

Remote Line Rendering for Mobile Devices

Joachim Diepstraten*, Martin Görke†, Thomas Ertl*

Visualization and Interactive Systems Group, University of Stuttgart
Universitätsstraße 38, D-70569 Stuttgart Germany

Abstract

There is a growing interest for providing interactive graphics also on mobile devices like PDAs, Smartphones, etc. Right now mobile devices are very limited concerning graphical resources basically only simple 2D rasterization operations are available, that run completely on the main CPU of the device. The desire to have more complex 3D graphics on these devices often results in a remote rendering solution. Classical remote rendering solutions produce the final images on a server and transfer these to the client device over a wired or non-wired network. With this paper we present an alternative method to the image-based approach by splitting the rendering process between client and server transferring couple of 2D line primitives over the network, which are rendered locally by the mobile device.

1. Introduction

Mobile devices, e.g. Personal Digital Assistants (PDAs), cellular phones with more capabilities than just phoning (Smartphones) enjoy a significant growing number of users over the past few years. Along with growing of the user base the number of applications that do not belong to the classical tasks of these devices increases as well. Some of them might require interactive graphics solutions either in 2D or 3D. Right now the devices are not powerful enough to render complex 3D content on their own. On the other hand access to wireless network technologies (e.g. WLAN, GPRS, Bluetooth and UMTS) allow for rendering the content on a remote server and only transfer finished images to the mobile client. Unfortunately wireless networks suffer of several problems. The major problem is the low bandwidth compared to wired networks and the high latency due to massive network package loss.

Using image compression techniques can help to reduce the amount of necessary image data transferred through the

network but also increases the work load on both sides. Besides it still requires an expensive fullscreen image blit operation on the client. Increasing power and functionality on mobile devices – some devices already have a second generation 2D accelerator, and first generation 3D accelerators are just around the corner – could require a rethinking of this strategy. It might be wise to re-balance the work load between mobile client and stationary server.

Our work presented in this paper is a first step into this direction. We try to split up some of the image generation tasks between client and server with the aim to reduce the necessary network traffic, exploit increasing – otherwise not used functionality – on the client, and to have a more fair balancing between client and server. The main idea is to just render 2D line primitives on the client, that have previously been generated on the server derived from arbitrary 3D scenes. Using line primitives introduces of course some limitations. For example right now we can not have fully colored and shaded images on the client, instead we have to rely on drawing feature lines of the objects contained in a 3D scene. But according to research from Tufte [19], in many cases a set of feature lines is sufficient enough to engender a good visual impression of the scene to the viewer. Besides, due to the small screen size, resolution and color depth of mobile devices, the benefit of having color right now is actually not that great.

The remainder of this paper is organized as follows. In the next section a brief summary of related work is presented. In the third section the basic concept behind our remote rendering solution is described, which is then explained in further detail in section 4. The paper ends with a discussion on the received results.

2. Related Work

Remote rendering and visualization methods are not that new, several attempts have been made in the past. Engel et al. [5] for example developed a remote control interface for OpenInventor and Cosmo3D applications. The system transfers compressed images from the server to a Java-based client and returns events generated at the client

* (diepstraten|ertl)@vis.uni-stuttgart.de

† martin.goerke@gmx.net

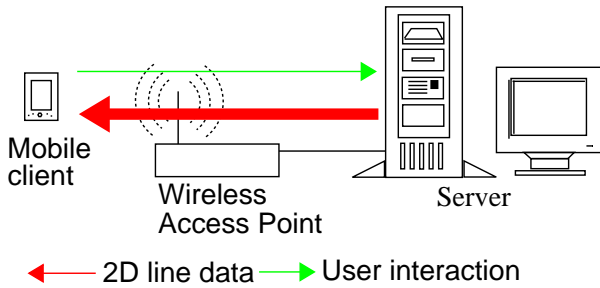


Figure 1. Architecture overview. The thickness of the lines represents the amount of data transferred over the network.

