de cada dispositivo (Java en Android y C++ en iOS), aumentando así el rendimiento de toda la aplicación, y permitiendo crear aplicaciones completas utilizando el framework. De hecho, concretamente en la plataforma iOS, no existe (ni se le espera) un intérprete de Java, con lo cual globos como NASA lo tienen un poco crudo. Además, actualmente no hay soporte WebGL en el navegador Safari para iOS, y no existen versiones de Chrome ni Firefox por el momento. Los globos virtuales que están escritos con tecnología JavaScript y WebGL pueden funcionar en

[1] http://worldwind.arc.nasa.gov/java/
[2] http://www.webglearth.org/
[3] http://readymap.com/
[4] http://osgearth.org/

algunos dispositivos móviles, pero siempre embebidos dentro de un navegador, y su rendimiento no es del todo eficiente. WebGL está basado a su vez en la tecnología OpenGL ES 2.0, que es la librería gráfica usada en la mayoría de móviles y tablets actuales. Ambas librerías son muy similares, aunque sólo puede usarse desde JavaScript para poder funcionar dentro del navegador.

Existen actualmente numerosos motores basados en WebGL, que permiten programar aplicaciones gráficas a más alto nivel (WebGLU, GLGE, C3DL, Copperlicht). Sin embargo, en este proyecto hemos optado por crear nuestro propio motor usando directamente webgl, ya que los motores anteriores no podrían funcionar con OpenGL ES en los móviles.

El proyecto integra a dos empresas y dos departamentos universitarios. Se persigue la utilización de estándares OGC y formatos de representación de más aceptación en la industria (WMS, WFS, KML). Otro de los requisitos de su diseño es que a corto plazo pueda incluir el acceso a pasarelas a APIs abiertas de Internet (Clima, Geonames, Twitter, Facebook, …), así como la visualización de fotos, vídeos, modelos 3d, etc.

La Figura 1 muestra la aplicación de Globo Virtual Glob3 Mobile corriendo en iPhone de Apple y Samsug TAB bajo Android. Puede testearse libremente desde la web del proyecto (http://ami.dis.ulpgc.es/glob3m) y desde los *Markets* en iOS y Android.

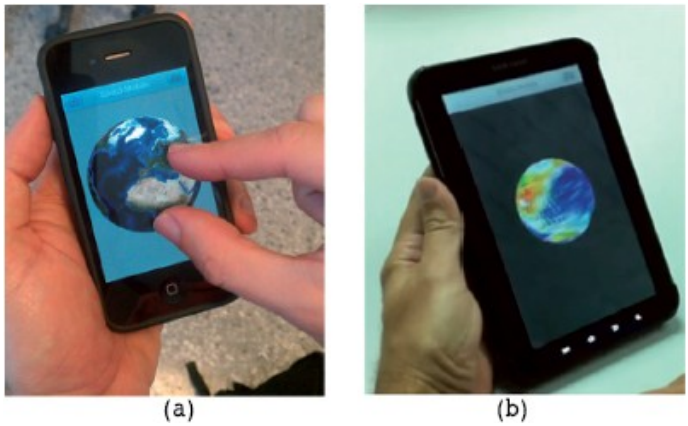

Figura 1: Glob3 Mobile en (a) iPhone de Apple y (b) Samsug TAB

EL FRAMEWORK DE GLOB3 MOBILE

Realmente, por la naturaleza de este proyecto, estamos obligados a tener 3 desarrollos en paralelo:

1. un proyecto en Objective C y C++ con OpenGL ES para funcionar en dispositivos iOS
2. un segundo proyecto en Java con OpenGL ES para funcionar en dispositivos Android
3. un tercer proyecto en JavaScript con Webgl para funcionar en navegadores HTML5

Existe una alternativa para Android que consiste en programar en C++ usando el NDK manteniendo las clases principales en java, pero la combinación entre ambos

fuentes nos dio bastantes problemas en las pruebas iniciales que hicimos. Esta alternativa podría parecer la opción más sencilla en un primer momento. Sin embargo, existen ciertas limitaciones que hacen más complicado el proceso de desarrollo.

En primer lugar nos encontramos con la necesidad de usar el compilador que nos ofrece NDK. Este, no soporta la totalidad de la STL, lo cual nos restringe de manera grave nuestra libertad a la hora de codificar. Por otro lado, el proceso de compilación y ejecución se hace algo más lento, pues implica la compilación del motor en una librería estática (que será cargada por la aplicación Android), la modificación del proyecto para forzar un resubido de la misma al dispositivo y por último la compilación y ejecución del proyecto Java.

Finalmente, y como desventaja fundamental y determinante del desarrollo con NDK es la imposibilidad de realizar un *debugging* interno del motor en la versión Android, que por experiencia sabemos que es en ocasiones necesario. Por lo tanto, se hace necesario desarrollar en los tres lenguajes de programación.

