An Open Source Virtual Globe Framework for iOS, Android and WebGL Compliant Browser

José P. Suárez
IUMA. Inf. and Comm. Sys.
Univ. of Las Palmas de G. C.
Las Palmas de G. C., Spain
jsuarez@dcegi.ulpgc.es

Agustín Trujillo
Imaging Technology Center
Univ. of Las Palmas de G. C.
Las Palmas de G. C., Spain
atrujillo@dis.ulpgc.es

Manuel de la Calle
IGO SOFTWARE
Departamento de I+D
Cáceres, Spain.
delacalle@igosoftware.es

Diego D. Gómez
IGO SOFTWARE
Departamento de I+D
Cáceres, Spain.
diego@consultar.com

Jose M. Santana
Imaging Technology Center
Univ. of Las Palmas de G. C.
Las Palmas de G. C., Spain
amazingsmash@gmail.com

ABSTRACT
The present article introduces the Glob3 Mobile, an efficient virtual globe framework, testable and freely accessible from the web and providing a real 3D navigation experience with smooth flying. We discuss the software design and the native code implementation for the family of iOS and Android mobile devices. In addition a novel web experience is facilitated by supporting the globe through a WebGL compliant browser. Glob3 mobile enjoys a user friendly API that allows for testing and scripting new features on the globe. The paper discusses the software design, the technical roll-out options and provides a comparison with other similar existing solutions.

Categories and Subject Descriptors
H.4 [Information Systems Applications]: Miscellaneous; D.2.8 [Software Engineering]: Programming Environments — Interactive environments, Graphical environments.

Keywords
Virtual globe, mobile device, geospatial application.

1. INTRODUCTION
Cross-platform 3D virtual globes integrating environmental data at any time or place, still represent a challenge in the geospatial community. They play a central role in research and applications, such as map browsers among others. Some recent applications are emerging worldwide, [3, 5] such as is demonstrated in the specialized literature in the work of Cozzi and Ring, [4] for example, a basic reference guide on the subject.

The development of virtual globes began in 2001 with companies pursuing effective communication of their research and results to a broader audience worldwide. It was Google Earth, created by Keyhole, Inc that brought virtual globes to the world’s attention when Google took over the latter company in 2004. At the same time, NASA World Wind, [2] also brought out a powerful open source implementation with highly versatile possibilities. It is written in Java, so it can be run as an applet, or embedded in a web page. World Wind, however, does not work on mobile Apple devices and is not expected to, since there is no Java interpreter on these devices. However, an Android SDK and API reference have been released recently which provide instructions on building and running examples of World Wind application on Android1. World Wind can also be expanded to include additional imagery and data whereas Google Earth is very limited in this terrain, using only commercial satellite imagery. If we look for real open source options to develop virtual globes, Google Earth is quite certainly not the best option. World Wind, however is open source and this has led to a proliferation of add-ons and plug-ins which are enhancing the power of virtual globes.

As of a sort time back, many other highly appreciated virtual globe engines can be found on desktop computers. WebGLEarth [9] is a valuable option consisting in a free software project that focuses on web browsers, using JavaScript and WebGL technology. ReadyMap2 tools are also based on JavaScript and WebGL. ReadyMap provides an easy-to-use map creation and publish using open standards. Neither of these two applications is native language nor computer platform dependent. They run on any device with a standard HTML5 compliant browser. A major drawback is that they still do not work properly on mobile devices, in part due to screen interaction problems and probably because the projects are still in their first versions. OpenWebGlobe

2ReadyMap: a complete suite of tools for publishing and viewing maps: http://readymap.com/
Another alternative worthy of attention is osgEarth\(^4\), a C++ terrain rendering toolkit that uses OpenSceneGraph. Although osgEarth does not consider mobile devices yet, OpenSceneGraph was released in 2011 to be used with handheld devices, supporting OpenGL-ES enabling both iOS and Android platforms. There is also increased interest in this field among the research community. Recent work on terrain navigation in mobile devices, still at the phase of prototype is presented in [7], and works efficiently on computational resources and network bandwidth. Several powerful World Wind based frameworks for desktop computers have emerged over the last few years, for example, iGlobe [3], and Glob3\(^5\). Glob3 is an open source 3D GIS multiplatform framework and can be viewed as the precursor of the Glob3 Mobile presented herein. A preliminary version of the virtual globe presented here, only available for mobile devices can be found in [11].

