A Multi-Client Architecture for Hybrid Terrain Rendering on Mobile Devices

J.M. Noguera, R.J. Segura, C.J. Ogáyar
University of Jaén
Escuela Politécnica Superior
Jaén, Spain

R. Joan-Arinyo
Universitat Politècnica de Catalunya
Grup d’Informàtica a l’Enginyeria
Barcelona, Spain

May 31, 2010

Abstract

Mobile devices such as Personal Digital Assistants (PDAs) or smartphones are rapidly increasing their graphics and networking capabilities. However, real-time rendering of large terrains is still a challenging task to accomplish in such limited devices. In this paper we describe the principles involved in the design and development of a scalable client-server architecture for hybrid rendering of terrains over wireless networks on mobile devices.

We have developed a hybrid adaptive streaming and rendering method based on a server-client approach. The rendering workload is distributed between a server and the clients and the terrain is partitioned into the close-range geometry and the background. The close-range geometry is downloaded from the database and rendered on the mobile client, and the background is portrayed as a view-dependent panoramic impostor and rendered by the server on demand. The system architecture is organized in three levels: the main server, the panorama server, and the mobile device client. This architecture provides support for efficient delivery of geometry and impostors to mobile clients according to their capabilities.

As a proof of concept, we have implemented a prototype and carried out exhaustive experiments considering different network scenarios and different number of connected clients. The analysis of the server workload and response times shows that our architecture achieves a great scalability and performance even when using low-end mobile devices.
**Keywords:** Terrain navigation, Terrain rendering, Mobile computing, Adaptive streaming, Image-based rendering, Hybrid rendering.

1 Introduction

Mobile devices are becoming both ubiquitous and every day more powerful. In fact, nowadays most high-end smart phones include a surprising graphics power and, in many cases, Global Positioning System (GPS) capabilities. Interactive rendering of large terrains on mobile devices play an important role in a number of graphics applications including mobile games, personal navigation, Geographic Information Systems (GIS), access to location based data and visualization of maps. However, the field of mobile applications still suffers from severe limitations. Computational resources in mobile devices are sparse, both main memory and secondary storage are limited, wireless networks are costly and slow, and displays are small. In general, their computational power is an order of magnitude smaller than the hardware commonly used in today desktop PCs.

Large-size terrain rendering is an extremely resource-demanding application, and the limited capabilities of mobile devices coupled with an even more limited bandwidth, force the rebirth of research on distributed rendering techniques. Many research has been done recently in the area of distributed rendering on mobile devices. Most solutions are based on an indirect rendering, where the 3D geometry is rendered in a dedicated rendering server, and the resulting images are then transmitted to the user. These techniques, although akin to very thin devices, require a powerful server and easily lead to network congestion. Moreover, the graphics capabilities of modern mobile devices equipped with graphics accelerator (i.e., the iPhone or the Nokia N95) are wasted.

In this paper we report on a novel client-server architecture for adaptive streaming and rendering of large-terrains on mobile devices which makes use of a hybrid rendering approach [19]. This approach provides tools for enhancing both the quality of rendered images and the interactivity when rendering large terrains on mobile devices using low bandwidth wireless networks. The clients are in charge of rendering the terrain close to the viewer, whereas the terrain in the background is portrayed as a panoramic impostor and is rendered on demand by a remote server.

The architecture consists of three different components: the Client, the Main Server and the Panorama Server. The Client is run in the mobile device and manages the user interface. The Main Server is in charge of handling all the requests of the clients. Whenever a client requests a chunk of terrain, the server retrieves it from a GIS database and sends it back to
the client, which uses it to render the nearby terrain. The Panorama Server provides compressed panoramic impostors on request which are streamed to the client.

The proposed framework supports a wide variety of clients, from low-end mobile phones to desktop PCs. Our architecture dynamically splits the rendering task between the server and the client according to the computing capacity of the client and the network load. Besides, the quality of the provided impostors is adapted to match the display capabilities of each client. Contrarily to server-based approaches, our architecture does not require of a powerful and expensive hardware on the server side. The hybrid rendering approach exploits the rendering capabilities of modern mobile devices, thus reducing the server workload and, consequently, improving the scalability of the system. Our experiments demonstrate that a commodity PC equipped with an accelerated graphics card is able to provide service to a large number of concurrent clients.

The rest of the paper is organized as follows: Section 3 summarizes the principles of the hybrid rendering approach used in this paper. Section 4 presents a general overview of the proposed architecture, while Sections 5, 6 and 7 fully describe its three main components. Section 8 proposes a real-time 3D locating system which makes use of this architecture. Section 9 presents and discusses experimental results. Finally, Section 10 summarizes results of our research and gives a vision of future work.

2 Previous Work

In general, current client-server 3D rendering methods can be classified into three major categories, according to where the geometry-rendering tasks is performed [17].

2.1 Client-Side Methods

In client-side methods, the rendering task is delegated to the client, which downloads the geometry from the server. They do not involve any rendering on the server side. This approach reduces both the server and the network load, but the client must provide the computational power required to render good quality and to manage a depth enough viewing distance.

Most existing techniques dealing with streaming and rendering large terrains fall in this category. For a detailed survey, we refer the reader to [21]. It is worth remarking that most of these solutions were designed for desktop PCs and present high CPU cost or rely on the programmable GPU, see [16, 1, 15, 6] just to cite a few. Since mobile devices, in general, do not
feature programmable GPUs, these techniques are still impractical.