Sería una locura mantener tres proyectos de forma paralela, porque exigiría un coste de trabajo muy grande para mantener actualizadas las tres versiones en todo momento. Sin embargo, es imposible programar en un único lenguaje para las tres plataformas.

La solución elegida ha sido la siguiente: hemos desarrollado inicialmente todo el motor en C++ estándar. Este motor usará en todo momento clases virtuales para las funciones básicas (acceso al disco, acceso a la red, captura de eventos, trabajo con la librería gráfica), de tal forma que luego, para cada proyecto específico, se realizarán implementaciones de todas estas clases en el lenguaje correspondiente (Objective C para las clases específicas en iOS, Java para Android y JavaScript para web).

De esta forma, combinando el motor C++ con las clases específicas en Objective-C obtenemos la aplicación nativa para dispositivos iOS.

A continuación, con este motor escrito íntegramente en C++, utilizamos una herramienta software de conversión, que es capaz de convertir código escrito en C++ a código escrito en Java. Si bien hay que usar algunas recomendaciones en el código C++ para que el conversor funcione correctamente (ver anexo). Este motor convertido, junto con las clases específicas en Java, obtenemos la aplicación nativa para dispositivos Android.

Finalmente, con el motor escrito en Java, se usa la tecnología GWT de Google para convertir a código JavaScript. Este código, junto con las clases específicas en JavaScript permite obtener la aplicación para navegadores.

GWT es una herramienta de Google que permite generar código JavaScript a partir de una aplicación escrita en Java. Aunque incluya todos los elementos sintácticos habituales del lenguaje Java estándar, realmente no dispone de todas las características comunes con las que se cuenta a la hora de realizar una aplicación de escritorio, debido a las limitaciones inherentes a JavaScript, que es el lenguaje que en última instancia se utiliza en tiempo de ejecución. Por ejemplo, desde GWT no se podría hacer uso del sistema de archivos, ya que JavaScript no lo permite, ni se podrían hacer peticiones remotas a dominios externos debido a las Same Origin Policies.

Sin embargo, teniendo en cuenta estas limitaciones, se puede conseguir una reutilización del código prácticamente del 100%, utilizando un subconjunto de clases Java que se conozca tengan su equivalente en GWT, y minimizando o eliminando el

uso de librerías o clases de terceros. La Figura 2 muestra el ciclo de vida del desarrollo del proyecto, donde se indican mediante flechas la migración de código, y los lenguajes de implementación del motor de Glob3 Mobile y de la clases dependiente de cada plataforma.

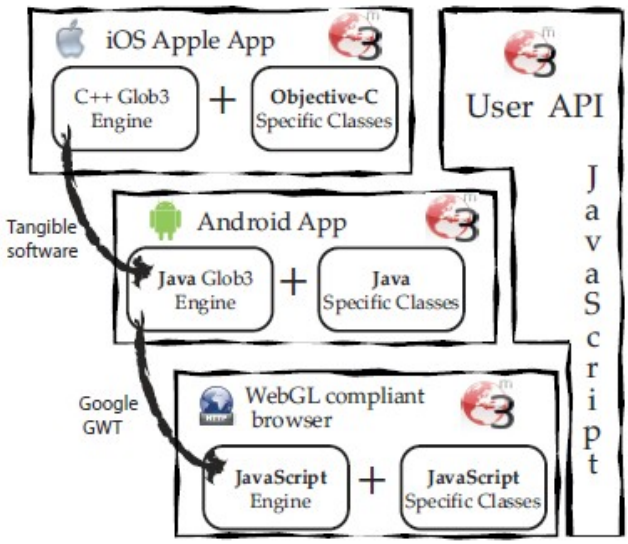

Figura 2: *Ciclo de vida de desarrollo de Glob3 Mobile.*

Arquitectura del Motor de Glob3 Mobile

El motor es el núcleo del Framework. A continuación se describen los distintos módulos del motor así cómo los aspectos más interesantes de su funcionalidad:

Ellipsoidal Terrain Engine: se crean una malla regular de triángulos sobre la superficie del ellipsoide, usando el sistema de proyección WGS84. El sistema debe permitir volar en 3D, hacer *drag&drop* del globo, hacer zoom, rotar la vista, etc., de forma rápida en cualquier plataforma.

LOD (Level Of Detail): Necesitamos ejecutar una estrategia LOD para representar la superficie con distintos niveles de detalle, atendiendo al criterio de distancia a la cámara, ángulo de inclinación del observador, área en la pantalla, etc). Utilizamos un algoritmo de niveles discretos llamado Chunk LOD. Para evitar los saltos entre tiles vecinos usamos faldas (skirts), ver Figura 2 de un ejemplo de terreno sin aplicar faldas y a la derecha con su aplicación.

