Interactive Three-Dimensional Rendering on Mobile Computer Devices

Javier Lluch[*], Rafael Gaitán[†], Emilio Camahort[‡], Roberto Vivó[§]
Computer Graphics Section
Department of Computer Science
Polytechnic University of Valencia
Camino de Vera s/n 46022, Valencia, Spain

ABSTRACT
We present a client/server system that is able to display 3D scenes on handheld devices. This kind of devices have important restrictions of memory and computing power. Therefore, we need to limit the amount of geometry sent by the server to each client. We extract the geometry that is visible for each client and send it. The clients render the geometry using the *OpenGL ES* [11] API. Our geometry extraction algorithm employs multiresolution and view-dependent simplification. We present results of our system running on a software implementation of *OpenGL ES* that runs on a PocketPC 2003.

Categories and Subject Descriptors
I.3.1 [**Computer Graphics**]: Graphics Systems,network graphics; C.2.4 [**Computer-Communication networks**]: Distributed Systems,Client/Server

Keywords
3D graphics, wireless PDAs

1. INTRODUCTION
Over the last few years mobile computing platforms have made important advances. We can see handheld devices with increasing processor speed and integrated wireless technologies. Computer graphics are also rapidly advancing in mobile devices. There is even a 3D graphics library called *OpenGL ES* for embedded systems. Still, mobile devices have limitations of memory and processing power. Also, most of them do not offer hardware accelerated graphics. This makes it difficult to render and interact with large 3D scenes on these devices.

Recent advances in 3D design, acquisition and simulation have led to larger and larger geometric data sets that exceed the size of the main memory and the rendering capabilities of current graphics hardware. Various algorithmic solutions have been proposed to bridge the increasing gap between the power of the hardware and the complexity of the geometry data. These proposals include: rendering with hierarchical multiresolution, viewing frustum culling, occlusion culling and image-based rendering.

Culling methods can be easily combined with multiresolution models to select levels of detail according to criteria like viewpoint location, illumination conditions and the scene's rate of change. This approach is called view-dependent simplification. However, using multiresolution increases the size of the geometry data and requires that the entire geometry be kept in main memory. This is the reason why multiresolution models can only be applied to data sets that do not exceed the size of the main memory.

To solve this problem new models have been developed that store large scenes in secondary memory (*out-of-core*). We can still render these models at interactive speeds using view-dependent simplification. One of the immediate applications of this type of models is rendering of three-dimensional scenes on mobile devices.

We propose applying current multiresolution, viewing frustum culling and out-of-core techniques to 3D graphics rendering on mobile devices. We present a client-server system that delivers simplified geometry to a PDA over a wireless connection. The server stores the scene graph and extracts levels of detail to be rendered at the client. Careful selection of appropriate levels of detail allows rendering at interactive rates on a handheld device.

Our system uses a geometry cache that manages two copies of the set of visible objects. One copy is located at the server and the other at the client. The server updates the cache with the visible geometry when the scene or the viewing parameters change. A synchronization process mantains the coherence between the server and the client copies of the cache. With this approach we only send geometry updates from the server to the client, thus reducing the latency and bandwidth requirements of our system. It also has the added advantage that we can use advanced geometry culling methods at the server side.

Our server system also supports multiresolution and view-dependent simplification, thus allowing the selection of a suitable level of detail for rendering each desired view. The system can render scenes with any number of polygons. It improves on previous systems because we do not store the entire geometry at the mobile device. Neither do we render at the server and send the rendered frames to the client. Instead we make sure that the client only receives

[*]jlluch@dsic.upv.es
[†]rgaitan@dsic.upv.es
[‡]camahort@dsic.upv.es
[§]rvivo@dsic.upv.es

those parts of the scene graph that are visible. Those parts are currently rendered using a software library. Once hardware accelerated rendering is available we will be able to render larger amounts of geometry on the mobile device.

This paper is structured as follows. First we review previous work related to 3D rendering on mobile devices, multiresolution modeling and out-of-core geometry processing. Then we introduce our system and describe its architecture. In the following section we show some results obtained with our system. Finally we present our conclusions and directions for future work.

2. BACKGROUND

In this section we survey previous work in the areas of 3D rendering on mobile devices, multiresolution modeling and out-of-core methods. We also introduce the idea of external memory view-dependent simplification and the *OpenGL ES* API.

2.1 Three-dimensional Rendering on Mobile Devices

There are three techniques for 3D rendering on mobile computer devices: *polygon-based rendering*, *image-based rendering* and *point-based rendering*. We briefly review these techniques.

2.1.1 Polygon-Based Rendering

Polygon-based rendering systems are like those implemented in traditional graphics accelerators. Such systems typically use their own graphics library, although some systems use *PocketGL* [12]. *PocketGL* is a software library for embedded systems. It was developed for specific devices, most of them non-*OpenGL ES* compliant. These systems render three-dimensional scenes with a software rasterizing using the native 2D API for writing to the graphics device.

D'amora and Bernardini presents a 3D viewer for PocketPC devices [2]. The viewer provides a stand-alone solution for access to MCAD 3D models during stages of the manufacturing process where using design workstations is impractical or limited. The viewer accepts models in its own compressed format. The models are sent over a wireless network and decompressed at the receiving end. The problem of this system is that it sends the entire model to the viewing device, instead of sending a simplified version. Therefore it is not very well suited for large scale geometric models.