Rendering and streaming of large terrains in mobile devices is still a largely unexplored field. [23] proposed a solution specifically designed for mobile devices which partitions the terrain data into a simple grid of tiles. The tiles around the viewer are transmitted to the client and rendered using a pre-computed triangle strip. Authors claim that they have managed to render a terrain of 3744 triangles at 7 fps (frames per second) on a PDA using a 480 Mbps USB 2.0 network. This approach, although fast, is very crude and presents some drawbacks. No multi-resolution data structure is used, and cracks are not avoided.

2.2 Server-Side Methods

In server-side methods, a dedicated remote rendering server is in charge of performing the geometry-rendering task and streaming images to a client over a network. Although these techniques are appropriate for thin mobile devices, they have some drawbacks. There techniques can easily cause network congestion and they require servers with high-performance graphics capabilities, especially if the number of clients concurrently connected increases.

The problem of streaming images is well-known in the literature, and several solutions have been proposed. [11] proposed a remote visualization system based on streaming compressed images to the client. They claim to achieve a speed of 5 fps using a 802.11b radio interface on a PDA. [13] presented a cluster-based remote rendering framework based on MPEG video streaming. They claim to achieve remote visualization on a PDA at 30 fps and $240 \times 240$ resolution using a rendering cluster of eight PCs. However, the authors admit that this cluster is not powerful enough to simultaneously provide two clients with two different video streams. A rather similar technique targeting terrain navigation was presented by [28]. Unfortunately, the paper provides a rather shallow description of the technique, and measures of the client performance and network usage are not reported.

[4] and [22] have presented alternative methods for image-based approaches by using scheduling mechanisms and partial streaming of images. However, these approaches severely limit how the viewer can move and do not perform well in dynamic scenes.

2.3 Hybrid Methods

In hybrid methods, the model is partitioned into parts that are rendered on the server and parts that are downloaded and rendered by the client.
These methods reduce the geometric complexity of the data transmitted by replacing parts of it with images.

Interactively rendering large terrains on resource-limited mobile devices connected to low bandwidth wireless networks is a challenging task. In [19] we reported on a hybrid rendering approach that successfully solves the problem. Our efforts centered on formalizing the hybrid approach and on studying its impact on the client side. On the contrary, this paper focuses on the server side, and proposes a completely redesigned architecture which significantly improves the server performance. This paper also addresses issues that were not considered previously, such as multiple clients support and its impact on the server performance, response times, network latency and adaptation of the panoramic impostors to the display capabilities of the client.

3 The Hybrid Approach

For the sake of completeness, we briefly recall our hybrid rendering approach here. This approach distributes the 3D rendering workload between a mobile client, usually with very limited resources, and a remote server, generally featuring high-end hardware and software resources. The server stores the complete dataset and it is responsible for providing the clients with small chunks of terrain close to the users’ position, and also for rendering and sending to the clients impostors for the terrain in the background. The clients are in charge of rendering the terrain close to the user’s position, displaying the panorama that replaces the actual distant terrain, and request from the server updates for both, the close range geometry and the impostors.

3.1 Terrain Representation

Since available CPU and memory resources in mobile devices are limited, adaptively streaming and rendering large-scale terrains on mobile devices require using specifically adapted algorithms and data structures.

Our algorithms and data structures have been designed aiming at simplicity, efficiency and scalability. We organize the terrain representation according to two different levels. The first level subdivides the complete terrain height map in a regular grid of equal size tiles, each tile covering a squared area of the height map. The second level consists of a set of restricted quadtrees, [25, 20], each quadtree associated with one terrain tile. Textures associated with the terrain are also structured according to a grid of quadtrees defined as before.

This structure is suitable for progressive data transmission [20, 14] over
Figure 1: Synthesis of the nearby terrain rendered by the client and the panorama rendered by the server.

wireless links. It is also optimized for fast rendering on mobile GPUs using indexed triangle strips, [19].

3.2 Panoramas

In our approach, view-dependent impostors are used to portray the terrain located far from the viewer, which are rendered by the server on demand and streamed to the client. These impostors consist of two-dimensional synthetic images that simulate a wide view of a physical terrain placed in the background far from the viewer. These impostors are called panoramas [3].

In order to visualize a panorama, it is first projected on the inner six faces of a cube centered at the viewer. Panoramic images projected on a cube are usually referred to as skybox [26] or environment map [2]. Since screen resolution of mobile devices is small, usually in ranges like 320×240 and 640×480, panoramas do not need to be generated at high resolution, thus saving bandwidth. The resulting image is composed by the client by merging the terrain and the panoramic impostor as illustrated in Figure 1.

The panoramic impostor, rendered on demand by the remote server, and the close-range geometry, rendered locally by the client, should be correctly matched to avoid visible discontinuities and artifacts. Therefore, we split the terrain into nearby terrain and panorama as follows. Let the view volume in the client be limited by the front and back clipping planes placed respectively at $z_{\text{front}_c}$ and $z_{\text{back}_c}$ distance from the viewing point. Similarly, let the view volume in the server be limited by the clipping planes placed at distances $z_{\text{front}_s}$ and $z_{\text{back}_s}$. Then, we require that $z_{\text{front}_s} = z_{\text{back}_c}$, that is, the front and back culling planes in the server and client respectively are coincident. See Figure 2. Clearly, the client renders the close terrain whereas the distant terrain is culled. On the contrary, the server only uses the distant terrain to render the panorama.

As long as the viewer does not move, the panorama remains valid. Under a perspective projection, a small move of the viewer causes a small displacement of the projection of distant parts of the scene. Given this large
temporal coincidence, it is wasteful to update the panorama for every small movement of the client. Nonetheless, if the viewer moves and the panorama is not properly updated, the displayed image is not longer correct. To fix the display, our approach updates the panorama whenever the error exceeds a predefined threshold. In this way, the hybrid approach makes a better use of the available computational resources and network bandwidth, which is crucial to wireless streaming applications on thin devices.