Nowadays, with the fragmented and rapidly emerging mobile platforms, applications must be developed to be portable to other platforms. It should be noted that, to our knowledge, there is no virtual globe that runs on iOS and Android mobile devices and on HTML browsers, with the exception in the latter case of Google Earth, which, however, is not open source.

All the virtual globes written in JavaScript and WebGL technology can run on some mobile devices, but always embedded within a HTML5 compliant browser. Therefore, the performance is not totally efficient. However, our project is written directly into native code for each device (Java for Android devices, and C in iOS). This allows for optimisation of the total performance over the application. In fact, and most especially on the IOS platform, the existence of a Java interpreter is unexpected. For this reason, virtual globes written in Java (such as NASA Worldwind) do not work on this platform. In addition, there is currently no WebGL support on the Safari mobile browser for iOS, and there is no version of Chrome or Firefox either at the moment for this platform.

With no integrated solution available for implementing virtual globes on iOS, Android and WebGL compliant browser, Glob3 Mobile presented here, offers an open source virtual globe, enabling a true 3D navigation experience and with the possibility of enhancing functionality through a user friendly API that allows for the testing and scripting of new features on the globe. Figure 1 shows the globe running on Apple iPhone and the Samsung Tab Android. Users benefit from a tool that enables them to reproduce their own virtual globe plus scrip. Instant access to the Glob3 Mobile is facilitated on the page project [1] and on popular iOS and Android popular Markets. In this paper we introduce both the free downloadable virtual globe application and the framework with user API that allows for the scripting of new features on the globe.

---

\(^3\)OpenWebGlobe SDK: http://wiki.openwebglobe.org
\(^4\)Terrain rendering toolkit: http://osgearth.org/
\(^5\)Glob3, an open source 3D GIS multiplatform framework:

![Figure 1: Glob3 Mobile running in (a) Apple iPhone and (b) Samsung Tab Android](image)

1.1 Development with graphics libraries

Android\(^6\) is an open source software project and operating system developed by Google for mobile devices. It is built on the open Linux Kernel and allows developers to write programming code in the Java language. In terms of graphic capabilities, Android currently supports OpenGL-ES 2.0 specifications. Lately, Android has been extended to other devices, beyond the smartphones, as example to notebooks and ebooks, proving to be a reliable and efficient open source operative system.

On the other hand, iPhone smartphones and iPad line tablets are designed by Apple Inc. They are powered by the iOS\(^7\) which is based on a variant of the same basic kernel that is found in Mac OS X. iOS also provides the OpenGL-ES framework which conforms to the OpenGL-ES v2.0 specification. If you want to contribute with a third party application to the iOS, this is only possible on payment of membership fee. On the contrary, the Android is free. Once a developer has submitted an application to the market, a control over its distribution is maintained, whether the application is free or not. Today, a surprising amount of innovative applications are being developed for mobile devices with amazing easy-to-use interface features. Mobile applications include entertainment, educational resources and research among others.

Computer graphics are rapidly advancing for mobile devices as a consequence of the improvement of mobile operating systems, i.e. iOS and Android. However, there are still some technical limitations to mobile devices that make it difficult to translate algorithms and programming strategies from desktop applications as they stand. In general handheld devices lack the CPU power and memory required to manage large data such as is needed for terrain representation. Fortunately, programming for mobile devices is improving considerably. OpenGL-ES (Embedded Systems) [6] is a well-

http://glob3.sourceforge.net

\(^6\)Android Mobile Operating System: http://www.android.com

\(^7\)iOS Mobile Operating System: http://www.apple.com/iphone/ios
defined subset of the well-known OpenGL 3D graphic API. It is especially designed for embedded devices including mobile phones, video game consoles and other handheld devices. OpenGL-ES provides a flexible and powerful low-level interface that facilitates software development through graphic acceleration. For a review of OpenGL-ES and other graphic library options for mobile devices see [8].

The WebGL (Web-based Graphics Library) is a graphic library based on the model of OpenGL ES 2.0, that extends the capacity of JavaScript to develop 3D graphics on any compatible web browser. WebGL code executes on a computer display card Graphic Processing Unit (GPU) and is embedded in a HTML5 webpage. The idea is that the WebGL could reach a satisfactory level of functionality enabling the 3D web application to become more and more independent of the platform. At present, the WebGL standard is still work in progress although impressive results are being given. There exist several WebGL based engines which allow programming graphical applications at the highest level (WebGLU, GLGE, C3DL, Copperlicht). However, due to the nature of our project, we chose to create our own engine directly using WebGL, as previous engines do not work with OpenGL ES on mobile devices.