In [17] the authors describe a *PocketGL*-based continuous multiresolution rendering engine for PDAs. The engine can dynamically adjust the level of detail of the objects according to the target frame rate set by the user. The frame rate is considered as an objective function that determines both the quality of the scene and the interaction capability. The viewer requires that the geometric models reside in the PDA's memory. This limits the amount of geometry that can be handled by the viewer.

Sanna et al. proposes a general architecture for searching, retrieving and rendering complex 3D models on PDA's[14]. The architecture includes geometry servers, rendering servers and client applications that display the resulting images.

2.1.2 Image-Based Rendering

Image-based methods use images to replace some or all of the geometry of a scene in 3D rendering. The use of precomputed images speeds up the rendering algorithm and provides realistic effects at a low cost. Images can also be rendered at a server system and delivered to a client system for remote display.

For example, [1] proposes a client-server system that renders geometry at the server and sends the resulting images to a PocketPC client for display. The images are accompanied by depth information obtained from the renderer's depth buffer. Using the depth information and 3D warping, the client can produced new images without requesting more information from the server.

Alternatively, [15] introduces a system that delivers images to a remote PC. The system uses a client-server collaborative scheme to determine which pixels need to be updated for every new frame. Instead of sending entire images, the authors use a prediction method to exploit spatial coherence and wipe out correct pixels from the difference image. This substantially reduces the bandwidth requirements of the client-server communication.

Finally, [16] describes a server system that sends a compressed video stream to a client. The stream has been rendered using the camera parameters provided by the client. The software implementation of this system allows engineers to view and interact with 3D CAD files using a wireless phone connected to a personal data assistant (PDA).

2.1.3 Point-Based Rendering

For complex scenes, the traditional graphics pipeline can be wasteful, with the processor spending much effort transforming and rasterizing numerous geometric primitives that might cover less than a pixel. Point-based rendering displays complex geometries by rendering a set of points located on the surface of the objects. The number of point samples rendered depends on the complex object's screen size.

One approach to point-based rendering is to create unstructured point sets by generating point samples stochastically as a preprocess or procedurally on the fly. Another approach is to generate structured point sets, for example by creating a hierarchy. This method also permits visualization of complex models, in particular those that do not fit in main memory. The algorithm creates a hierarchy of bounding volumes and achieves flexible rendering by halting the descent into the hierarchy depending on rendering speed requirements and screen size. This hierarchy stores intermediate normal and color attributes.

In [4] the authors use a packed hierarchical point-based representation for rendering. The method supports traversal of the hierarchy in a specific order, resulting in a fast, one-pass shadow-mapping algorithm.

2.2 Multiresolution Mesh Representation

Multiresolution meshes are a common basis for building representations of geometric shapes at different levels of detail. The use of the term multiresolution means that the accuracy (or level of detail) of a mesh in approximating a shape is related to the mesh resolution, i.e., to the density (size and number) of its cells. A multiresolution mesh provides several alternative mesh-based approximations of a spatial object (e.g., a surface describing the boundary of a solid object, or the graph of a scalar field).

A multiresolution mesh is a collection of mesh fragments, describing usually small portions of a spatial object with different accuracies. It also includes suitable relations that allow selecting a subset of fragments (according to user-defined accuracy criteria), and combining them into a mesh covering part or the whole object. Existing multiresolution models differ in the type of mesh fragments they consider and in the way they define relations among such fragments. The reader is referred to the survey [7] for a detailed description of multiresolution mesh representations.

2.3 External Memory View-Dependent Simplification

Recently, the idea of view-dependent simplification [6] was introduced. View-dependent simplification enables changes to multiresolution hierarchies that depend on parameters such as viewer location, illumination conditions and speed of motion. At each frame these simplifications adapt the mesh structure to obtain the right level of detail. The main problem is that these schemes increase the size of the datasets, and they require that the entire dataset be kept in main memory.

External-memory (or *out-of-core*) techniques solve this problem for datasets larger than the main memory [5]. The idea is to store the geometric data in disk. Then the data is preprocessed to obtain a set of view-dependent geometry trees that are I/O efficient. These trees are hierachical multirresolution structures. During run-time navigation, this technique keeps all the geometry trees in disk. In main memory it only stores those active trees that are necessary to render the current level of detail, plus some prefetched trees that are likely to be needed in the near future.

2.4 OpenGL ES (OpenGL for Embedded Systems)

OpenGL ES is a low-level, lightweight API for advanced embedded graphics using well-defined subset profiles of *OpenGL*. It provides a low-level applications programming interface (API) between software applications and hardware or software graphics engines.

This standard 3D graphics API for embedded systems makes it easy and affordable to offer a variety of advanced 3D graphics and games across all major mobile and embedded platforms. Since *OpenGL ES* (OpenGL for Embedded Systems) is based on *OpenGL*, no new technologies are needed. This ensures synergy with, and a migration path to, full *OpenGL*, the most widely adopted cross-platform graphics API.

Three-dimensional graphics on mobile devices have made substantial progress since the introduction of *OpenGL ES*. Currently only a few devices contain an integrated chip that supports this technology (see [3] for an example). Manufacturers are now beginning to add hardware graphics to the new generation of handheld devices.

The lack of hardware accelerated graphics in handheld devices has motivated the development of Klimt [9]. Klimt is an open-source software library that implements the *OpenGL ES* interface.