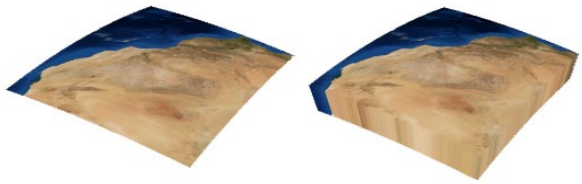

Figura 3: A la izquierda una imagen de textura del globo. A la derecha la misma textura con aplicación de faldas en los bordes.

En concreto, la condición del LOD que se chequea en cada *frame* de imagen es la siguiente:

$$\varepsilon(T) = \varepsilon_{\max} + (\varepsilon_{\min} - \varepsilon_{\max}) \frac{w_T - D/4}{\pi R - D/4} \qquad (1)$$

Donde w_T es el tamaño actual del tile de imagen, *D* es la distancia desde el ojo del observador a *CPV*, y *R* es el radio del globo. Así pues, se procede con una subdivisión en cuatro nuevos hijos de imagen, si se cumple las siguientes condiciones:

$$\frac{d_T}{w_T} < \varepsilon \qquad (2)$$

$$w_T > D/4 \qquad (3)$$

En la Figura 4 se ilustra la Ilustración de la técnica LOD aplicada en Glob3 Mobile.

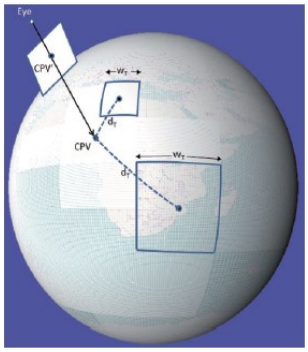

Figura 4: Ilustración de la técnica LOD aplicada en Glob3 Mobile.

Out-of-core rendering: Se llama así cuando toda la escena de trabajo no cabe en memoria, con lo cual mantenemos en memoria solo un subconjunto del terreno (lo necesario para visualizar en cada frame) y hay que traerse desde el disco el resto de la escena a medida que se navega por el globo. Para eso se utilizan políticas de cacheado en disco.

Conexión a servidores WMS: para pegar la imagen aérea de cada tile conectamos a servidores WMS (protocolo estandar del OGC) para descargarnos las texturas de cada tile que se va pintando. El sistema WMS sirve una imagen a partir de las coordenadas de las esquinas del rectángulo (bounding box que se quiere obtener), y por lo tanto las vamos pidiendo a medida que volamos. Hay un montón de servidores públicos, lo que ocurre es que no todos tienen imágenes de todo el mundo. Algunos solo tienen de un país (como el Instituto Geográfico Nacional (IGN)), o de una comunidad (como Grafcan). Así que el sistema permite al usuario crear una lista de servidores, y en función de la localización de cada tile se elige automáticamente a qué servidor pedir en cada caso. A continuación se muestra un ejemplo de petición mediante URL para acceder vía WMS a una textura de la isla de Gran Canaria, España:

```
http://idecan.grafcan.com/ServicioWMS/OrtoExpress?
SERVICE=WMS&LAYERS=ortoexpress&
REQUEST=GetMap&VERSION=1.1.0&
FORMAT=image/jpeg&SRS=EPSG 32628&WIDTH=512&
HEIGHT=512&BBOX=416000,3067000,466000,3118000&
REFERER=GLOB3MOBILE
```

Conexión a los servidores de la NASA: el único servidor que conocemos que cubre todo el mundo con buena resolución es de la NASA (el de Google es mejor pero no es público, sólo a través de google maps o earth). Aunque usa el protocolo WMS para las imágenes más lejanas, usa un sistema de tiles prefijado para las imágenes de más resolución (Virtual Earth), con lo cual se ha desarrollado un módulo para poder realizar las peticiones.

Petición de texturas y memoria. Cuando se ha pedido una textura a la red para un tile, se guarda en una cache de disco, para que cuando se necesite otra vez se vaya primero al disco. Tampoco se puede colapsar un servidor pidiendo demasiadas texturas a la vez. Y tampoco puedo mostrar demasiadas texturas simultáneamente, porque no caben en la memoria de la GPU de un móvil. Se ha creado una política de peticiones para poder cumplir todas estas condiciones.

Ejecución multihilos (multithreading): cuando se piden texturas a la red o a la cache se hace en un hilo (thread) aparte, para no interrumpir la tasa de frames por segundo (FPS) del renderizado, con lo cual no se pintan nuevos tiles hasta que no lleguen las texturas.

Compresión de imágenes: para que las texturas ocupen menos espacio se utilizan texturas de 16 bits por cada tile, sin canal alpha. No pueden usarse formatos de compresión ya que el formato manejado por el hardware de los móviles no es el mismo que exporta un servidor WMS, y además el hardware del móvil no puede comprimir y descomprimir en tiempo real a ese formato. Por lo tanto usamos JPG.