The paper is organised as follows. Section 2 introduces the software aspects of the virtual globe, focusing on the architecture, object classes and details of implementation. In Section 3 we define the policy to represent the Earth globe, the outline of the LOD strategy and the underlying maths besides describing the 3D view interaction. The accessing and handling of earth images is described in Section 4, recalling access to the imagery data from public and WMS related servers, and explaining texture handling in mobile GPU. In section 5 we give a comparison with other existing solutions and close with some conclusions and proposals for future work.

2. THE FRAMEWORK ARCHITECTURE
At present and due to the nature of our project, three parallel developments are needed: (i) a project written in Objective-C and C++ with OpenGL-ES to run on iOS devices, (ii) a second project written in Java with OpenGL-ES for Android devices and (iii) a third project written in Javascript with WebGL to run on HTML5 compliant browsers.

On the one hand, it seems very complex to maintain three parallel projects. However, on the other hand, it is impossible to develop in a single language for all three platforms. The methodology used to develop the Glob3 Mobile thus was organised over the following sequential steps:

1. A full globe engine is developed using standard C++. This engine is built using virtual classes in order to deduce the specific details of each hardware (file access, network access, event handling, graphic library programming). Then, for each platform, a specific implementation is built, its own native language (Objective-C for specific classes in iOS, Java for Android devices, and JavaScript for the web version). Thus, by combining the engine with C++ specific classes written in Objective-C, a native application for iOS devices is obtained.

2. A commercial conversion software tool8 is used to convert the whole engine code from C++ to Java code. Some basic recommendations must be taken into account when developing C++ code, in order to achieve adequate performance on the part of the converter. This Java code, along with specific classes written in Java, produces the native application for Android devices.

3. Google GWT9 technology is used to convert Java into JavaScript. It includes all the usual syntactic elements of the Java language standard. However, it does not have all the shared features required for a desktop application, due to the inherent limitations of JavaScript. However, once these limitations have been taken into account, we can reuse almost 100%, using a subset of known Java classes to find their equivalent in GWT, and by minimising or eliminating the use of third-party libraries or classes. This code, along with specific classes written in JavaScript, allows us to produce the final application for web browsers.

To clarify the methodology, Figure 2 shows the life cycle of the development process, where code migration is indicated by arrows.

![Figure 2: Glob3 Mobile life cycle. The development process.](http://tangiblesoftwaresolutions.com)

Virtual classes are classified as follows:

- **IFileManager**: reads and writes files (only used in iOS and Android, but not for the Web version).
- **INetwork**: retrieves information from an url (images, binary, xml files).
- **IEvent**: captures specific events on every platform, including multitouch technology. Although this technology is not included for desktop browsers, our globe web version can be:

8[http://tangiblesoftwaresolutions.com](http://tangiblesoftwaresolutions.com)
9[code.google.com/webtoolkit/](http://code.google.com/webtoolkit/)
run on mobile browsers, where this technology is enable.

- **Location**: allows access to the compass (if any) and GPS (if any) of the device. Even for the web version, there is a proprietary technology that enables HTML5 browsers to obtain the location of a desktop computer via a standard API defined by the W3C organization. This web geolocation is performed based on the position of the telephone to which you are connecting the user’s computer, so the estimation is not as accurate as you get through a GPS device.

- **Image**: accesses image data in different formats (JPG, TIFF, PNG).

- **IGL (Interface Graphic Library)**: encapsulates all the functionality of the graphics library, OpenGL ES on the mobile device and WebGL on the web version. The OpenGL-Es specific classes have been developed using two different versions: 1.1 for older devices, and 2.0 for newer ones.

- **Timer**: Timing measurement (frame time, animation, time out of requests to network).

- **Utils**: several common utilities (print to screen, debug messages to the console).

Figure 3 shows the software layer model. The lowest layers are platform dependent, whereas the other layers are designed to be independent of specific platforms.

![Software Layer Model](image)

**Figure 3: Glob3 Mobile software architecture**

### 2.1 Application Programming Interface (API)

An significant challenge of the project is that users and developers can develop enhanced features in the form of scripts, thereby improving the functionality of the globe. The user API within Glob3 Mobile is under development. As an open source project, users can access provided engine functions which allow the addition of WMS layers, moving the camera, and also geometry on the globe.