3. OUR APPROACH

In this paper we address the problem of rendering complex three-dimensional scenes on mobile computer devices. The issue is that the scene does not fit in the devices's main memory. So we use out-of-core storage and view-dependent simplification to keep the smallest possible number of polygons stored in the mobile device for rendering. To achieve this goal we employ multiresolution meshes. In this Section we introduce our system's architecture and describe its different components.

3.1 Architecture

Our system's client-server architecture is illustrated in Figure 1. The system is split into three main subsystems: (i) the server system capable of loading a 3D graphics scene and extracting the geometric data to be rendered at the client, (ii) the client application that connects to the server and receives the data to render the scene. and (iii) the communication layer that implements the communication protocols and an object/triangle cache to improve transfer rates. Figure 2 shows the internal architecture of the server system and the client applications.

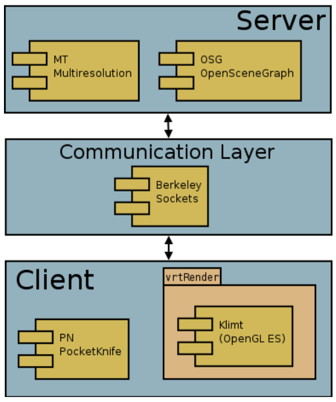

Figure 1: System architecture

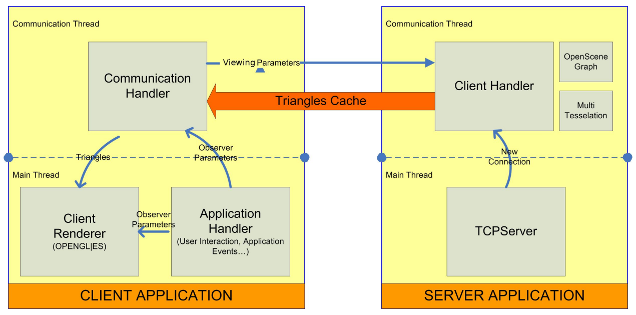

Figure 2: Internal architecture

3.2 Server System

The server loads a scene graph with *OpenSceneGraph* and begins listening to connections. We use *OpenSceneGraph* (OSG) [13] to store and manage complex 3D models. OSG is an open source high performance 3D graphics toolkit, used by application developers for visual simulation, games, modeling, virtual reality and scientific visualization.

Once a client establishes a connection, the server creates a new thread to manage the connection. Then the client sends the viewing parameters to the server, and the server configures the camera inside OSG. After that, the server calls the cull process.

OSG implements culling methods that substantially reduce the geometry sent to the render stage. However, it does not support multiresolution meshes. So we merged the *multitesselation library* (MT) [8] with OSG. That way we can manage large 3D scenes with multiresolution. The MT library supports focusing on a given volume of the scene for detailed extraction of the relevant geometry. Depending on the type of scene the volume can be a pyramid, a 3D box or another object. Figure 3 shows an example with a 3D box.

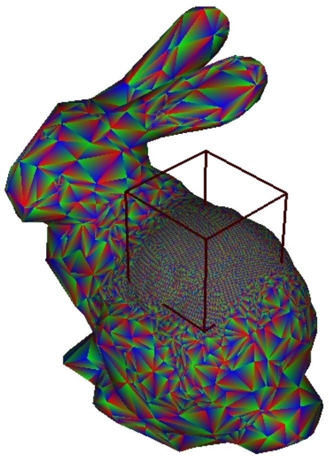

Figure 3: Multiresolution bunny rendered with *OpenSceneGraph*. Note that the focus of the MT library, a 3D box, contains a higher resolution representation of the surface of the bunny.

As for culling, we have created a new class *ExtractVisitor* that extends the standard class *CullVisitor* of OSG. A *Visitor* calls the culling process and extracts the geometry to be rendered at the client. For this purpose the *Visitor* uses the viewing parameters and the geometry stored in the scene graph. We support all geometric primitives supported by OSG. For *multiresolution meshes* we do extra work to extract the right level of detail. We create the box that contains the viewing frustrum. Then we use this box to extract the appropriate resolution using the MT library. The geometry thus obtained is sent to the client using the system's communication layer.

3.3 Client Application

We create a client application capable of establishing connections to the server using integrated wireless technology. The client application can also run on laptops and desktop computers with wireless connectivity. Once the client establishes a connection, it creates a communication thread. The communication thread sends the viewing parameters to the server and receives the geometry from the server. The main thread of the client renders the scene and handles the user's interactions.

Our client application runs on both Win32 and PocketPC platforms. Running the client on Win32 systems allows using a remote desktop or laptop to view the scene. To make this possible we use the *Pocketknife* library. *PocketKnife* is an application framework that supports multi-platform development for both x86 Windows and Windows CE. This means that we can develop on a desktop computer and compile for the PDA just for final tests and device-dependent debugging.

The problem of rendering 3D graphics on mobile devices is greatly simplified by *OpenGL ES*. Still, only a few devices implement *OpenGL ES* in their graphics hardware. So we use a software solution: the Klimt library. This multi-platform library is a software implementation of *OpenGL ES* that runs on both mobile devices and desktop computers. On top of Klimt (or *OpenGL ES*) we developed an object-oriented library, *vrtRender*, that provides an abstraction for meshes, materials, light sources and textures. The client application uses this rendering library that offers a higher-level interface to *OpenGL ES*.