4 The Framework

The aim now is to define an architecture for the hybrid approach able to allow a variable number of clients to be connected simultaneously to the server. The higher the number of clients that can be served, the better the system scalability.

The architecture developed is illustrated in Figure 3. It consists of three software components: the Main Server, the Panorama Server and the Client Server. The Client runs in the mobile device and manages the user interface. It displays the terrain according to its computational capabilities and the network load. The Main Server runs in the server and is in charge of handling all the requests of the clients. Whenever a client requests a chunk of terrain, the server retrieves it from a GIS database and sends it back to the client, which uses it to render the nearby terrain. The Panorama Server runs in the server and provides compressed panoramic impostors on request which are streamed to the client. Since each component has been designed as an independent application, the system works even if no Panorama Server is present. In this case, any panorama request issued by the client is simply discarded and the system works like a standard client-side render architecture.
In the following sections we describe in detail each component.

5 Main Server Architecture

For the Main Server, we propose a multi-threaded architecture as illustrated in Figure 3. The data flow is managed by a master thread that listens to a network socket, waiting for incoming clients. A typical session can be described as follows. When a client connects to the server, a new Server Instance is created with an associated socket and a connection is established with the client. The Server Instance stays alive until the connection is closed or the server application dies. Therefore, multiple clients can be connected to the server at the same time with a dedicated Server Instance per client.

The client-server communication consists of a simple request-response protocol. A request can query a quadtree node or a panorama from the server. The server then issues a response message, which provides the requested data to the client. When the client needs to download a chunk of terrain from the server, it sends a quadtree node request to the server. In response, the Server Instance retrieves the height values and the associated texture from a GIS database and sends it back to the client. Panorama requests receive a slightly different treatment. Once the panorama request reaches the Server Instance, it is passed to the Panorama Server, which will in turn provide a new panorama to the Server Instance according to the position of the camera in the client. This panorama is eventually sent to the client, which will use it to portray distant terrain.

The Server Instance has been designed following a multithreaded paradigm, in which communication and processing are performed in different threads. The Server Instance comprises three modules, namely, the Requests Queue module, the Data Encoder module and the Server Module. See Figure 3.
The first two modules deal with network transmission, while the third one runs in a separate thread and drives the internal logic of the server.

The Requests Queue module is responsible for listening to the socket and placing all the incoming requests from the client in an input queue. The Server Module is the core module of the Server since it manages the data flow across the Server. It extracts the client requests from the Requests Queue according to a first-in first-out scheme and provides adequate responses to the Data Encoder. If the Server Module needs quadtree nodes, terrain textures or vector data, it will access to a GIS database to gather the data. Panorama requests, however, are passed to the Panorama Server. Finally, the Data Encoder module receives processed responses from the Server Module, and it is responsible for packing them according to our network protocol, and writing the packet in the network socket.

6 Panorama Server Architecture

The Panorama Server is responsible for rendering and encoding all the panoramas which are requested by the multiple Server Instances. Concurrent panorama requests are handled by a first-in first-out scheduling system. A panorama is built by the Panorama Server by projecting the distant terrain on a framebuffer. Once it is rendered, the image is copied from video memory to main memory. The raw images are compressed using any standard image compression algorithm suitable for distribution to mobile devices, such as JPEG. The coded panorama is then delivered to the Server Instance which requested it, and finally sent to the client.

6.1 The Panorama Renderer

As stated in Section 3, the hybrid rendering approach splits the rendering workload between the client and the server. The Panorama Renderer is responsible for carrying out the rendering workload of the server.

The construction of a cubic panorama is straightforward [4]. Each face of the cube covers 90 degrees of view both horizontally and vertically. The panorama is built by the Panorama Renderer by placing the camera referred to the viewer coordinates in the client and making use of the terrain nearby. Then six orthogonal images are rendered. Finally, the resulting images allocated in the framebuffer are copied from video memory to main memory. As soon as the raw images are copied to main memory, they are placed in a coder queue waiting for their turn to be encoded by the Panorama Encoder.

In practice, only the four lateral faces of the cubic panorama have to be rendered and streamed to the client. As we are working with height maps,
there will never be any terrain above the user’s position. Also, the bottom face of the cube remains hidden by the close-range terrain rendered by the client.

Most 3D graphics libraries, such as OpenGL/OpenGL ES and Direct3D 10 [18], are not thread-safe. That is, in a multi-threading environment the graphics context is bound to a specific thread, making it impossible for multiple threads to access to the graphics API. In our architecture, the graphics context is bound to the Panorama Renderer thread. However, if there are more than one graphics adapter in the server, one instance of the Panorama Renderer can be allocated per GPU. As each instance runs in a separate thread, multiple graphics contexts can be bound simultaneously, thus allowing the server to concurrently render multiple images.

6.1.1 Adapting Panoramas to Client Capabilities

Our server aims at supporting a wide variety of clients. Therefore the Panorama Renderer must render panoramas with resolution according to the capabilities of the clients without incurring in any extra costly operation. This issue can be solved in a simple but efficient way by using an OpenGL extension known as framebuffer object (FBO).