via CORBA requests. Ma and Champ [9] developed a solution for remote visualization of time-varying data over wide area networks. It involves a dedicated display daemon and a custom transport method which allows arbitrary compression techniques. Richardson et al. [13] developed a system which allows remote control of a complete desktop environment. They implemented a simple protocol permitting event transfer and some basic compression methods. Recently Stegmaier et al. [17] developed a truly generic solution, that takes advantage of the transport mechanisms part of the X Window System leading to an extremely compact implementation. In a later enhanced version [16] an additional image transfer channel has been added to allow for arbitrary compression methods. Both versions were successfully tested also on mobile clients.

All these works have one aspect in common, they all use image transfers over networks for remote rendering. Other approaches have been proposed in the past [15] but were not very successful compared to the previous mentioned image streaming architectures. The main disadvantages of these are that they rely on sending large parts of scene data through the network which makes them especially unsuitable for mobile devices, since they have very limited data storage capabilities.

As we derive our 2D lines from feature lines of 3D objects there is a huge overlap with particular research in this area. Numerous works have been published on deciding which lines of a 3D object are important for the visual perception of a human [14, 6, 10]. Also various research has been conducted to find these lines efficiently both in object and in image space [14, 6, 10, 7, 21, 2].

3. Basic concept

The idea of the remote line rendering system is to split the workload between client and server and at the same time reduce the required network bandwidth for each frame. Of

course most of the tasks are still done on the server as the client has limited CPU and especially memory resources, which prohibit to store and to process large amount of data locally. Therefore the work between client and server is split in the following manner:

3.1 Server

The server is doing the complete scene handling, storing all the 3D objects of the scene – if possible – in its main memory and processes them for each render pass. The server also handles the user input events which have been sent by the client, updates the corresponding transformation matrices and starts processing the next frame. It is responsible for finding all the feature lines including the view dependent silhouette lines and transforming, projecting them to 2D window coordinates. Additionally it has to take care of hidden surface removal. As only line primitives are drawn and no filled polygons with depth information, this can not be done on the client. Also before sending the extracted 2D lines it does some packaging of lines to linestrips and additional optimization steps.

3.2 Feature lines

The following different types of feature lines can be distinguished and are found by the server and stored in an extra edge buffer:

- **Boundaries** (see Figure 2(a)), these are edges at the boundary of a triangle mesh and occur when the mesh can not be described through a closed polyhedron. They can be simply found by just looking if the edge in the edge list has two faces or not.
- **Ridges** (see Figure 2(c)), these are edges resulting from a bump in the object surface and often are used to emphasize the shape of the surface. To test if an edge belongs to the group of ridges, the dot product between the normals of the two faces \vec{n}_{face1} and \vec{n}_{face2} connected by an edge is computed and tested against a user-defined threshold value.
- **Valleys** (see Figure 2(d)) are closely related to ridges and can be detected by the same method. To distinguish between a valley and a ridge edge a second vector is needed. This second vector is taken from the edge of the first face pointing away from the current testing edge. Afterwards another dot product is computed between this vector and the normal of the second face, and if its sign is positive it is a valley edge, otherwise a ridge edge.
- **Silhouettes** (see Figure 2(b)), are edges that connect a front-facing face with a back-facing face. They differ

from the previous mentioned feature lines as they depend on the position of the viewer and therefore must be computed for each frame. Our silhouette detection routine is very similar to Buchanan and Sousa [2]. It iterates over all faces and decides if the face is back facing. When the face is back facing all the type flags of the edges of this face will be marked as silhouette by using an XOR bit operation. At the end edges marked as silhouette are silhouettes, all other edges are not. This is due to the fact that either this type will never be set, because only front faces have been found, or in the case of two back faces it is unset again through a double XOR operation.

3.3 Client

The client only has to rasterize the received line packages into its frame buffer and propagate input events back to the server. A schematic illustration of the remote rendering system is shown in Figure 1.