3.4 Communication Layer - The Scene Cache

The communication between the client and the server is handled by a library we developed on top of standard Berkeley sockets. The library implements data structures for client-server communication. Two data structures are usually exchanged between client and server. The first one encapsulates the viewing parameters sent by the client to the server. The second one maintains the scene cache.

There are two copies of the scene cache. One is kept at the client and the other at the server. The communication layer keeps both copies synchronized at all times. During scene navigation the cache is updated by the server for each new set of viewing parameters. Then the cache is accessed by the client to retrieve the new geometry to be rendered. The goal of the cache is to exploit the temporal coherence between consecutive frames. For small camera movements we only need to make small changes to the (local) geometry to be rendered.

The scene cache is organized in two levels: the scene cache itself and the geometry cache. The scene cache mantains all the information needed to render the scene, including the viewing parameters and the geometry of the scene. The scene cache is invalidated when the viewing parameters change. In that case, the server extracts the new geometry visible, and passes it on to the communication layer. Then the layer synchronizes the scene cache with the copy at the client.

The geometry cache stores the polygon, color and normal data associated to each geometric object. The geometry cache is updated by the server when the scene cache is invalidated. To improve efficiency the server determines which geometric objects are already stored in the cache. Then it adds to the geometry cache those objects that have become visible. The synchronization process tests, for all cached geometries, a visibility marker generated by the server. If the geometry is visible only a *geometry cache id* is sent to the client. If the geometry is not visible, then the communication layer sends all that geometry information to the client.

The method is extended to support multiresolution meshes. In the extraction process we determine if the multiresolution geometry is changed. The idea is to test the bounding box of the visible MT geometry with the last bounding box extracted. If there are changes then we need to recompute the right level of detail and fill the cache with the new MT geometry, then the syncronization process does the rest for us. The system also computes an *error factor* for the MT geometry, making possible, for example, the extraction of more detail for objects closer to the viewer.

This configuration prevents sending unnecessary mesh information through the network. It also supports navigation through the geometry data stored in main memory at the client, even if the client-server connection fails.

3.4.1 Double Buffer Cache

The main problem of this configuration is the concurrent access of the scene cache by the two threads at the client: the main thread and the synchronization thread. If we lock the cache during synchronization, then the frame rate drops. Instead, we employ a *double*

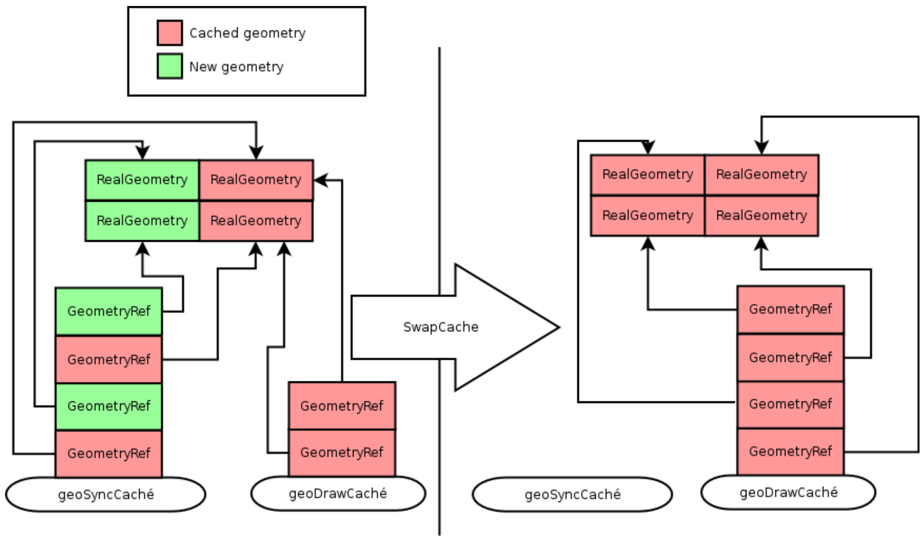

Figure 4: Double Buffer Cache operation. On the left is the state of the cache during the synchronization process. On the right is the state of the cache right after swapping.

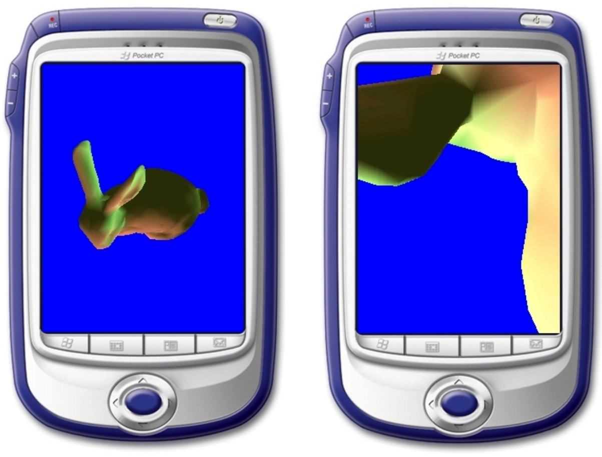

Figure 5: Our client system running on the PocketPC 2003 emulator.