A FBO consists on a collection of logical buffers each with and associated state and defines where the output of OpenGL rendering is directed. FBOs provide a mechanism for performing off-screen rendering on an attachable image buffer instead of on the standard framebuffer provided by the window system. Attachable image buffers are known as renderbuffers and that consists on a color buffer and a depth buffer. FBOs also provide an efficient mechanism to detach and attach renderbuffers to them. Switching between attachable renderbuffers is a simple and fast operation and does not require switching between different contexts or FBOs.

Our approach to adapting panoramas proceeds as follows. We first create a FBO and a set of renderbuffers at different pre-defined resolutions.

Since most mobile graphics libraries only support square shaped textures with side lengths expressed as integer powers of two, we have predefined a collection of attachable renderbuffers at four different resolutions, say, 128×128, 256×256, 512×512 and 1024×1024, that cover a wide spectrum of clients, from mobile devices to desktop PCs. At run-time, the desired renderbuffer is attached to the FBO according to the needs of the client, see Figure 4. After performing the rendering, the renderbuffer contains the scene at the desired resolution, and can be copied to main memory. The number of renderbuffers and the resolution used are parameters that the user can adjust according to the actual needs.
6.1.2 Rendering the Terrain from Multiple Viewpoints

The multi-client nature of our architecture rises another issue that has to be addressed. Most terrain rendering approaches exploit frame-to-frame coherence to avoid complex re-meshing and re-transmission of the terrain. However, this is impracticable for us, since in a multi-client environment, subsequent requests of panoramas are likely to belong to different clients that might be navigating over different terrain zones far from each other. Moreover, different clients may move at different speeds and in arbitrary divergent directions.

For our system to be scalable, the architecture should allow a large number of clients to be connected simultaneously to the server. Therefore, managing an individual representation of the terrain per connected client is impracticable. Thus, defining a data structure capable of sharing terrain data used for rendering panoramas requested by different clients is paramount. Also, this structure should be capable of computing the terrain triangulation in a fast way, regardless of the viewer position in the last rendered frame.

To overcome this issue, we use an approach based on an implicit restricted quadtree structure, [15]. First, the height map and the texture map are partitioned into a regular grid of squared size tiles. The partitioning described in Section 3.1 for building the terrain representation applies here. Then, the height values and the texture map of the area covered by each tile are stored, respectively, as a two bytes per component texture, plus a RGB texture. This set of textures is cached in GPU memory.

In each frame, the Panorama Renderer chooses a set of tiles which are visible from the current viewpoint. Then, a restricted quadtree is built on the fly for each selected tile by recursively dividing it into four equal tiles. This structure does not store any type of geometric information. Instead, the recursive traversal simply figures out the position and size of each quadtree node with respect to its parent node by using shift operations.

Each node is rendered as a planar tessellated mesh using precomputed tri-
angle strips. The elevation is assigned by a vertex shader, which uses the
texture coordinates of each vertex to fetch the appropriate height value from
the cached texture. Cracks are avoided by using precomputed triangle strips
that stitch the neighbor nodes together. These meshes are cached in GPU
memory using Vertex Buffer Objects.

Height maps, texture maps and meshes are cached in GPU memory. In this
way, they can be reused to render the panoramas of different clients without
incurring in extra CPU-GPU data transfers. Moreover, since panoramas are
used as impostors to portray distant terrain, there is no need to render the
terrain with a high geometric and texture quality. Let us justify this claim.

Clearly, having a terrain resolution such that two or more terrain points are
projected onto the same pixel does not improve the quality of the image
rendered. In what follows refer to Figure 5. Assume that \( h \) is the resolution
of the height map measured in absolute world coordinates, and let \( p \) denote
the distance between two adjacent height values projected onto the screen-
space coordinates measured in pixels. If \( r \) is the resolution of the screen
in pixels and \( w \) is the width of the viewing frustum at distance \( d \) from the
viewer, and according to the Thales first theorem, it is easy to see that

\[
\frac{h}{p} = \frac{w}{r}
\]

Now, if \( \theta \) denotes the field of view, we have

\[
\tan \frac{\theta}{2} = \frac{w/2}{d}
\]

Combining the previous equations, we have, [5],

\[
h = \frac{2}{r} \cdot \tan \frac{\theta}{2} \cdot p \cdot d
\]

According to our definition of panorama given in Section 3.2, terrain points
in the panorama fulfill \( d \geq z_{\text{front}_s} \) and \( \theta = 90^\circ \). If we set the screen space
distance between adjacent height values \( p \) to 1 pixel, we get

\[
h \geq \frac{2}{r} \cdot z_{\text{front}_s}
\]

To illustrate what this relationship entails, assume that the front clipping
plane of the server is placed at \( z_{\text{front}_s} = 7500 \)m from the viewer and that
the panorama resolution is \( r = 512 \) pixels. In these conditions, there is no
need for storing a height map with a resolution finer than 29.29 meters.

Nowadays most modern graphics cards feature at least 512 MB of video
RAM. Thus, a 80 meters resolution height map of a country of the size

12
of Spain (500 000 Km$^2$) and its associated texture map can be completely cached in video RAM. According to Equation 1, and assuming the same conditions as before, a height map resolution of 80 meters causes a maximum screen-space distance between adjacent height values of $p = 2.73$ pixels, which is a good enough approximation considering the fact that the panorama is used to simulate terrain in the background.

6.2 The Panorama Encoder

The Panorama Encoder module is fed with the raw panoramas generated by the Panorama Renderer, encodes them in a compressed format suitable for both network transmission and fast decoding by the mobile clients and sends the compressed images to the Panorama Server Module.

We have implemented the Panorama Encoder on top of an ad hoc version of the open-source Libjpeg Library, [9]. The Libjpeg library has been slightly modified to allow the output compressed images to be stored in memory buffers instead of files. The degree of JPEG compression can be adjusted to improve the tradeoff between bandwidth requirements and panorama quality.