4. Implementation

This section describes how each step is realized for both the mobile client and the stationary server. The mobile client is assumed to be a Pocket PC class type, the server has no general restriction and can be any type that has 3D hardware installed.

4.1 Client

Pocket PC clients currently have a very small screen resolution which is usually 240x320 pixels and 16bit color depth. Hardware accelerated 3D graphics is not yet available for Pocket PCs, some 2D acceleration chips showed up recently in a few devices but are not a general requirement and therefore still not very wide spread. The well-known interface for text- and graphics output of Windows, GDI is also available on Windows CE Pocket PC clients and can be used in the same fashion as on normal Desktop PCs. Right now there is nothing like DirectX available on Windows CE, but for graphically intense applications like games, Microsoft released the Game API (GAPI) [11] for Pocket PCs. This Game API is very limited and it only provides direct access to the frame buffer for blitting. No abstract operations for drawing primitives or copying regions in the buffer are provided. Thus other people have developed several 3rd party libraries [20, 8, 18] which are built on top of GAPI providing a broader range of functionality.

Interesting to note is the significant performance difference between GDI and GAPI, a simple full size back-buffer clear with `bitblt` takes about 21.79ms on a iPAQ 3850 Pocket PC 2002 StrongArm 211 MHz device. While using

GAPI the same operation takes about 6.14ms. To support possible subsequent 2D acceleration in GDI an implementation for both interfaces has been realized. For 2D line rasterization the well-known Bresenham algorithm [1] was implemented in GAPI and in GDI the standard `PolyLine()` function was used.

After each frame has been rendered on the client, a synchronization message is sent to the server to acknowledge the receipt of the next frame. Also stylus interactions and button events will be sent uninterpreted directly to the server.

4.2 Server

In our remote rendering solution the server still has a lot of work to do. Before anything can be displayed to the client, the 3D data has to be loaded, typically from hard disk. Its content is stored in an object list. An object again contains a list of vertices, edges, an index face set of the mesh, and a list of faces. After loading, all edges of each mesh are extruded and stored in a corresponding list. With the help of this edge list the feature lines of the objects are extracted.

After finding all the feature lines the next step for the server is to remove all those lines which are not visible to the viewer. We decided to use an image-space algorithm previously described by Northrup and Makosian [12]. It has the major advantage that it makes use of the hardware-accelerated depth buffer of the graphics card and therefore is very reliable and fast.

In a preprocessing pass all the objects are rendered in their normal solid filled form into the depth buffer. It is wise to use a slight polygon offset while rendering to avoid z-aliasing in the next pass. Next all previously determined feature lines are rendered into depth and color buffer with each feature line having a unique color ID. To ensure that this color ID is unaffected till rasterization all lighting computations should be turned off.

The transformation of the feature lines to 2D window coordinates is done automatically when a 3D hardware library like OpenGL or Direct3D is used. The only problem is to get this 2D data back from the library to send it over the network to a client. To do this we developed two different methods.

Method I The first method uses a special function only available in OpenGL – the feedback buffer. When using the feedback mode of OpenGL, primitives are not rasterized like the usual way but rather the data of the transformed coordinates of each vertex are stored in a special buffer. In this so-called feedback buffer they can be read back by the application. For each feature line the transformed starting and ending point in window coordinates and each color ID