The API allows for code to be written once and then makes the result available on all the platforms. JavaScript is chosen as the API programming language. Thus, for the WebGL engine version, the API code can be run directly. For Android and the iOS platform, JavaScript interpreters are integrated into the software architecture, with Rhino engine for Android and Apple JavaScriptCore for iOS. JavaScriptCore is an essential building block for WebKit, which in turn is the web rendering engine used by the Mac OS X and the iPhone OS. The Mozilla organization has released Rhino its JavaScript engine for Java.

Some short samples programmed with this API can be found on the project web. Below a sample of how a camera movement can be implement over the globe is shown:

```javascript
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);
}
```

![IDE Online](image)

**Figure 4: IDE online is accessible from the Glob3 Mobile project page.**

An integrated development environment (IDE) presently being worked on is available on the same page [1]. This IDE works entirely online, only requiring the JavaScript code, which is run to view the result on the virtual globe. Users can download the script to insert a Glob3 Mobile on their own web page. Figure 4 shows the IDE.

3. INTERACTION WITH THE GLOBE

We are dealing here with the Globe representation, LOD strategy and screen gestures for navigating with the globe.

3.1 Globe representation and LOD strategy

The Earth model most widely used is ellipsoid. The geographic coordinate system defines each position on the globe by longitude, latitude and height whereas the projection system used is WGS84. As with the World Wind application, see [2], due to the restrictions of the NASA World Wind server, we initially constructed the ellipsoid with a tessellation of $10 \times 5$ patches (tiles, this point forward), each one obtaining each $36^\circ \times 36^\circ$. At the same time, on each tile, a triangle mesh is generated using $n \times n$ vertices, where $n$ depends upon the screen resolution of the device, see [11].

A Chunk LOD system, see for example [4] and the references therein, is used to incrementally render the surface of the globe. We break the terrain into a quadtree of tiles, called chunks hereinafter. The root of that quadtree is a low-detail representation of the globe and the successive child chunks are new divisions of the globe into four-equal-sized areas that provide higher-detail of the terrain. A geometric error dictates in which portion in the quadtree the screen scene is displayed.

When the observer is located close to the ground, the number of visible tiles may be substantial (the LOD value is close to 18). We must use a LOD strategy that keeps the maximum number of visible tiles under a certain limit. To check if the current LOD is correct or not, a test on each visible tile is computed for every frame. This test indicates whether the tile should be changed to a more detailed level, subdividing it into four children, or to a less detailed level, grouping it with its three brothers. The LOD is then maintained using the classical efficient quadtree data structure.

For each tile $T$ the geodesic distance $d_T$ is calculated from this point to the centre of the tile. Whether a tile $T$ should be divided or not depends on the following relationship:

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

where $w_T$ is the current width of the tile, $D$ is the distance from the eye to the CPV, and $R$ is the radius of the globe. Unlike traditional screen-space pixel error, $\varepsilon(T)$ does not consider neither viewer frustum nor number of pixels of the tile. Technically, a benefit of $\varepsilon(T)$ is the gained accuracy for objects in the centre and around the center of viewport, whereas screen-space pixel error slightly underestimates the true error. After several experiments, chosen values for $\varepsilon_{\text{max}}$ and $\varepsilon_{\text{min}}$ were delimited to 1.1 and 0.9 respectively. Then, a tile is subdivided if the following two conditions hold true:

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

$$
\frac{w_T}{D/4} > \varepsilon \quad (2)
$$

where Equation 2 is forcing the highest LOD for the scenario. Two different situations of sample tiles are seen in Figure 5. The CPV point in said figure indicates the point of the highest definition and detail. The criteria to decide whether a visible tile $T$ should be subdivided or not depends on its width and the geodesic distance from its center to the CPV.

![Figure 5: The CPV point on the globe indicates the point of the terrain with greatest detail. The criteria to decide whether a visible tile $T$ should be subdivided or not depends on its width and the geodesic distance from its center to the CPV.](image)

![Figure 6: A terrain chunk (a) without skirts and (b) with skirts](image)
strip covering the four perimeters of the chunk. This triangular strip is then mapped together with the wrapping texture of the chunk, as shown in Figure 6. This solution is a very efficient technique, because it only depends on each tile, and not on its neighbors. Likewise the mesh is not readapted which could be very time-consuming were it the case. So skirts cover the holes formed by these gaps avoiding undesirable visual artifacts.