Manejo de las alturas del terreno: Para incorporar elevaciones al terreno, usamos el formato BIL que exportan los servidores de la NASA, que nos devuelve una matriz de valores dado un contorno o *bounding box,* que son usados para levantar los vértices de la malla dicha altura sobre la altura del eliposide y dar la ilusión de elevación del terreno. A continuación se muestra un ejemplo de petición *WMS* de una imagen para recuperar las alturas de la isla de Gran Canaria, España.

```
http://www.idee.es/wcs/IDEE-WCS-UTM28N/wcsServlet?
REQUEST=GetCoverage&SERVICE=WCS&
VERSION=1.0.0&FORMAT=GeoTiff&
COVERAGE=MDT_canarias&
BBOX=416000,3067000,466000,3118000&
CRS=EPSG:25828&RESX=100&RESY=100&
REFERER=GLOB3MOBILE
```

Capas transparentes: a veces el usuario quiere mostrar capas WMS con transparencia (como un callejero) sobre la ortofoto, con lo cual deben mostrarse texturas. Normalmente esto se realiza en un equipo sobremesa usando multitextura (dos texturas por tile se le mandan a la GPU y ésta las mezcla). Pero en el caso de un móvil sería demasiada carga de memoria, así que la mezcla se hace en CPU. La Figura 5 muestra un ejemplo de combinación de texturas con transparencias en Glob3 Mobile.

Figura 5: *Ejemplo de combinación de texturas con transparencias.*

Marcadores sobre el terreno: para poder mostrar una lista de marcadores, que representen posiciones sobre el terreno, se ha implementado usando una técnica de billboards para poder mostrar texturitas que tengan tamaño contante en pantalla, y que siempre den la cara al usuario.

Clases virtuales del motor de Glob3 Mobile

Las clases virtuales usadas en el proyecto, y común a la implementación en las tres plataformas son:

IFileManager: permite leer y escribir en disco (sólo usada en las versiones iOS y Android, pero no en la de web).

INetwork: permite recuperar información de internet a partir de una url (imágenes, binarios, xml files).

IEvent: captura de eventos específica en cada plataforma (incluso en la versión webgl, aunque al ejecutarse en un equipo normal no existe tecnología *touch* ni *multitouch*, sí que existe en las versiones móviles de los navegadores.

ILocation: permite acceso a la brújula (si existe) y al GPS (si existe) del dispositivo. Incluso para la versión web, existe una tecnología propia de HTML5 que permite obtener la ubicación de un equipo de escritorio mediante una API estándar que se encarga de definir la organización W3C. Esta geolocalización web se realiza en base a la posición de la centralita telefónica a la que se está conectando el equipo del usuario, por lo que la estimación no es tan precisa como la que se puede obtener mediante un dispositivo GPS.

IImage: acceso a los datos de una imagen, en diferentes formatos (JPG, TIFF, PNG).

IGL: encapsula toda la funcionalidad de la librería gráfica, bien sea OpenGL-ES en los móviles, o WebGL en la versión web.

ITimer: mediciones de tiempo (cálculo del tiempo de un *frame*, animaciones, time out en las peticiones a red).

IUtils: utilidades varias (imprimir por pantalla, *debug* mensajes por consola.

En la Figura 6 se muestra la arquitectura en capas del proyecto GLob3 Mobile, donde se observan la interconexión de módulos. Se percibe las capas dependientes de las plataformas en la parte inferior del diagrama, mientras que la vista de usuario se encuentra en la parte superior, reflejando aquí la propia aplicación del globo virtual así como la API de programación que se facilita mediante una IDE sencilla, aún en construcción, y que se describirá más adelante.

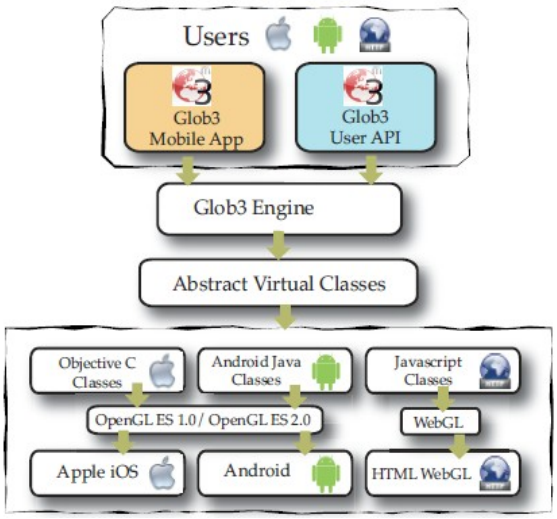

Figura 6: *Arquitectura en capas del proyecto GLob3 Mobile*

ALGUNOS ASPECTOS DE LA IMPLEMENTACIÓN DE GLOB3 MOBILE

Peticiones de red en la aplicación Glob3 Mobile

La forma de hacer una petición *url* para descargar una imagen, o un fichero binario (como el formato *BIL*) es diferente en cada plataforma, y por ello tenemos una clase virtual *INetwork*. En *iOS* y en *Android* se dispone de una clase con métodos que se le pasa la *url* y de forma asíncrona, se lanza un hilo que hace realmente la petición. Luego existen otros métodos activados con los eventos cuando la petición ha llegado, o bien si ha dado error, o bien si se ha producido un *timeout*.