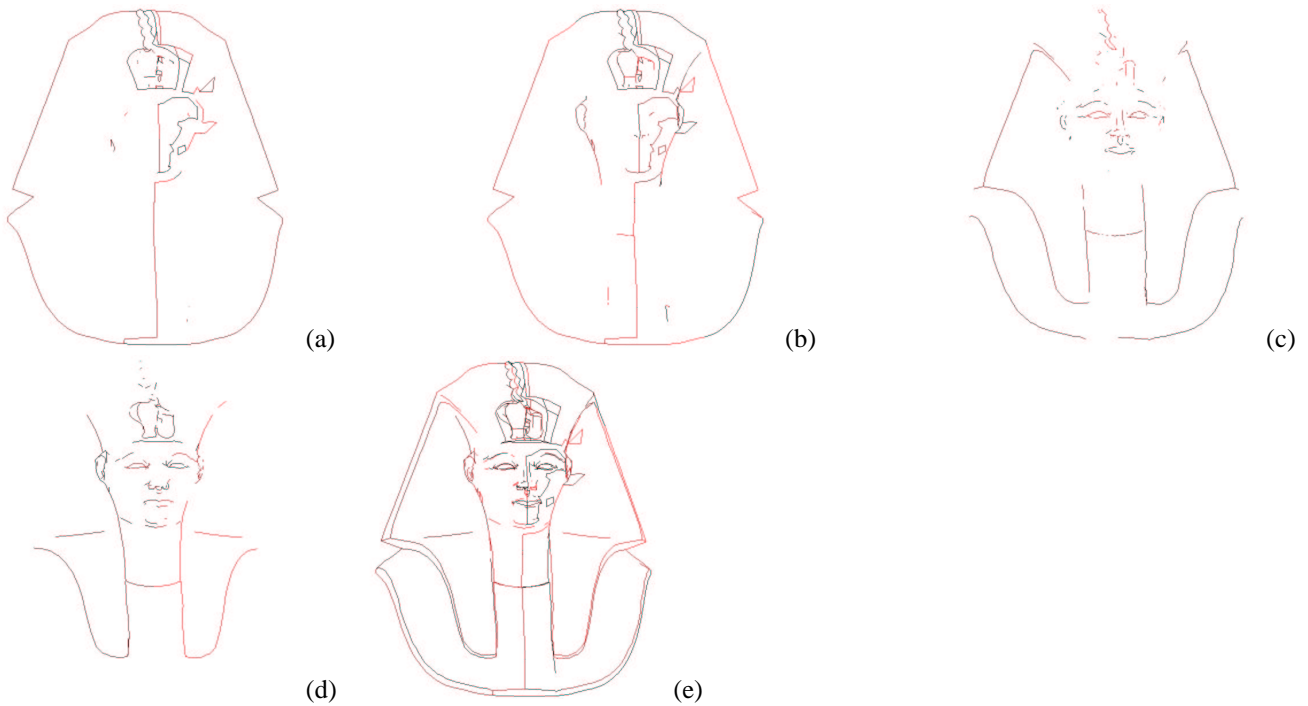


Figure 2. Explanation of different types of feature lines. (a) boundaries, (b) silhouettes, (c) ridges, (d) valleys, (e) all feature lines combined

is stored in the feedback buffer. In the next step the scene is rendered in normal rendering mode. Now the data from the feedback buffer is taken and a scanline conversion of each feature line is performed. But instead of writing pixels to the image plane, pixels are read from the previous generated image and their color is compared with the current color ID. If it matches the color, the current line will be marked as visible. Due to occlusion, lines can be disconnected by other lines. During the scan conversion of the line a visible segment will be generated for each interruption that occurs. At the end all these line segments resample the visible lines, and their start- and endpoints are stored in a list.

Method II The previous method requires two rendering passes, first the pass for feedback mode and then another pass for rendering the lines in the output image. (Feedback mode and normal mode can not be executed at the same time in OpenGL). Additionally a software line rasterization is required to find the visible lines in the output images. To avoid the double rendering and scanline rasterization we propose a second different method. In this method the output image is analyzed and the visible line segments are determined line by line.