The Panorama Encoder has been designed as an independent module and is run on a specific thread. In this way, the Panorama Encoder and the Panorama Renderer can work in parallel, thus obtaining a pipeline effect.

From a computational point of view, the JPEG encoding stage represents the most expensive task of the panorama generation process. Since most current CPUs feature multiple processing cores, we can make better use of the available computational resources by launching multiple instances of the Panorama Encoder running in parallel. This reduces the encoding time by processing multiple images concurrently. This parallel scheme is specially

Figure 5: Relation between world-space and screen-space coordinates.
valuable considering that the server will likely receive multiple panorama requests at the same time, each panorama consisting of several independent images. Each image can be concurrently encoded by an independent instance of the Panorama Encoder, thus obtaining a better overall performance.

The number of concurrent Panorama Encoder instances is a configurable parameter of the application. In our implementation, we trigger a Panorama Encoder on each processing core available in the server.

7 Client-Side Architecture

In our architecture, clients should run a dedicated application. However, since mobile devices suffer from severe memory constraints and lack of computing power, this application must be lightweight and fast. Several cross-platforms languages designed for mobile devices have been developed. For example, JME, [27], offers a language that combined with JSR-184, [10], usually know as M3G, allows fast development of 3D applications on mobile devices. Unfortunately, after conducting some preliminary tests to assess mobile platforms performances, we discarded this technology due to the low performance of the resulting applications.

The client-side application was developed having in mind two goals: portability to different platforms, [24], and optimizing the use of the limited available resources. Since the C++ programming language is supported by most platforms, as software tools we selected C++ with OpenGL ES, [12].

The application was structured as a lightweight multilayer software architecture illustrated in Figure 6. This architecture provides an intermediate layer that decouples the application from the specific platform. The Common 3D API provides a high level, object oriented common graphics library. Currently, it is built upon OpenGL and OpenGL ES. The Common System API provides a common interface for platform-dependent tasks such as window creation, user interface or input/output. Both APIs are a slim wrapper over the underlying native API and do not grows the overhead of the application. Currently, our client application works under iPhone OS (iPhone, iTouch and iPad), Symbian OS, Win32 and GNU/Linux 32/64 bits. Figure 7 shows an screenshot of our system running a multiuser session involving a laptop PC, a Nokia N95 and an iPhone 3GS connected to the same server.

The client-side application has also been designed as a multithreaded application, as depicted in Figure 3. A main thread manages the user interface, renders the scene, updates the Local Database and processes the quadtree nodes and panoramas provided by the server. To reduce the CPU load in the main thread, networking tasks are moved to a second thread which manages the communication with the server and that, in parallel, decodes JPEG
Figure 6: Software abstraction layer.

Figure 7: A multiuser session involving a laptop PC, a Nokia N95 and an iPhone 3GS connected to the same server. Red dots show in real time the location of the clients.
7.1 The Local Database

The client Local Database, unlike its server’s counterpart, resides in main memory of the client and it is optimized for fast rendering and updating. It serves as a temporal repository where those components of the scene needed for rendering are stored. The Local Database maintains a very small subset of the complete terrain dataset, consisting on a small grid of incomplete quadtrees centered on the viewer and the panorama currently being displayed.

7.2 The Visualization Module

The information stored in the Local Database is used by the Visualization Module to render the scene according to the current viewer position. The final image to be shown to the user is built by merging the close-range geometry and the background terrain portrayed in the panorama. The Visualization Module also manages with user interaction. Currently, it handles user events generated by touch screens, keyboards and accelerometers. This module also handles the GPS geolocation.

7.3 The Database Updater Module

The Database Updater Module of the client takes care of updating the Local Database dynamically, according to the current needs of the application. This module is in charge of three tasks. First, it adds to the Local Database the information incoming from the server through the network thread. Second, this module constructs a triangulated mesh which is a good approximation of the terrain for the terrain in the current view. It determines whether new terrain data should be downloaded from the server, issuing a request if needed. Also, it discards those parts of the terrain dataset that are no longer needed. Finally, the Database Updater requests a new panorama whenever it must be updated. The request is sent to a temporal queue, where they await until the network thread transmits it to the server.

7.4 The Data Decoder and the Requests Generator

These two modules run in a separate thread, and manage the communication with the server. The Data Decoder module recieves network packets from the server and it is responsible for unpacking, decoding and passing them to
the DataBase Updater module. The decoding is performed in a secondary thread thus it does not have an impact on the rendering performance.

The Requests Generator receives requests from the DataBase Updater module and stores them in an output buffer. Such requests are then packed and written in the network socket.

8 Real-time 3D Locating System

In this paper, we have described a complete client-server architecture which allow multiple mobile devices connected to a server to perform an interactive navigation over large terrains. The navigation can be based on simulated coordinates or in actual geographic coordinates provided by a positioning system. Navigation using simulated coordinates is useful in a wide number of fields including computer games, interactive guides and virtual tourism. However, combining the technology proposed in this paper with a positioning system allows for a larger variety of applications.

Our architecture has been specifically designed to be easily combined with positioning systems in a natural way. Those clients equipped with a positioning system, i.e. a GPS receiver, can use the proposed architecture as a real-time 3D positioning system. The viewer position in the virtual environment is determined by the physical position of the client in the real world provided by the positioning system. As the client device physically moves, its position in the virtual environment is consequently updated and new terrain or panoramas are downloaded from the server if needed. This way, a realistic 3D representation of the environment which surrounds the user can be portrayed in the screen of a mobile device. This representation can be enhanced with vector information concerning road maps and points of interest (cities, name of the mountains in the line of sight, etc.).