Cada plataforma permite ejecutar peticiones *HTTP* de una forma sencilla. En iOS contamos con la clase *NSURLConnection* que realiza la descarga por nosotros avisándonos de las diversas contingencias que puedan darse como son un *TimeOut*, un servidor no encontrado o la llegada de datos corruptos. El buffer de salida de datos por su parte será un *NSMutableData*.

En *Android* la clase *URLConnection* realiza exactamente el mismo papel, abriéndonos un *stream* de datos con el servidor del que podemos leer los datos que necesitemos. Posteriormente cerraremos esta conexión cerrando el *stream* de datos (clase *InputStream*). Cualquier fallo en la comunicación conduciría a producir una *IOException*.

En la plataforma *Android* contamos con una clase del sistema llamada *AsyncTask* que puede ser instanciada e invocada. Esta crea un hilo donde ejecutar la tarea principal de forma asíncrona y permite a su vez ejecutar funciones específicas antes y después de la ejecución de la misma. Una de las ventajas que presenta esta clase, a parte de su simplicidad, es que aprovecha un pool de hilos del sistema para ejecutar las tareas, lo que nos ahorra una parte importante del tiempo de lanzamiento y destrucción de hilos.

Sin embargo, en *JavaScript* la cosa es diferente. *JavaScript* no es un lenguaje que esté pensado para tratar directamente con datos binarios como pueden ser los de una imagen, sino que fue ideado para la interacción con los elementos de una página web, los cuales son en su mayor parte objetos textuales. Los elementos más

complejos como las propias imágenes se suelen manejar a través de abstracciones que ocultan los detalles de bajo nivel.

Es por esto que cuando se pide un recurso remoto en *JavaScript*, el navegador presupone que la respuesta contiene texto, y la traduce a la codificación de caracteres que esté utilizando en ese momento. Por lo tanto, no se puede asegurar que los valores de los bytes que se reciben sean los mismos que los que se enviaron originalmente, al depender del sistema de codificación que esté utilizando el cliente. Ni siquiera se puede asegurar que se encuentren en un rango entre 0 y 255, ya que muchos de los juegos de caracteres que se utilizan en los navegadores modernos utilizan más de un byte para representar la información.

Un método para intentar solventar estas dificultades es indicarle a JavaScript que ignore la propia configuración del navegador y utilice un mapa de caracteres conocido para realizar la conversión Esta estrategia haría posible recibir información binaria y solventaría casos como los ficheros *BIL*. Sin embargo, a la hora de manejar imágenes se añade otra dificultad, y es que *JavaScript* no permite acceder a la matriz de píxeles de una imagen ni para escribir ni para leer la información. Luego, aún disponiendo de la cadena binaria que representa el contenido de una imagen, no sería posible generar la imagen en sí a partir de ella.

Afortunadamente, *JavaScript* cuenta con un objeto específico para el manejo de imágenes, que dispone de mecanismos para obtenerlas de la red simplemente indicando su *URL*, además de eventos que indican su estado en cada momento: si se ha producido algún error, si se ha abortado la transferencia, etc. Como se ha comentado, existe la limitación de no poder acceder directamente a su contenido ni modificarlo. En la práctica, esto no es necesario en la mayoría de los casos, por lo que no supone un gran inconveniente.

También hay que mencionar el problema de seguridad *cross-domain* que recientemente ha sido extendido a *WebGL*, habiéndolo integrado la mayoría de los navegadores en sus versiones más recientes, y que afecta a la forma de solicitar las texturas a los servidores *WMS* desde nuestro motor. Realmente el problema es general a la obtención de cualquier tipo de recurso que se encuentre en un sitio remoto, aunque hasta hace poco *WebGL* era la excepción a la regla.

El problema se debe a que se considera inseguro que un cliente dado pueda solicitar un dato a un servicio que no se encuentre en el mismo dominio en el que se encuentra él mismo. Como se comentó antes, estas políticas se denominan *Same Origin Policies* y son probablemente la restricción más problemática que se puede encontrar a la hora de desarrollar un proyecto de estas características en el que todos los mapas se van a descargar siempre de servidores externos.

Existen mecanismos (las cabeceras *CORS* del protocolo *HTTP*) pensados para que los que proporcionan servicios online indiquen explícitamente que los clientes pueden saltarse las restricciones *SOP* y solicitar datos desde dominios externos, aunque desgraciadamente su implantación es mínima en la actualidad. Se espera que en el futuro su uso se generalice y se elimine esta barrera.