For each rendered feature line there is a list of visible segments. If during the scanning of the image a color ID

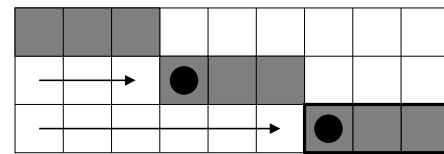


Figure 3. Scenario of what might happen when using bottom up, left to right scanline scanning for finding visible line segments.

is found it will first be checked if there is already a visible segment of the corresponding feature line belonging to this color ID. When there is already a segment belonging to the line the pixel will be tested if it borders the last found segment. In case it does, the segment is extended by the current pixel position. Otherwise if there is no segment, or the last segment is too far away a new segment is inserted.

There is a certain problem when scanning the image from bottom to top and left to right with lines that have a slope smaller than zero and greater than minus one. Figure 3 shows such a scenario. In this figure a segment has been found at the bottom line, but the marked pixel of the next line is too far away from the marked pixel of the cur-

Table 1. Comparison of the efficiency between Method I and Method II for getting the 2D line data.

Number of Lines	230	820	1530	2400
Ratio m_1/m_2	1.53	0.97	0.42	0.37

rently found segment. In this case unnecessary segments might be introduced. To avoid these unwanted segments, first all segments of each line are temporarily stored until the end of the current scanline is reached. Afterwards all temporary segments are connected to each other. Alternatively it is possible to check the current segment after each extension to see whether it can be connected to the previous one.

Depending on the number of feature lines either method I or method II is faster. Table 1 shows the proportion between the two methods for different numbers of lines.

To reduce the network bandwidth even further it is wise to merge single lines to polygon lines if possible. To do this the nine nearest neighboring pixel of each line endpoint are tested in the rendered image if they contain a pixel with a different color ID. If a pixel with a different color ID is found, and the corresponding line is not yet part of the poly-line the location of its endpoints are tested. If one of the endpoints lies also in its 3x3 pixel neighborhood the line will be added to the poly line group.

Afterwards the polylines resulting from this process can be simplified by using a technique described by Douglas and Peucker [4].

5. Results and Discussion

The server component of our remote rendering architecture is implemented in OpenGL using C++ and runs on the Win32 platform. A minimal GUI was implemented on the server to provide the user some control options for rendering. For example the user can decide on the server which feature lines should be extracted. It also has a control output viewport and some additional statistics like current frames per second, connected client IP, etc.

The client has been implemented in Embedded Visual C++ 3.0 and runs on all Pocket PC 2000 and Pocket PC 2002 compatible machines. A rendering interface for GDI and GAPI [11] is provided and the user can choose at startup which interface should be used. When the client connects successfully to the server it switches to full screen mode. Basic navigation like rotate, zoom and translate have been implemented using stylus and button input.

To test the performance of our remote rendering solution,

Table 2. Frame rates achieved on the mobile client with different network settings.

Scene	FPS		# Line	TxRate (in MBit)
	GAPI	GDI		
Bunny	30 (23)	16 (-)	270	11 (1)
Tree	13 (7)	4 (-)	3500	11 (1)
Liberty	19 (10)	5 (-)	1500	11 (1)

three different 3D scenes have been examined, each of them having different characteristics. The first scene (see Figure 4(a)) uses a model of the well known Stanford Bunny; it has 27808 vertices and 52649 faces. Its feature lines basically consist of silhouettes and a few border lines leading to very few lines. The second scene (see Figure 4(b)) uses a tree model with leafs, it contains 32992 vertices and 20352 faces. Due to the many leafs, the scene contains a lot of border feature lines and silhouettes, leading to a high number of 2D lines. The last test scene (see Figure 4(c)) is a model of the Statue of Liberty, it has 19006 vertices and 37186 faces. The line representation contains a high amount of border and valley lines. The overall 2D lines per frame to represent the individual scenes depend largely on the current viewpoint. In the case of the bunny it is approximately 270 lines in average, for the tree scene around 3500, and for the liberty about 1500 lines.