### 3.2 Screen gestures and interaction

To help the applications detect gestures, the iOS introduces gesture recognizers. However, the Android systems lack gesture event handling similar to that of the iOS that facilitates gesture detection. Thus to normalise programming over both platforms, we developed our own control of gesture events, using only two low-level gesture events: *finger-down* and *fingermove*. We go on to describe the types of gestures introduced in the Glob3 Mobile. Readers may test the performance and quality of such gestures by testing the free application available at [1] and also downloading it from popular iOS and Android popular *Markets*.

To clearly describe the geometrical meaning of the gestures, we give detailed explanations of the following gestures in Figure 7:

**Tapping.** Normally, tapping is the first event before other gestures, such as dragging one or more fingers across the screen. The first step is always to find the unprojected point on the globe surface. This option is also useful for future functionalities, such as obtaining information with respect to the location (height, geographic coordinates), or picking out some object of the terrain. When the user taps on the screen with one finger, point $P_0$ is defined. If the ray starting from $c$ and passing through $P_0$ is projected back directly to the earth globe, we have point $P_0$.

**Panning.** The user moves the fingertip over the screen without losing contact. The globe continues to rotate where the finger is in contact, see Figure 7 (b). The rotation axis is given by the cross product of vectors $\overrightarrow{OP}_k$ and $\overrightarrow{OP}_0$ where the rotation angle is obtained from the dot product. If the finger is released while dragging across the screen, a short few-second animation occurs continuing the rotation at a decreasing angle until the globe comes to a halt or until the screen is touched again. This results in a pleasant and attractive visual effect.

**Pinching.** The user touches the screen with two fingers and brings them closer together or moves them apart. A zooming effect is attained by moving the camera along said direction. The goal is displaced by a magnitude proportionate to the points where the fingers touched down on the screen, keeping approximately in contact with the finger movement. The idea is similar to stretching a 2D image. The distance the camera is displaced is given by the length $\overrightarrow{OQ}_0$, see Figure 7 (c). This magnitude is calculated from the two rays given by $c$ and passing through the points given respectively by fingers $P_k$ and $Q_0$. These rays are projected back onto the globe giving the points $P_0$ and $Q_0$. Moreover, two other points $P_k'$ and $Q_k'$ are produced when the user continues to drag their fingers. Then, the new camera position is estimated along its view direction, searching for the position that verifies that the projection of the original points $P_0$ and $Q_0$ after displacement are close to $P_k'$ and $Q_k'$.

**Double tapping.** If the user quickly touches the screen twice with one finger, similar to the mouse double click, the resulting action will be a combination of panning and pinching. The visual effect is a smooth two second animation where the point on the terrain touched by the user moves towards the centre of the screen, and simultaneously, the camera moves in a little.

**Rotation.** To rotate the globe, the user touches the screen at two points, and then rotates them around themselves on the screen, as seen in Figure 7 (d). The starting and finishing points of the rotation define two rays and then it is straightforward to obtain the rotation angle via the formula:

\[
\frac{|\overrightarrow{P_0Q_0} \cdot \overrightarrow{P_kQ_k}|}{|\overrightarrow{P_0Q_0}| \cdot |\overrightarrow{P_kQ_k}|}
\]

**Swiping.** This gesture is produced when the user touches the screen with two fingers, giving points $P_0$, $Q_0$ and drags them parallel to each other without losing the contact through to the final position $P_k$, $Q_k$. The desired globe movement is obtained by rotating the camera around the globe with respect to an axis that depends upon weather the swiping is vertical or horizontal. When the finger movement is on the vertical the globe is rotated through the horizontal axis parallel to the screen, whereas the horizontal swiping implies rotation through the vector normal to the surface earth, see Figure 7 (f). Both rotation axes are positioned at the $F$ point on the terrain, that is, the unprojected point representing the centre of the screen.

Regarding the web browser implementation, desktop and mobile browsers should be distinguished. For desktop browser we use the typical mouse and keyboards events:

- **Tapping with one finger:** captured as a mouse click.
- **Panning with one finger:** captured as a mouse dragging.
- **Pinching with:** captured as a wheel mouse movement.
- **Double tapping:** captured as a double click.
- **Swiping:** captured as synchronous mouse dragging and shift key pressed.

Meanwhile, for mobile browsers, an incipient standard declaration for multitouch events is included in W3C. For example in the Safari browser those events work properly. However WebGL is not as yet implemented on Safari. Mozilla Firefox, however, does not support this standard. Although Firefox has its own event implementation, our experience of date is that it does not work properly. Thus, an open problem is to handle multitouch in web browsers.
4. INTEGRATING MAP IMAGES

The Open Geospatial Consortium (OGC), an international voluntary consensus standards organization, works to enable more usable and useful geographic information in GIS related applications. For example, users can access the enormous amount of available internet data layers by means of the Web Map Service (WMS), as it provides map images from almost everywhere on Earth.