The multi-client capabilities of our architecture can be exploited in order to create a real-time locating system. There is an essential difference between a locating system and a positioning system. Positioning systems inform the users of their location, i.e., the GPS receivers widely used in vehicles to inform drivers of their location and direction. However, locating systems provide the user with the locations of tracked objects. A real-time locating system can be used by an organization to monitor the movements of vehicles, persons or any kind of resources in order to more effectively optimize their resources. Although the idea of real-time locating systems is not new, no attempts have been made to combine this technology with a 3D realistic representation of the terrain on mobile devices.

In our architecture, each client stores a list with the last position of every client connected to the server. The viewer position is updated in real time
according to the geographical coordinates provided by the GPS receiver. Periodically, the client provides the server with its current coordinates. The Server Instance connected to the client, see Section 5, sends the client coordinates to every running Server Instance that broadcast them to all the connected clients. Consequently, each client updates its list with the new data.

In our implementation, connected clients are portrayed in the screen as labeled icons according placed at the client current position, as illustrated in Figure 7. Whenever a client moves, its icon moves accordingly, allowing to the connected users to track the real-time positioning of other users. Icons describe paths linearly interpolated between the old and the new positions.

It is also possible to define a policy to figure out how the positioning information should be distributed. The policy is based on defining two categories of clients, namely, controllers and standard. Controllers are provided in real time with the location of the connected clients. The standard clients are responsible of providing their geographical location to the server, but they are not aware of the position of any other client. This policy can be used in applications such as fleet management, vehicle tracking, emergency services, or personal tracker for child safety and elderly assistance.

This policy can be extended to easily track the location of objects in real time using as clients simple, inexpensive devices that connect to the server like any standard client. However, they do not perform any terrain rendering but only provide the server their location. These clients are called thin clients. A functional thin client only requires an Internet connection and a positioning system, such as a GPS receiver, and can be deployed when we need to track the location of mobile agents, such as vehicles or persons, which do not require of a 3D terrain visualization. Note that, from the server point of view, there is no difference between a standard client and a thin client. However, as stated in Section 5, the server only provides geometry and panoramas on demand. As thin clients never issue any request to the server, associated Server Instances are permanently sleeping without incurring in any extra operation other than receiving and processing positioning information.

The combination of standard clients and thin clients facilitates to apply our approach to a broad variety of applications. For example, we have applied our approach to develop a prototype for real-time tracking for sport events. Each racer carries a thin client which periodically provides its geographical coordinates to the server via Internet. The server provides the users with the exact location of each racer in real-time. Further vector information can also be provided, including the stage route and the weather conditions. Figure 8 shows the prototype running on an Apple iPhone to track the progress of racers in a bicycle race. The prototype is specially amenable for
open environments where visualizing the 3D environment of the stage is of special interest and can take advantage of large terrain rendering and of the streaming nature of the application.

9 Results

For a comparison between performances of our hybrid rendering approach and other approaches, as well as for a detailed study of the performance on the client side, we refer the reader to the work reported in [19]. Here, we focus on the server side.

We assess our architecture performance on the server side on the basis of three main requirements that any client-server architecture which aims at providing a real time 3D navigation on mobile devices should address: server performance, scalability and network performance.

As stated in Section 1, the proposed architecture does not require of any expensive hardware on the server side. Our server implementation was tested on an ordinary desktop PC with 2.40GHz Intel Core-2 Duo CPU, 4 GB system memory, a nVidia GeForce 8800 GTS GPU and a commodity S-ATA hard disk. In our tests, both the Main Server and the Panorama Server components were run in the same computer.

We used in our experiments a terrain made of 10240 × 10240 samples with a horizontal resolution of 10m and a vertical resolution of 0.1m. The texture map included 10240 × 10240 pixels, with a resolution of 10m per pixel.
Table 1: Experimental values for different panorama resolutions and JPEG encoding quality.

<table>
<thead>
<tr>
<th>Panorama Resolution</th>
<th>JPEG Quality</th>
<th>Render Time (s)</th>
<th>Read Time (s)</th>
<th>Coding Time (s)</th>
<th>Panorama Size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256^2</td>
<td>60</td>
<td>0.00084</td>
<td>0.00460</td>
<td>0.00996</td>
<td>10.64</td>
</tr>
<tr>
<td>256^2</td>
<td>80</td>
<td>0.00072</td>
<td>0.00464</td>
<td>0.01076</td>
<td>12.48</td>
</tr>
<tr>
<td>512^2</td>
<td>60</td>
<td>0.00088</td>
<td>0.00864</td>
<td>0.02932</td>
<td>30.40</td>
</tr>
<tr>
<td>512^2</td>
<td>80</td>
<td>0.00104</td>
<td>0.00872</td>
<td>0.02972</td>
<td>35.80</td>
</tr>
<tr>
<td>1024^2</td>
<td>60</td>
<td>0.00184</td>
<td>0.02344</td>
<td>0.11484</td>
<td>101.92</td>
</tr>
<tr>
<td>1024^2</td>
<td>80</td>
<td>0.00156</td>
<td>0.02328</td>
<td>0.11528</td>
<td>118.76</td>
</tr>
</tbody>
</table>

9.1 Server Performance

To measure the performance of the Panorama Server component of our framework, we connected one client to the server and performed a fly-over at a constant height of 100m over the terrain. We generated 100 panoramas with different viewer positions. Terrain boundaries were never reached. Table 1 shows the average performance values yielded by the Panorama Server. From left to right, Table 1 lists the resolution of the panorama defined as the resolution of the panorama skybox faces, the JPEG compression quality required, the time needed to perform the rendering of all the images in one panorama, the time required to copy the images resulting from rendering the panorama from GPU memory to main memory, and the time required to perform the JPEG encoding for one panorama and the compressed panorama size. The average number of triangles rendered by the server for each panorama was 336752.