Table 2 shows the received frame rates on an iPAQ 3850 Pocket PC using a 802.11b Wireless Network adapter running first with a TxRate of 11Mbit. In a second test run the access point was artificially slowed down to a 1Mbit TxRate to test a slower network scenario, expecting similar performance of the next generation of wide-area mobile networks, for example UMTS. A Pentium IV 2.8 GHz with a Radeon 9700 Pro graphics board and running Windows XP has been used as server.

As can be seen from the numbers, in the "high" bandwidth case the limitation in frame rates on the client does not seem to relate to the network bandwidth, but rather how fast either the server can produce the necessary lines or the client can actually draw them. This is a nice indication that our approach is going into the right direction, by shifting the problems of remote rendering from the usual network bandwidth issue to other areas. These problems are more likely to be solved with the next generation of hardware, because CPU and GPU power increases a lot faster than network bandwidth. Also it can be seen that the performance gap between the GAPI and the GDI implementation increases with the number of lines, that have to be drawn at each frame on the client. The GDI implementation definitely seems to be a bottleneck right now. It will be interesting to see if mobile

Table 3. Amount of time spent on the server for each task for the three different test scenarios.

Task	Time (\approx in ms)		
	Bunny	Tree	Liberty
Silhouette detection	6.0	1.8	1.5
Visibility	5.6	15.0	30.0
Extract Poly lines	1.1	3.4	11.9
Simplify	0.4	0.4	3.4
Package Data	0.3	0.4	4.7

devices having a 2D accelerator installed can close this gap or even surpass the GAPI implementation in the future.

With the decreased network bandwidth of 1Mbit/s, the system is likely running into the point where the network is actually the limiting factor. However, 7-23 FPS are still respectable frame rates and lie in an interactive range. This means our solution should perform quite well with next generation mobile wide-area networks.

To examine the performance drain in the high bandwidth case we profiled the server to check where most of the time is spent. Table 3 shows the resulting figures. The timings uncover some interesting behavior. In the case of the bunny and the tree scenery, the limitation appears clearly to be on the clients ability to draw the 2D lines. (The total processing time for the bunny is ≈ 13.5 ms and for the tree ≈ 21 ms resulting in possible achievable frame rates of ≈ 74 FPS and ≈ 48 FPS) But in the third scenario it seems that the limitation lies actually on the server itself and not on the client. The total time spent on the server is around 50ms per frame which is about 20 FPS. This is also approximately the number of frames received by the mobile client.

Not surprisingly most of the time on the server is spent either on visibility checking or silhouette detection. Note that the times for visibility checking include a readback from the framebuffer. This operation takes with basic *glReadPixels* ≈ 2.8 ms on the Radeon 9700Pro and ≈ 1.7 ms on the Geforce FX 5800. Further increase might be expected by using special AGP memory areas.

There is also still room for improvements on the client side. First choice would be to use a faster line algorithm or strategies described by Chen et al. [3]. Also as the client has 2D vector data some transformation operations could – to some extent – be completely done on the client (like translating and scaling) without requesting new data from the server. Right now this function is not exploited but could become quite useful if the mobile client runs into a network signal loss. It might be interesting to investigate special line compression schemes to further reduce network load. Right

now we only have tested LZO compression, that leads to a reduction of data by just 3%. Other possibilities could be to use different primitive types for example curved representations.

Another issue which should be addressed in the future is aliasing. Aliasing is a very pressing challenge especially on small screens where it becomes quite noticeable. Unfortunately on current generation of mobile devices there are not enough resources left to cope with these problems.

As conclusion we think that rendering and transferring vector data will in the long term be a better solution than the common image-based remote rendering systems, especially with upcoming new displays for mobile devices, that allow full VGA or even SVGA resolutions.