4.1 Access to Web Map Services

The Glob3 Mobile supports access remote data repositories from any public WMS layer. OGC standards dictate the definition of any WMS request which is then simply encapsulated in a URL address and finally launched to the server via, for example a web browser. This example shows the request of a tile image corresponding to the island of Gran Canaria, Spain, see the requested image at Figure 9 (a):

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

Maps are dynamically handled thanks to a LOD algorithm. When a tile must be subdivided attendant to some error criteria, four new tile children are required and so four server WMS requests are triggered. To select an appropriate server, a list is visited sequentially, until a server is reached that totally covers the bounding box of the tile. The last server on the list is always the NASA World Wind server, because that server covers the whole world with adequate resolution and over several levels of detail. The NASA World Wind server consists of of two different servers, [2]. The first one is the Blue Marble WMS Global Mosaic, that includes the first levels of detail for images from the whole globe (from LOD 0 to LOD 3). The second one is the Virtual Earth Tile Server, that is used for higher detail images. This server is the backend for Microsoft Map Web services. It is not a standard WMS server, but works with fixed location images of predefined size. The first level of this server if composed of a mosaic of 160 × 80 images, that represents areas of 2,25 × 2,25 degrees. Therefore, in order to match the resolution of the Virtual Earth tiles, our globe starts with an initial mosaic of 10 × 5 tiles at LOD 0.

To avoid delaying the navigation, network requests are made in an asynchronous fashion, using multi-threading. Thus, several requests can be sent at the same time, while the user continues to navigate the globe. Obviously, each tile is
not subdivided into its four children until all of their textures have been achieved. Experiments have validated this technique for several bandwidth connections, as an adequate navigation experience.

It is worth noting that some WMS server does not allow many simultaneous requests. Thus, only the highest priority tiles according to distance from the observer are requested to the server. Moreover, no more than 50 tiles are allowed to be sent simultaneously, which is not proven to be optimal although works fine for the tested cases.

### 4.2 Handling network requests and cache

An issue deserving attention is how to handle network requests. They are the basis of client-server communication on the virtual globe. A virtual class called the $INetwork is devoted to this purpose, where separate treatment is given to iOS, Android and WebGL. For example, the $NSURLConnection and $NSURLConnection methods are available on Android and iOS respectively. This leads us to asynchronous threads for network operations. The network connection automatically schedules the downloading of images and then callbacks are received as it progresses. Using asynchronous operations reduces the concern with respect to blocking or delaying the virtual globe application and with creating multiple of requests. A queue per operation is created that can easily be controlled as for example in cancelling downloading in progress or to adjust the number of operations we allow to be executed in any given bandwidth scenario.

Less intuitive is the treatment of requests on HTML browsers, as JavaScript is a language not truly designed for handling internet images. Fortunately, in most situations, there is no need to edit images as in the access to particular fields, operating or transforming images. In practice, JavaScript is used to simply download a remote image from its URL.

### Transfers and security domains

Cross-domain security has to be dealt with when downloading and applying WMS textures to our globe. The problem arises in general when any type of resource or content is obtained from a remote site. The problem is that it is considered unsafe for a client to attempt to request server data not located in the same domain of the client. This is probably the most serious restriction on applications that access maps on external geolocation servers. Cross-origin resource sharing (CORS) is a web browser technology specification that defines ways for a web server to allow its resources to be accessed by different web domains. While widespread use is expected, unfortunately CORS implementation currently depends on enabling these features by services provider.

Meanwhile, the solution to this problem is to place a proxy on our server which is responsible for relaying the requests between the two parties (the remote server and the local client). This proxy can be implemented in many ways: from a simple PHP script, to systems developed specifically for these situations, such as the MapProxy tool. For each solution, the advantages and disadvantages must be evaluated to come to a decision on the best option. For example, a tool like MapProxy is an open source proxy for geospatial

4.3 Dealing with textures in mobile GPU

Some limitations on graphic hardware in mobile devices make it impossible to use normal compressed image formats such as PNG or JPG that require hardware-dependent compressed formats instead. For example, the iPhone only supports a specific type of compression called PVRTC, supported by a PowerVR chip, the iPhone's graphic processor. The problem however is that a desktop computer is needed to compress to this format, because there is no available source code to carry out the compression in the iPhone hardware. The data. It is very powerful and rich in multiple configuration options. However, our tests revealed that MapProxy may originate time-consuming tasks, especially for mobile device hardware. In addition, one needs to explicit the list of WS servers to be accessed in the file configuration, which is not appropriate when our requirement is that users add and remove WMS layers at runtime. Thus, a PHP script is implemented, producing a simple solution to overcome security domain issues.