The analysis of values in Table 1 allow us to identify strengths of the Panorama Server as well as potential bottlenecks. As expected, computing times in all stages of the panorama generation process increases with the panorama resolution. However, the JPEG compression quality required does not seem to have an effect on the processing time but causes an increase in the panorama size.

Coding Time is always significantly longer than Render Time and Read Time. This suggests that the encoding phase is the most expensive task when rendering panoramas. Differences become larger as the panorama resolution increases. The render phase is clearly takes the smallest computational load. These results favor the strategy of providing our architecture with multiple Panorama Encoders running in parallel.

In our tests, Coding Time ranges between 0.00996s for the 256^2 resolution and 60 JPEG quality scenario, and 0.11528s, for 1024^2 resolution and 80
JPEG quality scenario. The first scenario would allow an encoding rate of 100.40 panoramas per second, while the second would allow an encoding rate of 8.67 panoramas per second. These figures give us an approximate upper limit for the number of clients per second that can be provided with panoramas if the system includes just one instance of the Panorama Encoder.

Further improvements of the panorama encoding phase would require to increase the computational power of the machine hosting the Panorama Server by adding more processing cores or to use a specific hardware-based JPEG encoding implementation. Currently, we use the Libjpeg Library, which is a pure software-based JPEG encoder.

9.2 Scalability

To assess the scalability of our architecture, we carried out a set of experiments with an increasing number of connected clients. For each test, clients simultaneously established a connection to the server and performed a rectilinear flyover at a constant speed and a constant height of 100m over the terrain. The starting point and the flight direction of each client were random values. The navigation speed was also a random value in the range 100 to 150km/h. To avoid fake results, the terrain boundaries were never reached by any client. In all tests, the minimum viewing distance was 30Km. The panoramas were placed 7.5Km away from the client. We used the panorama updating criteria reported in [19] with a maximum allowed error of 5%. Each test lasted 300 seconds. The average number of triangles rendered by the server in rendering one panorama was 317835.

Mobile clients were simulated by running several instances of the client application on a second PC. Each client was locally rendering around 10000 triangles. Note that, from the server point of view, there is no practical difference between an actual and a simulated mobile client.

For each test, we recorded a set of measures in the server. Table 2 summarizes the averaged results yielded by the experiments. From left to right Table 2 lists the number of panorama requests received by the server per second, the time required by the Panorama Server to actually generate a panorama, the number of quadtree node requests received by the server per second, and the time required by the Server Instance to get ready the response to a quadtree node request. This last measure includes extracting the data from the GIS database resident in a local hard disk. Each response contains four brother quadtree nodes, each consisting on a $9 \times 9$ data points height map and a $64 \times 64$ pixels texture.

Our approach manages to achieve an almost constant processing time for panorama generation, regardless of the number of concurrent clients. The
Table 2: Server performance for increasing number of clients.

<table>
<thead>
<tr>
<th>Clients</th>
<th>Resolution</th>
<th>Panorama/s</th>
<th>Panorama Time (s)</th>
<th>Nodes/s</th>
<th>Node Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256²</td>
<td>0.07</td>
<td>0.01962</td>
<td>0.37</td>
<td>0.00162</td>
</tr>
<tr>
<td>1</td>
<td>512²</td>
<td>0.07</td>
<td>0.04420</td>
<td>0.37</td>
<td>0.00145</td>
</tr>
<tr>
<td>4</td>
<td>256²</td>
<td>0.29</td>
<td>0.01835</td>
<td>1.41</td>
<td>0.00194</td>
</tr>
<tr>
<td>4</td>
<td>512²</td>
<td>0.30</td>
<td>0.04141</td>
<td>1.41</td>
<td>0.00235</td>
</tr>
<tr>
<td>8</td>
<td>256²</td>
<td>0.55</td>
<td>0.01924</td>
<td>2.73</td>
<td>0.00280</td>
</tr>
<tr>
<td>8</td>
<td>512²</td>
<td>0.56</td>
<td>0.04367</td>
<td>2.74</td>
<td>0.00284</td>
</tr>
<tr>
<td>16</td>
<td>256²</td>
<td>1.28</td>
<td>0.01799</td>
<td>5.85</td>
<td>0.00335</td>
</tr>
<tr>
<td>16</td>
<td>512²</td>
<td>1.25</td>
<td>0.04193</td>
<td>5.78</td>
<td>0.00457</td>
</tr>
</tbody>
</table>

The low rate of panorama updates stems from the fact that clients only need to update the panorama they are displaying when they move over a certain threshold. At automobile-like speeds, a client only issues a panorama requests at intervals of a few seconds. Therefore, splitting the rendering workload between the clients and the server causes a noticeable reduction in the server workload when compared to classic server-based rendering approaches. Results in Table 2 show that the most demanding scenario corresponds to a panorama resolution of 512² and 16 connected clients. Here, the averaged number of panorama requests 1.25 per second. Each panorama requires 0.04193 seconds to be completed, that is, the Panorama Server spends only a 5.24% of its time processing panoramas. Thus the 94.76% of the time, it is idle waiting for new incoming requests. Therefore, we expect that there is room to significantly increase the number of concurrent connected clients and the navigation speed without incurring in any performance issue.