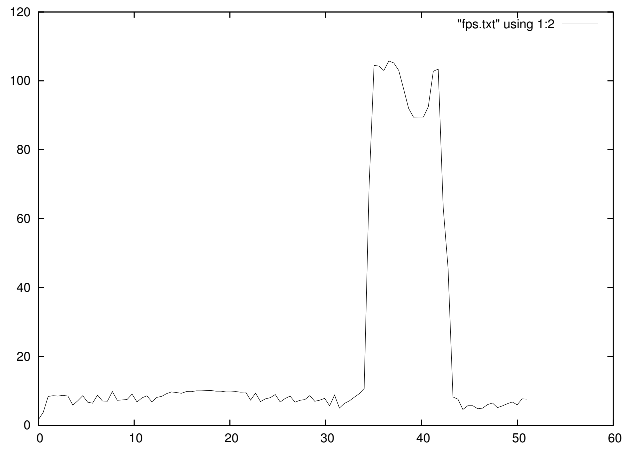

Figure 6: Frame rate of the MT bunny rendered on our test PDA with changes in the geometry and one light. The peak frame rate occurs due to culling when the bunny moves outside of the viewing frustum.

buffer cache. We mantain two buffers, one for the drawing process and the other for the synchronization process. When the synchronization process ends we swap buffers and prepare the system for the next synchronization. That way the main thread can draw from one buffer while the synchronization process updates the geometry in the other buffer.

To avoid using too much memory for this cache, we only keep references to the real geometry. We store the real geometry in a geometry manager. Then the two double buffer caches store pointers to the real geometries handled by the geometry manager. This implies a substantial savings, since the geometry itself is not replicated, only the references. When swapping buffers, only the references need to be updated, thus reducing the locking time of the main drawing thread. Figure 4 shows the operation of the *double buffer cache*.

This configuration runs well for static geometries. For dynamic geometries, like multiresolution meshes, we need to keep replicated information to avoid problems during navigation. For this kind of geometries the client system receives the new geometry information during the synchronization process. That information is stored in auxiliary buffers. When the cache swap is performed, the system sets the new level of detail to draw.

4. RESULTS

To test the implementation of our system we run the server on a desktop PC and the client on different devices. We run the client on a laptop with wireless access, on the PocketPC 2003 emulator and on an HP ipaq 4150 running PocketPC 2003. Figure 5 shows two images of the system running on the emulator.

We used a couple of scenes to test our system. Figure 11 shows some images obtained from running our system with a *multiresolution* mesh made from a 2400-polygon bunny. Figure 12 shows

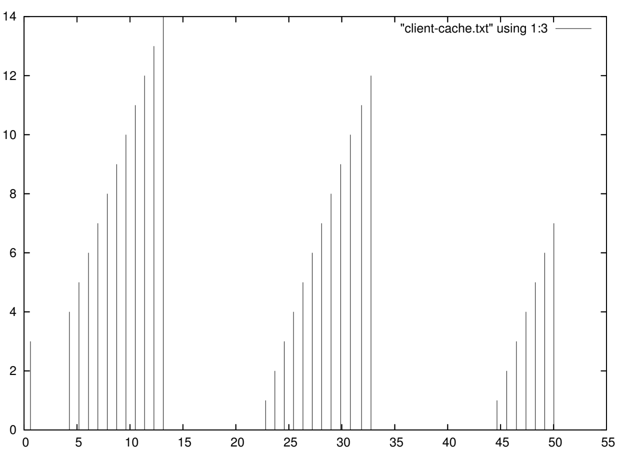

Figure 7: Cache misses of the MT bunny with dynamic geometry.

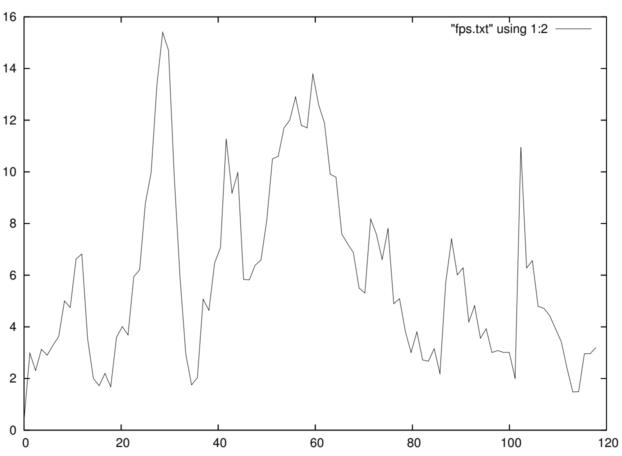

Figure 8: Frame rate of the virtainer scene rendered on our test PDA and no active lights.

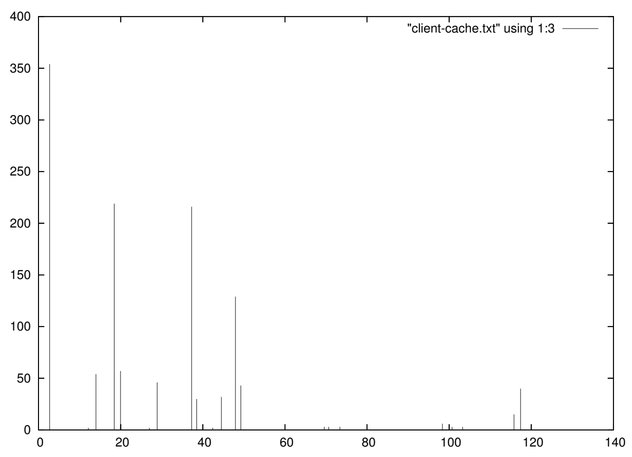

Figure 9: Cache misses of the virtainer scene.