Acknowledgement

We would like to thank the anonymous reviewers for their useful comments for improving this paper. This project is funded by the Deutsche Forschungsgemeinschaft within the Center of excellence 627 “World Models for Mobile Context-Based Systems”

References

- [1] J. E. Bresenham. Algorithm for Computer Control of a Digital Plotter. *IBM Systems Journal*, 4(1):35–40, 1965.
- [2] J. W. Buchanan and M. C. Sousa. The edge buffer: a data structure for easy silhouette rendering. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, pages 39–42. ACM Press, 2000.
- [3] J. Chen, X. Wang, and J. E. Bresenham. The Analysis and Statistics of Line Distribution. *Computer Graphics & Applications*, 22(6):100–107, 2002.
- [4] D. H. Douglas and T. K. Peucker. Algorithm for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.
- [5] K. Engel, O. Sommer, and T. Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 167–177, May 2000.
- [6] A. Gooch. *Interactive Non-Photorealistic Technical Illustration*. PhD thesis, University of Utah, 1998.
- [7] B. Gooch, P.-P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999.
- [8] Inmar Software. Diesel Engine SDK. <http://www.inmarsoftware.com/>, 2003.
- [9] K.-L. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Supercomputing*, 2000.
- [10] L. Markosian, M. Kowalski, S. Trychin, and J. Hughes. Real-time non-photorealistic rendering. In *Proceedings of SIGGRAPH 1997*, pages 415–420. ACM Press, Aug. 1997.

- [11] Microsoft Corporation. Using the pocket pc game api. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/guide_ppc/html/_gamex_using_the_gapi_run_times.asp, 2003.
- [12] J. D. Northrup and L. Markosian. Artistic silhouettes: a hybrid approach. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, pages 31–37. ACM Press, 2000.
- [13] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
- [14] T. Saito and T. Takahashi. Comprehensible rendering of 3-D shapes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, volume 24, pages 197–206, 1990.
- [15] D. Schmalstieg. *The Remote Rendering Pipeline*. PhD thesis, Vienna University of Technology, 1997.
- [16] S. Stegmaier, J. Diepstraten, M. Weiler, and T. Ertl. Widening the Remote Visualization Bottleneck. In *Proceedings of ISPA '03*. IEEE, 2003.
- [17] S. Stegmaier, Marcelo Magallón, and T. Ertl. A Generic Solution for Hardware-Accelerated Remote Visualization. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '02*, 2002.
- [18] T. Tremblay. PocketFrog SDK. <http://pocketfrog.droneship.com/index.html>, 2003.
- [19] E. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphic Press, 1997.
- [20] Viktoria Institute. GapiDraw graphics SDK. <http://www.gapidraw.com>, 2003.
- [21] H. Zhang and K. Hoff. Fast backface culling using normal masks. In *Proceedings 1997 Symposium on Interactive 3D Graphics*, pages 103–106, 1997.