The same rational applies to geometry requests. The most demanding scenario takes place with 16 concurrent connected clients each one locally rendering 10000 triangles and moving at automobile-like speeds. The server receives less than 6 quadtree node requests per second. Since the server only requires some milliseconds to process each request, our server can clearly attend 16 clients simultaneously or even a higher number of concurrent clients. Finally, it is worth to note that the computer used in our experiments features a conventional S-ATA disk. However the modular design of our architecture allows the GIS database to be located in another host, preferably a dedicated database server featuring faster hard disks with wider caches.

### 9.3 Network performance

To complete the evaluation of our system behavior, we carried out several experiments to measure the user response time using real-world wireless
connections. The user response time was defined as the time elapsed between the client request and the arrival of the complete response and measured on the client clock.

To study the influence of the network link, we used three popular mobile telecommunications technologies: General Packet Radio Service (GPRS), Universal Mobile Telecommunication System (UMTS) and IEEE 802.11g Wireless Local Area Network (WLAN). We did not account for the effects resulting from the viewer roaming back and forth in neighboring cells.

The client was an Apple iPhone 3GS smartphone. For each test, we performed a flyover using the conditions defined in Section 9.2. To facilitate comparisons, the starting point, direction and navigation speed were common to all the tests. The panorama encoding quality was set at 80. The iPhone locally rendered about 10000 triangles per frame. Averaged response times for 300s flyovers are summarized in Table 3. The first column shows the network used, the second the time elapsed between the client request and the arrival of the panorama, the third the time elapsed between the client request of a quadtree node and the response is completed.

User response times largely depend on the round trip time (RTT) of the communication channel, especially when transmitting small messages, such as the quadtree node responses. Unfortunately, RTTs in both GPRS and UMTS are rather large [7, 8].

<table>
<thead>
<tr>
<th>Network</th>
<th>Panorama Resolution</th>
<th>Panorama Response (s)</th>
<th>Quadtree Node Response (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPRS</td>
<td>256²</td>
<td>1.379</td>
<td>1.615</td>
</tr>
<tr>
<td>GPRS</td>
<td>512²</td>
<td>1.674</td>
<td>2.019</td>
</tr>
<tr>
<td>UMTS</td>
<td>256²</td>
<td>0.253</td>
<td>0.212</td>
</tr>
<tr>
<td>UMTS</td>
<td>512²</td>
<td>0.275</td>
<td>0.207</td>
</tr>
<tr>
<td>802.11g</td>
<td>256²</td>
<td>0.139</td>
<td>0.014</td>
</tr>
<tr>
<td>802.11g</td>
<td>512²</td>
<td>0.171</td>
<td>0.013</td>
</tr>
</tbody>
</table>

Table 3: Effect of the panorama resolution and the network channel on the user response times.
As expected, user response times for the WLAN connection are significantly smaller than for the UMTS or GPRS connections. This can be attributed to the direct communication between the client and the server offered by WLAN. In the WLAN scenario, RTT is always below 0.015s, with average response times ranging from 0.14 to 0.17s for panoramas. The average time needed to provide a quadtree node is 0.014s. However, in the UMTS scenario, all user response times were slightly higher than 0.2s, which is a consequence of the RTTs of about 0.2s that we experienced when using this network.

Considering the low rate of requests issued by the client, see Table 2, the user response times experienced in WLAN and UMTS connections clearly allow for a high interactive and smooth navigation. The low network requirements of our architecture does not result in any network congestion under these network scenarios.

Concerning GPRS, our experiments showed that most RTT values exceeded one second. The mean user response times fall within the range between 1.3s and 2s, see Table 3. Consequently, our application is subject to noticeable delays under demanding situations, like the initial loading of the scene or when the user moves at very high speeds. Nonetheless, even under these unfavorable conditions, the architecture still provides a stable number of triangles and a good quality when moving at car speeds. Also, contrarily to pure server-based approaches, our technique achieves full interactivity even when using the GPRS slow connection. Quadtree nodes transmission is considerably delay-tolerant because the Local Data Base partially hides high network latencies. Since the database can be progressively updated as the viewer moves, a coarse version of the terrain nearby the viewer can be rendered without waiting for all the data to be totally streamed from the server.

10 Summary and Future Work

Due to the limited computing resources and restricted bandwidth available in current mobile devices technologies, designing systems for adaptive streaming and rendering of large terrains over wireless networks for mobile devices is a challenging task. In this paper, we have described a complete and scalable client-server architecture which successfully overcomes these limitations. The architecture is based on a hybrid rendering technique which dynamically splits the rendering workload between a remote server and the mobile clients.

To assess scalability and performance robustness, we carried out an exhaustive analysis of the server workload and response time with respect to
different network scenarios and number of simultaneously connected clients. Contrarily to most server-based rendering approaches found in the literature, our results show that a commodity PC is capable of providing a smooth navigation to a large number of concurrent clients.

Future work will include enhancing our network protocol in order to further improve the performance under high latency networks. We also plan to study the use of a prediction mechanism to enable loading tiles and panoramas in advance, based on the path followed by the viewer. This would result in a smoother streaming of data from server to client. Finally, we plan to study the roaming effects when the user moves back and forth along neighboring cells and the connection problems that might appear.

Acknowledgements

This work has been partially granted by the Junta de Andalucía, the Spanish Ministry of Education and Science, and the European Union ERDF funds under research projects P06-TIC-01403, P07-TIC-02773, TIN2007-67474-C03-01 and TIN2007-67474-C03-02.

References