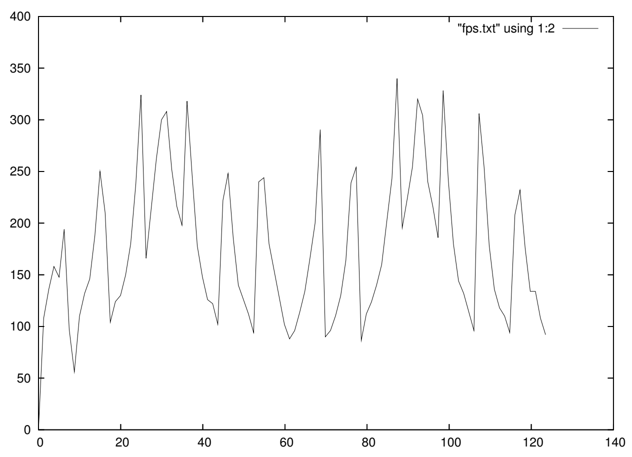

Figure 10: Frame rate of the virtainer scene rendered on desktop PC with accelerated graphics hardware.

the other test scene made of multiple static meshes representing a container terminal. This scene was built using *virtainer* [10], a 3D system to handle and render container terminals in real time.

To analyze our system we made some plots with the evolution of the frame rate (see Figures 6 and 8) and the number of cache misses during navigation (see Figures 7 and 9). The plots were made moving the camera along precomputed paths.

For the MT bunny scene the maximum number of triangles viewed is at most 2400 when the geometry is close to the viewer and less than 1500 when the geometry is far from the viewer. The geometry changes resolution progressively during navigation. We observe that the dynamic geometry mantains the frame-rate at the maximum possible with the Klimt library. When the geometry is not visible we appreciate a large increase in the frame rate.

The *virtainer* scene is more complex, containing many objects and transformations. The precomputed path for the scene moves the camera back and forth along a street with several blocks of containers on both sides.

We observe that the frame rate varies a lot when navigating through the *virtainer* scene. This variable frame rate is caused by the amount of geometry visible from each camera position. Still, the frame rates are not very large due to the limited capabilities of the Klimt software rendering library. Figure 10 shows the frame rate obtained when running the same test on a Win32 client with a high-end graphics card. Note that the evolution of the frame rate is similar, but at a different scale. We conclude that the clients behave in the same way whether we use or not accelerated hardware. This implies that the problem of the PDA-based system is only due to the use of a software renderer.

Table 1 summarizes the information showing the average framerate achieve during the different tests. With the MT bunny and making changes to the geometry, the client frame rate is roughly $8 fps$. With changes in the scene but a large number of static objects the frame rate only drops to $7.7 fps$.

Finally, we have measured the latency of the client-server cache synchronization process. Table 2 shows the average latency achieved during the synchronization process and the maximum and minimum times obtained for the two test scenes.