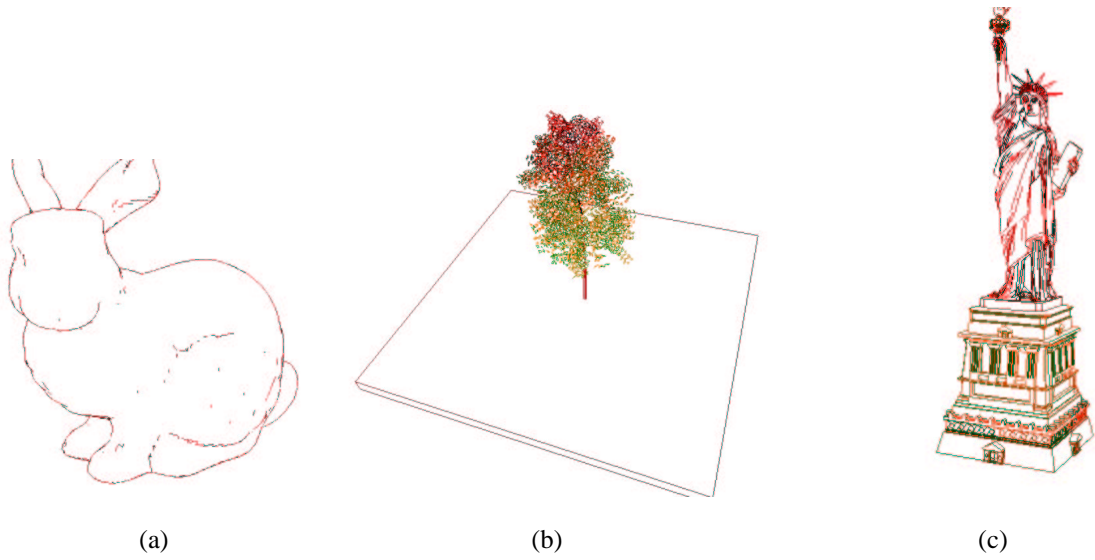


Figure 4. The three different test scenarios in their line representation. (a) Stanford Bunny model, (b) tree model, (c) Statue of Liberty

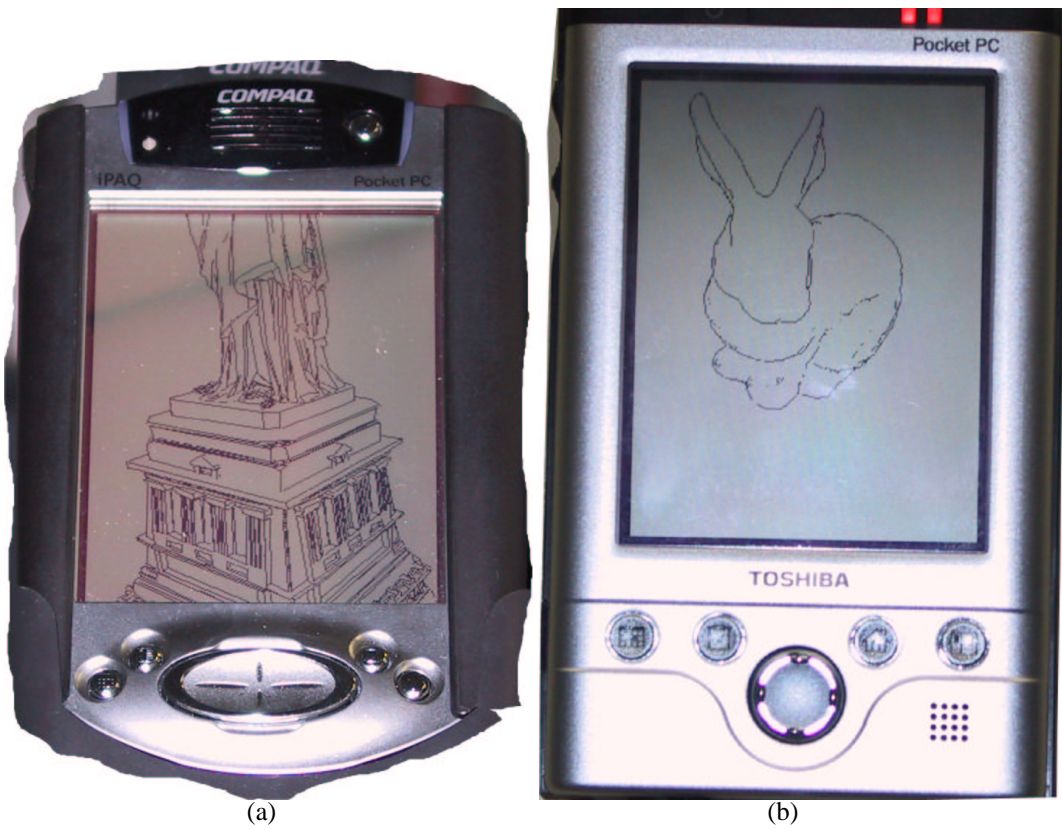


Figure 5. Two images showing our system actually running on the mobile client. (a) Statue of Liberty model on a iPAQ 3850. (b) Stanford Bunny on a Toshiba e740