It was also noted that some WMS servers did not seem to accept resource requests from web browsers. To solve that problem, additional code was introduced into the PHP script to remove the HTTP headers of any information relating to a regular browser petition, such as the User-Agent parameter. This is an example of the versatility of the PHP scripts. Implementing this functionality in MapProxy is impractical.

Handling cache memory

Tiles are cached to disk to improve performance. Thus if a tile is removed from the quadtree, because the observer has navigated to a different place, on return to the same place, the texture is first searched in the cache before sending the request to the WMS server. This allows efficient and fast visualization as has already been proved in many other visualization systems.

In the iOS and Android platforms, implementing a cache for the downloaded tiles of the globe is straightforward. The globe engine thus developed creates a temporary memory cache where the downloaded tile images are stored. This allows for further use of the image content instead of launching new network requests. With a web browser it is possible to use the existing browser mechanisms of web cache instead. HTTP defines three basic mechanisms for controlling cache: (i) Freshness, (ii) Validation and (iii) Invalidation. We use freshness, where a max-age directive is set up indicating for how many seconds the response is fresh.

The following PHP code is used to request the browser to place an image on its own cache.

```php
if(!$validContent) {
$response_status="HTTP/1.1 404 Not Found";
$content="Data error";
} else {
$response_status="HTTP/1.1 200 OK";
$content=strr($response,$Header_length);
$response_headers[]="Content-Lgh: ".strlen($content);
$response_headers[]="Cache-Control: max-age=$maxTTL"
}
```

4.4 MapProxy Tools: http://mapproxy.org/
iPhone SDK only comes with a command-line program that must be used to generate PVRTC data. Thus, conversion of images must be carried out first and then attached to the iPhone project.

Obviously, this solution of command-line compression is not at all useful. The WMS server only responds to well-known formats, such as JPG and PNG, and we are be forced to compress images on the fly. For this reason, we read the pixel data at the original uncompressed 24 bit color values, and convert them to 16 bit representation. This simple image compression has proved to be successful on the tested devices without detriment of visual quality.

We use $256 \times 256$ textures for each tile, with 16 bits/pixels, that must be stored in GPU memory (130 Kbytes approximately). After several tests on various different smartphones, a number of 300 textures in the GPU memory has proved to be acceptable for most. In order to keep the maximum number of visible tiles close to this value, we have included this parameter in the LOD strategy.

After the network petition, a JavaScript image object is obtained. In JavaScript, we can access individual pixels using the image object, but it is slow. Instead, the image object is passed directly to the GPU, and the hardware is responsible for making JPG decompression and accessing pixel data.

Transparency or alpha channels are used frequently and thus are supported on our framework. Some WMS textures include this channel to indicate that some areas on the maps must not be displayed, see Figure 8. In this case, two textures are required for each tile. In the JavaScript version, this is done using webGL multi-texturing in the GPU. But this solution is not suitable for mobile platforms, because it needs double memory requirements in the GPU. For this reason, the merge of both textures is produced in the CPU, to maintain the maximum value of 300 textures in the GPU memory.

4.4 Including elevation data

The elevation data are also obtained using a WMS server from NASA World Wind. This server uses a specific format (BIL) to return the elevation data. Every pixel in the resulting image (named height map) is a 16 bit signed value, that indicates the height in meters of the corresponding point. Then, each tile is composed by a triangle mesh of $n \times n$ vertices, and a BIL image of $n \times n$ pixels. With this image, each vertex on the mesh is translated in the normal direction to the globe surface as many meters as indicated in the height map. After several experimental results, we have seen that the precision of height maps is slightly inaccurate. To obtain more precision in the elevation values close to the tile borders, it is better to send requests that include a wider area. For this reason, if the tile has a mesh resolution of $n \times n$, we ask for a height map with a resolution of $3n \times 3n$, that includes an area three times larger (in both dimensions). In this way, the obtained accuracy in the $n \times n$ inner area of the tile is more precise.