Mientras tanto, la solución típica para resolver este problema es colocar un proxy en nuestro servidor que se encargue de retransmitir las peticiones entre las dos partes (el servidor remoto y el cliente local). Este proxy puede implementarse de muchas formas: desde un simple script en *PHP*, hasta sistemas específicamente desarrollados para la solventar este tipo de situaciones, como la herramienta *MapProxy*. Para cada solución hay que valorar ventajas e inconvenientes y decidir cual conviene más para el proyecto en función de los requisitos. Por ejemplo, un script en *PHP* es una solución ligera y sencilla de mantener. Por otro lado, una

herramienta como *MapProxy* es muy potente al disponer de muchas opciones de configuración, aunque en caso de necesitar adaptarla a casos particulares puede ser complicada de modificar.

En el caso concreto de este proyecto, se decidió optar por la implementación de un script *PHP* ya que tras diferentes pruebas con *MapProxy*, se observó que era demasiado pesada para la infraestructura hardware con la que se contaba. Además, *MapProxy* necesita que se defina explícitamente en su fichero de configuración la lista de servidores *WMS* con los que se va a trabajar, cosa que no interesaba, ya que la idea era que los usuarios pudieran añadir y eliminar capas *WMS* en tiempo de ejecución.

También se observó que ciertos servidores *WMS* no parecían aceptar peticiones de recursos por parte de navegadores web. Se aprovechó de que se disponía de un script intermedio para eliminar de las cabeceras *HTTP* cualquier información que indicara que el origen de las peticiones era un navegador, como por ejemplo el campo "*User-Agent".* Esto es un ejemplo de la versatilidad que permite el uso de un script *PHP*. Implementar esta funcionalidad en *MapProxy* habría sido muy complicado.

Manejo de la cache

Glob3 mobile para móviles va creando una cache donde va almacenando todas las imágenes que se van descargando, para que cuando haya que volver a usarlas, acuda aquí primero antes de ir a la red. Sin embargo, cuando trabajamos con *JavaScript*, no tenemos acceso al disco del cliente, y el navegador maneja su propia cache. En principio esto es una ventaja, porque permite obviar la lógica de preguntar primero antes si está en cache. Sin embargo, hubo que hacer algún arreglo para garantizar que las imágenes que se pide por el protocolo *WMS* queden almacenadas en la cache del navegador, porque el comportamiento del sistema de caché es muy variable de un navegador a otro. Para asegurar que todos los navegadores se comportasen de la misma forma, se forzó a incluir en las cabeceras *HTTP* el campo "*Max-Age*", que indica al navegador que tiene que cachear un recurso y la cantidad de tiempo en segundos de vigencia. Nuevamente, esto ha sido posible gracias al script PHP intermedio, que permite modificar las cabeceras *HTTP* antes de que lleguen al navegador.

UNA API PARA INCREMENTAR LAS POSIBILIDADES DEL GLOBO VIRTUAL

Una vez tenemos el *framework* terminado, la idea es que cualquier usuario desarrollador pueda escribir su propio script o *plugin* con nuestra API. Además, se ha puesto énfasis en que el usuario pueda acceder a prácticamente todas las funcionalidades del motor desde el *plugin*. De esta forma, el usuario puede añadir capas *WMS* al globo, mover la cámara, e incluso dibujar su propia geometría sobre el globo. Por ahora es una *API* muy básica pero que irá creciendo en futuras versiones.

La idea que tenemos es que el *plugin* se escriba una única vez y se pueda ejecutar en todas las plataformas. Por lo tanto, hemos elegido como lenguaje el *JavaScript*. De esta forma, para la versión del motor en *WebGL*, el *plugin* se puede ejecutar directamente. Para las versiones de *Android* e *iPad* hubo que integrar dentro del proyecto un motor de JavaScript que permite interpretar código *JavaScript* dentro de la plataforma. Al final se consiguió usando los motores libres *Rhino* para *Android* y *JavaScriptCore* de *Apple* para *iPad*.

En el caso de la plataforma *iOS* las políticas de Apple sobre lo que es "legal" ejecutar en sus dispositivos ha desanimado a la mayoría de desarrollares a intentar utilizar scripting en sus aplicaciones. Es por esto que no existe ningún motor oficial de *JavaScript* para *iOS*. Estas políticas permiten (al menos de momento) la ejecución de scripts que no sean descargados de la red y bajo estas premisas trabajamos.

JavaScriptCore es un módulo fundamental de *WebKit*, que a su vez es el motor de renderizado web utilizado por la plataforma Mac OS X e iOS. A pesar de ser código abierto, su implementación es muy extensa y dependiente de la estructura hardware de la máquina.

Sin embargo, y tras una búsqueda exhaustiva de otros proyectos que tuvieran el mismo problema encontramos referencias de motores de juegos que a finales del año pasado habían portado el motor *JavaScriptCore* a iOS. La integración en el proyecto a partir de ahí fue bastante más sencilla[5].