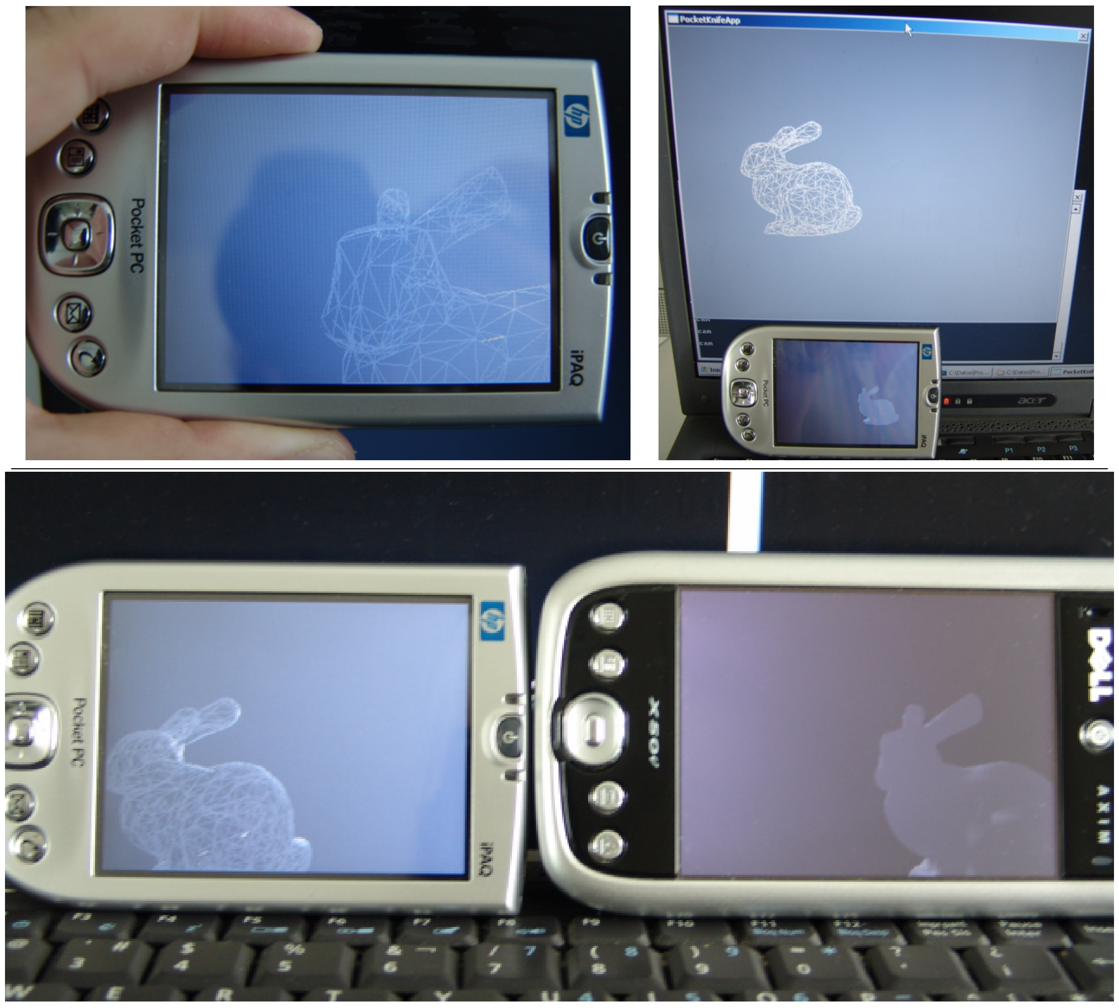

Figure 11: Some images of our system running with two connected clients, two test PDAs and a Desktop PC. Scene with dynamic multiresolution bunny.

	1 Light
Static Bunny	9.63 fps
Dynamic Bunny	8.16 fps
Virtainer Scene	7.76 fps

Table 1: Average fps of MT Bunny and Virtainer scene running on our test PDA. Static Bunny is without changes in geometry during navigation. Dynamic Bunny is with changes in visible geometry during navigation. Virtainer is a scene with multiple static geometries.

	Average	Min Latency	Max Latency
Dynamic Bunny	0.23 s	0.121 s	0.288 s
Virtainer Scene	0.59 s	0.191 s	1.219 s

Table 2: Average, minimum and maximum latency of MT Bunny and Virtainer scene running on our test PDA with wireless connection. The latency is measured from the time when the server cache is filled until client cache is updated.

5. CONCLUSIONS AND FUTURE WORK

In this paper we introduce a system for interactive three-dimensional rendering on mobile devices. We implement a client-server system that delivers geometry to a mobile device and renders the geometry on the device. For rendering we use Klimt, a software implementation of *OpenGL ES* that runs on PocketPC 2003. Our system will thus run on future hardware implementations of *OpenGL ES* for mobile and other devices.

The system's server uses *OpenSceneGraph* and multiresolution meshes for scene management. Given the camera parameters delivered by the client, the server culls the scene graph and sends to the client the geometry to be rendered. The geometry is cached in an object/triangle cache managed by the system's communication layer.

Our system improves on previous systems because we do not store the entire mesh at the client. Instead we only send to the client those polygons that are currently visible. Our system can be applied, for example, to computer gaming and Augmented Reality systems for remote process monitoring, museum guided tours, warehouse management, etc.