![Figure 8: Displaying a transparent texture (street map of a city) merged with an aerial photo.](image)

After several experimental results, we have seen that the precision of height maps is slightly inaccurate. To obtain more precision in the elevation values close to the tile borders, it is better to send requests that include a wider area. For this reason, if the tile has a mesh resolution of $n \times n$, we ask for a height map with a resolution of $3n \times 3n$, that includes an area three times larger (in both dimensions). In this way, the obtained accuracy in the $n \times n$ inner area of the tile is more precise.

As browsers may have different character configurations (ASCII, UTF-8 etc.), we had to face the problem of character encoding in interpreting image data. To overcome this difficulty, we set up the JavaScript application to work with the same character encoding over all the browsers.

JavaScript is a language that was not designed to deal directly with binary data. It was originally designed to deal with the elements of a web page, which are mostly textual objects. The images could be obtained using the JavaScript image object, but the process needed to obtain other binary...
data is more complex. When a remote resource is requested in JavaScript, the browser assumes that the response must contain text. Then, this text data is translated to the current character set used by the browser. Therefore, there is no assurance that the original byte values that have been received are the same as those that were originally sent, depending on the encoding scheme used by the client. It is not possible to assure that they are in a range between 0 and 255, as many of the character sets used in modern browsers use more than one byte to represent information.

A solution for overcoming these difficulties is to force JavaScript to ignore the configuration of the browser, and to use a character set known to make the conversion. This strategy would make it possible to receive binary information and would solve cases such as BIL files.

5. FINAL REMARKS AND FUTURE WORK

Virtual globes integrating environmental data at any time or place, represent a challenge within the technical constraints imposed by mobile devices. We describe here the main software components for an easy design and implementation of a virtual globe together with user-specific enriched features on iOS, Android and WebGL compliant browsers. Excepting Google Earth, which however is not open source, there is no virtual globe -to our knowledge- that runs on iOS and Android mobile devices and on HTML browsers.

Our server-client approach provides a highly scalable architecture, capable of dynamically balancing the workload of the textures and height fields. Careful attention has been paid to the 3D user interaction on multitouch iOS and Android platforms. We have tested the application on a dozen different mobile devices, leading to analogous (real-time) performance for an average of internet bandwidth wifi connection between 10 kbps/s and 100 kbps/s.

The problem of embedding the 3D application within a HTML browser is also covered in this paper. The main benefit here is the pursuit of independence of the platform on hand. For compliant WebGL browsers, we have solved some multitouch screen interaction and disk cache limitations. When browsers try to access remote data, a PHP script is implemented producing a simple solution to overcome security domain issues. New enhanced features to the globe can also be developed thanks to our IDE, freely available on the project page. We offer an API that allows for code to be written depending on the coding scheme used by the client. It is not possible to assure that they are in a range between 0 and 255, as many of the character sets used in modern browsers use more than one byte to represent information.

Glob3 Mobile can be instantly accessed on the page project [1] and also from popular iOS and Android Markets. The near future of Glob3 Mobile lies in adding the new and demanding requirement of the Geospatial community such as display of large clouds of points, streaming video or including geographic formats such as KML, WFS or GML. The enhancement of the user API so that additional functions may be added is of growing interest to the market.

Acknowledgements

This work has been supported in part by Spanish firm IGO software and by the CYCIT Project MTM2008-05866-C03-02 of the Spanish Ministerio de Educación y Ciencia.

6. REFERENCES


Table 1: A comparison of features in available virtual globes

<table>
<thead>
<tr>
<th></th>
<th>Op. Source</th>
<th>iOS</th>
<th>Android</th>
<th>Native C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Earth</td>
<td>×</td>
<td>ν</td>
<td>ν</td>
<td>ν</td>
</tr>
<tr>
<td>WebGL Earth</td>
<td>ν</td>
<td>×</td>
<td>ν</td>
<td>×</td>
</tr>
<tr>
<td>OpenWebGlobe</td>
<td>ν</td>
<td>×</td>
<td>ν</td>
<td>×</td>
</tr>
<tr>
<td>World Wind</td>
<td>ν</td>
<td>×</td>
<td>ν</td>
<td>ν</td>
</tr>
<tr>
<td>Glob3 Mobile</td>
<td>ν</td>
<td>ν</td>
<td>ν</td>
<td>ν</td>
</tr>
</tbody>
</table>

Table 1: A comparison of features in available virtual globes

- **Google Earth**: × (not available), ν (available)
- **WebGL Earth**: ν (available), × (not available)
- **OpenWebGlobe**: × (not available), ν (available)
- **World Wind**: ν (available), × (not available)
- **Glob3 Mobile**: ν (available)