Una vez llegado a este punto se necesita que las clases C++ sean accesibles desde el motor de JavaScript. Esta tarea, siendo sencilla, también reporta un trabajo considerable en mantenimiento debido a que *JavaScriptCore* esta pensado para interactuar con funciones C. De esta forma hay que crear una interfaz específica para cada función de cada clase que queramos sea accesible y añadirla en el motor.

En el caso de Android no teníamos ninguna política que restringiese el uso de engines de scripting. En este sentido la organización Mozilla, ha liberado su motor de JavaScript sobre Java llamado Rhino. A partir de ahí la integración Java-Javascript es bastante directa pues basta con exponer una clase o paquete de Java al motor JavaScript mediante las llamadas correspondientes y estos pasan a ser accesibles desde el motor de forma directa sin necesidad de *wrapping*.

Por ahora sólo hay unos cuantos ejemplos muy básicos disponibles desarrollados con nuestra API, pero que iremos ampliando. Además, hemos creado una página web con un entorno de desarrollo integrado (IDE) muy básico, que permite al usuario partir de un código ejemplo de un *plugin*, modificarlo y ejecutarlo dentro de la misma página. De esta forma, una vez probado, puede descargarlo directamente a su propia web para que se pueda ejecutar desde allí, o bien descargar una aplicación específica para una de las plataformas, ya con su plugin integrado en ella.

A continuación se muestra un ejemplo sencillo de *script*:

```
function glob3m_main() {
//Globe instance
var globe=Globe.createDefault();
//Latitude, longitude and height
var latitude=27.9;
var longitude=-15.86;
var height=100;
//Set the camera position
globe.camera.position=Position.create(..
latitude,longitude,height);
```

En la Figura 7 se presenta la interfaz web para Glob3 Mobile donde se aprecia la IDE en desarrollo para desarrollar nuevos scripts

[5] http://www.phoboslab.org/log/2011/06/javascriptcore-project-files-for-ios

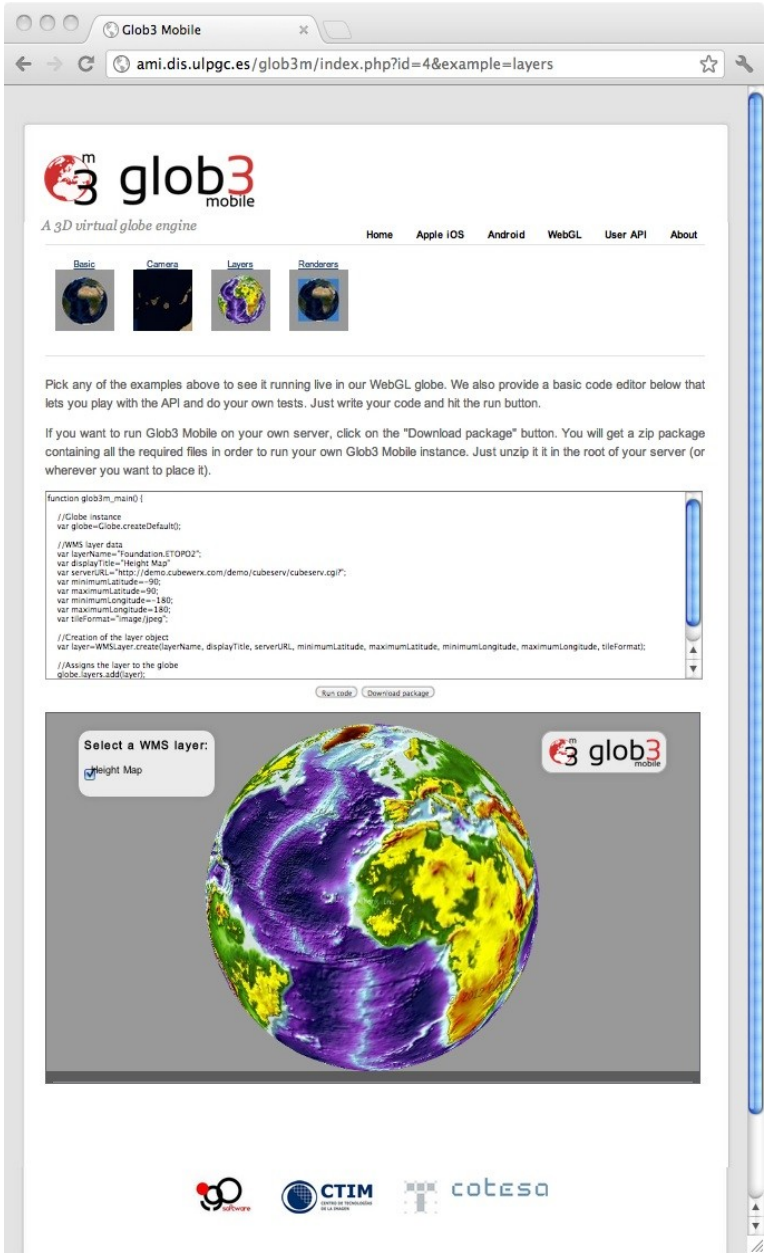

Figura 7: *Interfaz web para Glob3 Mobile donde se aprecia la IDE en desarrollo para desarrollar nuevos scripts. Acceso a Glob3 Mobile desde la web del proyecto[6] y desde los Markets en iOS y Android.*