We run tests with different scenes and concluded that our system

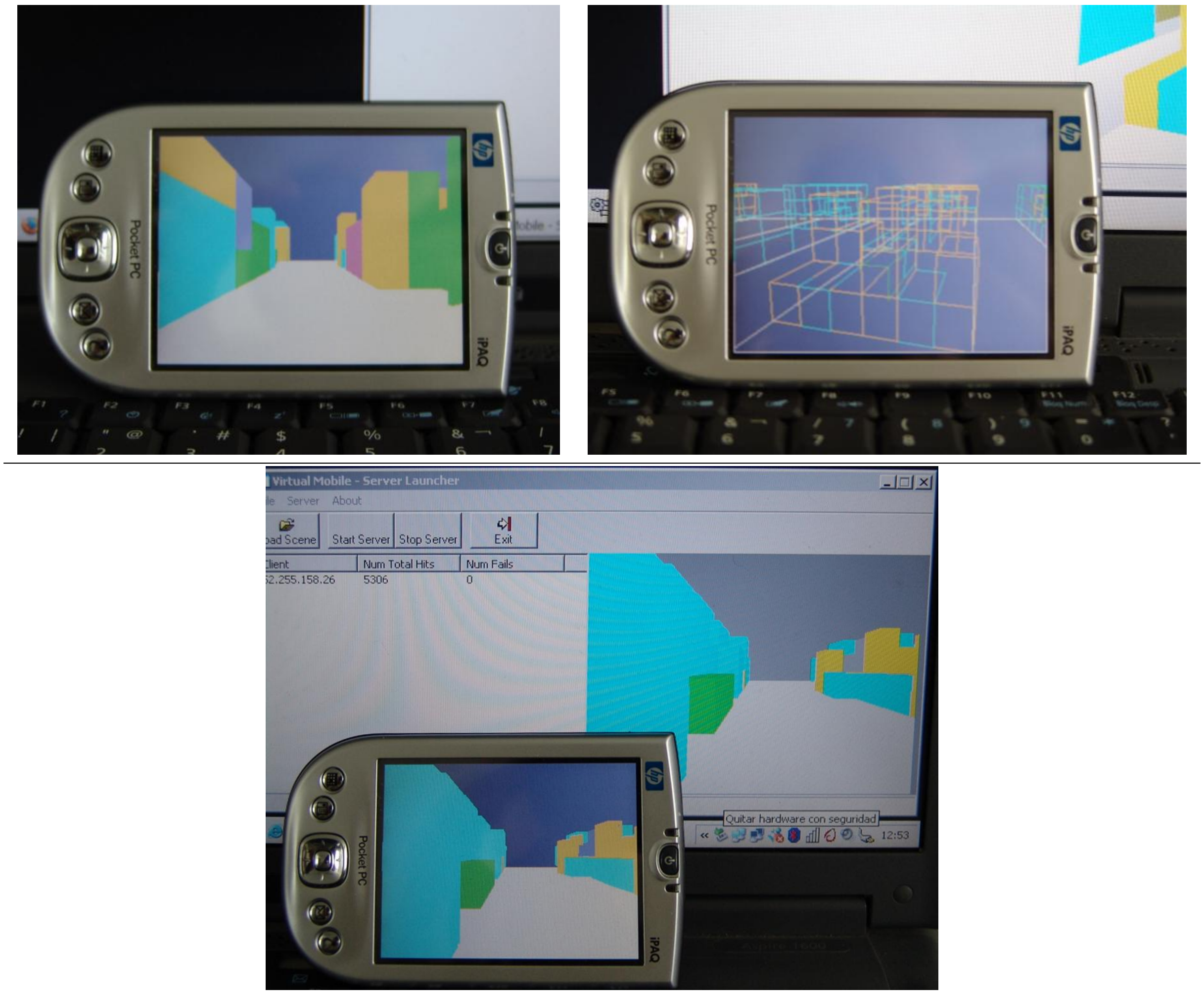

Figure 12: Some images of our system running on our test PDA. Scene of containers terminal (*virtainer*).

scales well as the size of the 3D scene grows. The systems runs very well even with continuos changes in geometry. Our tests were run on both static and dynamic scenes both with camera movement. The advantage of the system is a notable bandwidth reduction when there are no major changes in the viewing parameters.

In the future we would like to improve the triangle cache of our system. The idea is to maintain a cache of multiresolution nodes at the client using external memory view-dependent simplification that supports multiresolution.

6. ACKNOWLEDGEMENTS

This work was supported by grants: TIC2002-0416-C03-01 of the Spanish Ministry of Science and Technology, GV04B-497 2004-2005 of Generalitat Valenciana and IST-004363 of the 6th Framework Program of the European Union.

7. REFERENCES

[1] C.-F. Chang and S.-H. Ger. Enhancing 3d graphics on mobile devices by image-based rendering. In *IEEE Third Pacific-Rim Conference on Multimedia (PCM 2002)*, 2002.

[2] B. D'amora and F. Bernardini. Pervarsive 3d viewing for product data management. *IEEE Computer Graphics and Applications*, pages 14–19, March/April 2003.

[3] Dell Computer Corporation, http://www1.us.dell.com/content/products/productdetails.aspx/axim_x50v. *Axim X50v Details*, 2004.

[4] F. Duguet and G. Drettakis. Flexible point-based rendering on mobile devices. *IEEE Computer Graphics and Applications*, pages 57–63, July/August 2004.

[5] J. El-Sana and Y.-J. Chiang. External memory view-dependent simplification. In *EuroGraphics '2000*, volume 19, 2000.

[6] J. El-Sana and A. Varshney. Generalized view-dependent simplification. In *Proceedings EuroGraphics*, 1999.

[7] L. D. Floriani, P. Magillo, and E. Puppo. Multiresolution representation of shapes based on cell complexes. In *Discrete Geometry for Computer Imagery*, number 1568, pages 3–18. Lecture Notes in Computer Science, 1999.

[8] L. D. Floriani, P. Magillo, and E. Puppo. *The MT (Multi-Tesselation) Package*. Dipartimento di Informatica e Scienze dell'Informazione, Universita' di Genova,

http://www.disi.unige.it/person/MagilloP/MT/index.html, 2004.

[9] IMS Group, http://studierstube.org/klimt/. *Klimt - the Open Source 3D Graphics Library for Mobile Devices*, 2004.

[10] P. Jorquera, J. Lluch, and R. Vivó. Virtainer, “walkthrough” en apilamientos. In *Actas del XIII Congreso Espaol de Informtica Grfica, CEIG’2003*, pages 373–376, 2003.

[11] Khronos Group, http://www.khronos.org/opengles/. *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*, 2004.

[12] P. Leroy. *Pocket GL, 3D library for Pocket PC*. http://pierrel5.free.fr/, 2004.

[13] OSG Community, http://www.openscenegraph.org. *OpenSceneGraph - Open Source high performance 3D graphics toolkit*, 2004.

[14] A. Sanna, C. Zunino, and F. Lamberti. A distributed architecture for searching, retrieving and visualizing complex 3d models on personal digital assistants. *Internet Technology*, 3(4):235–244, 2002.

[15] P.-P. Vázquez and M. Sbert. Bandwidth reduction techniques for remote navigation systems. In *International Conference on Computational Science*, pages 249–257, 2002.

[16] C. Woodward, S. Valli, P. Honkamaa, and M. Hakkarainen. Wireless 3d cad viewing on a pda device. In *Proceedings of the 2nd Asian International Mobile Computing Conference (AMOC 2002)*, 2002.

[17] C. Zunino, F. Lamberti, and A. Sanna. A 3d multiresolution rendering engine for pda devices. In *SCI 2003*, volume 5, pages 538–542, 2003.