CONCLUSIONES Y TRABAJO FUTURO

Glob3 Mobile es un proyecto vivo que crece día a día. En la Tabla 1 se ofrece una comparativa de características entre diversos globos virtuales actuales. Dichas características son: si es realmente de código abierto (open source) o no, si trabaja en 3D, si está implementado para dispositivos móviles o tabletas, y si se ha desarrollado en código nativo independiente siendo más eficaz. Se observa que GLob3 Mobile posee de forma satisfactoria las características estudiadas.

[6] http://ami.dis.ulpgc.es/glob3m/ Glob 3 Mobile.

Tabla 1: Comparativa actual de Glob3 Mobile y otros globos virtuales existentes.

	Op. Source	3D	Mob. Dev.	Native C.
Google Maps	×	×	V	×
Google Earth	×	V	V	V
Open Layers	V	×	V	×
WebGLEarth	V	V	×	×
Glob3 Mobile	V	V	V	V

Algunos retos recientes que se plantean en el proyecto son, por ejemplo la integración del globo con Glob3 Desktop & Euclid: realmente Glob3 Mobile pretende ser la versión móvil del framework glob3, desarrollado por IGO software, que funciona sobre el motor EorldWind de la NASA en sobremesa. El objetivo es poder utilizar toda la funcionalidad que ya existe en un dispositivo móvil, entre ellas: streaming de vídeo, visualización de grandes nubes de puntos por de ejemplo de datos atmosféricos, y de geometría diversa (vectors, polilíneas) o uso de streaming a un servidor de escenas Mostrar modelos 3D, edificios, etc. Por otro lado, las siguientes funcionalidades son deseables: Mostrar fotos y vídeos sobre el terreno, conexión a API's de internet (DropBox, Facebook, Twitter, Geonames) y acceso a más formatos geográficos (finalizar KML, WFS, OSM) etc.

AGRADECIMIENTOS

Este proyecto ha sido financiado por la empresa IGO software y parcialmente por: (i) COTESA, Área de Sistemas de Información, (ii) proyecto de la AECID, REF A/030194/10, del programa de cooperación interuniversitaria e investigación científica 2009-2011, y (ii) Proyecto CYCIT Project MTM2008-05866-C03-02 del Ministerio de Educación y Ciencia de España.

REFERENCIAS

[1] Glob3 mobile page project [online]. http://ami.dis.ulpgc.es/glob3m/, 2012 [cit. 2012-03-11].

[2] D.G. Bell, F. Kuehnel, C. Maxwell, R. Kim, K. Kasraie, T. Gaskins, P. Hogan, and J. Coughlan. Nasa world wind: Opensource gis for mission operations. In *Aerospace Conference, 2007 IEEE*, pages 1 –9, march 2007.

[3] V. Chandola, R.R. Vatsavai, and B. Bhaduri. iglobe: an interactive visualization and analysis framework for geospatial data. In *Proceedings of*

the 2nd International Conference on Computing for Geospatial Research & Applications, COM.Geo '11, pages 21:1–21:6, New York, NY, USA, 2011. ACM.

[4] P. Cozzi and K. Ring. *3D Engine Design for Virtual Globes*. A. K. Peters, Ltd., Natick, MA, USA, 1st edition, 2011.

[5] Khronos Group. Opengl es: The standard for embedded accelerated 3d graphics [online]. http://www.khronos.org, 2004 [cit. 2012-03-11].

[6] J.M. Noguera, R.J. Segura, C.J. Ogáyar, and R. Joan-Arinyo. Navigating large terrains using commodity mobile devices. *Computers & Geosciences*, 37(9):1218–1233, 2010.

[7] K. Pulli, T. Aarnio, K. Roimela, and J. Vaarala. Designing graphics programming interfaces for mobile devices. *IEEE Comput. Graph. Appl.*, 25(6):66–75, November 2005.

[8] P. Sloup. Webgl earth [online], 2011 [cit. 2012-03-11].

[9] U. Thatcher. Rendering massive terrains using chunked level of detail control. In *Proceedings of the SIGGRAPH-2002*. ACM Press, 2002.

[10] A. Trujillo, D. Gómez, M. de la Calle, J.P. Suárez, A. Pedriza, and J.M. Santana. Glob3 mobile: An open source framework for designing virtual globes on ios and android mobile devices. In *7th International 3DGeoInfo Conference*, Quebec, Canada, 2012. To appear.