Interactive high definition 3D face rendering on common mobile devices

Peter Barth, Ulrich Schwanecke, Friederike Wild

University of Applied Sciences Wiesbaden

Abstract

Smartphones are suitable as interactive visualization platform for high quality 3D models representing, for example, faces. We provide an efficient implementation of a point-based renderer in Java. A 3D-model is adequately reduced in size, represented as memory efficient octree and converted to Java code for packaging and deployment on smartphones supporting Java Mobile Edition (JavaME). The render algorithm is carefully implemented in platform independent Java and performs well on common mobile devices. While interacting with the model, we routinely achieve more than 10 frames per second providing for an agile user experience, while sustaining perceived high quality by the user.

1 Introduction

As smartphones, mobile devices have entered everyday life as personal items, while remaining prestigious products. At the same time, mobile devices hardware has advanced sufficiently to interactively display highly detailed 3D models (Duguet & Drettakis 2004). Consumers often have pictures of relatives and best friends always with them. While pictures can already be stored and displayed on smartphones, having an interactive 3D model, such as in figure 1, is bound to attract consumers. Thus, a viewer application needs to run on most smartphones, the visualization needs to be of high quality and the interaction such as scaling and rotation needs to be smooth and agile.

The number of smartphones in the world that support Java has surpassed one billion in 2006 (Pulli et al. 2007). Today, most mobile phones ship with Java support. However, only new and high end devices start to provide hardware support for 3D graphics. Thus, on common mobile phones we can only expect 2D graphics and solid Java support. Therefore, we provide an efficient implementation of an interactive point-based renderer using octrees (Botsch et al. 2002) relying only on JavaME features that are supported on common devices (Williams & Burge 2004) found in the market since 2004 such as the ones shown in figure 2.

Figure 1: Three different views of 3D model on smart phone

Figure 2: Panasonic X700, Nokia N70, Nokia N95 showing different views of 3D model

In the following section we review point-based rendering. The design decisions for the interactive point-based rendering approach and its implementation in Java are presented in section 3. We detail the preparation of the 3D model allowing for effective rendering and reduction of visual artefacts on the final image in section 4. Section 5 describes the user interaction of the viewer and then focuses on the efficient Java implementation of the renderer. Finally, we evaluate performance and user experience in section 6 and conclude in section 7.

2 Point-based Rendering

There are three different concepts to achieve interactive 3D rendering on mobile devices, among them are *image-based* (Chang & Ger 2002, Lluch et al. 2005), *point-based* (Duguet & Drettakis 2004, He & Liang 2006) and *polygon-based rendering* (Pulli et al. 2005). On the one hand, image-based systems replace most of the 3D data by precomputed images. These images are rendered at a fast stationary server system and send to the mobile client for remote display. Limited bandwidth and round-trip times greater than 100 ms of current GSM as well as 3G networks makes image-based interactive rendering unsuitable for mobile devices. On the other hand, polygon-based systems need hardware acceleration implementing the standard rasterization pipeline. Even though mobile devices are increasingly equipped with graphics hardware, this is still not common. Thus, polygon-based rendering of faces can not be done at interactive frame rates on the majority of common mobile devices.

Point-based rendering (PBR) uses points instead of polygons as graphic primitives and displays objects by a set of points located on its surface. Note, that for complex scenes or small display resolutions a rasterized polygon often covers less than a pixel. As point-based rendering can be implemented very efficiently, it is an adequate rendering concept for common mobile devices. Although the human brain is fault-tolerant when it comes to face recognition (Bruce & Young 1986), it is very sensitive regarding finest details. For a high definition face visualization details of e.g. facial expressions need to be preserved focussing on relevant parts such as eyes, nose, and mouth. The 3D face models in our setting are high resolution range images consisting of several hundred thousand points generated by a single measurement of a structured light 3D scanner. These models preserve most information in frontal view, which makes them suitable for face visualization. Point-based rendering is

preferable on limited hardware as, in contrast to polygon-based rendering, more details with the same amount of memory can be represented. Due to the simplicity of the drawing primitive, it can also be rendered much faster. (Botsch et al. 2002) have introduced a compact representation and a fast rendering method for a point-based viewer, which makes it suitable for mobile and thus resource-constrained devices (Duguet & Drettakis 2004).

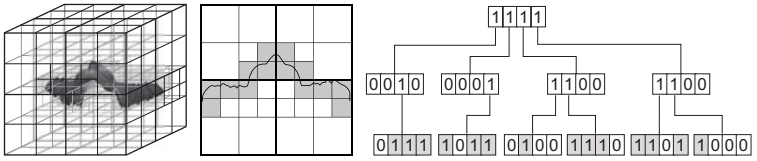


Figure 3: Point set with octree of depth three and quadtree-representation of its projection

The key to interactively rendering large point sets is an efficient hierarchical representation. Recursive grids such as *p*-grids (Duguet & Drettakis 2004) are appropriate spacial data structures for representing point sets due to their flexibility and fast traversal algorithms. In a *p*-grid each cell is uniformly subdivided into p^3 sub cells. For e.g. a two-grid (p=2) each cell is subdivided into eight sub cells, which results in the well known *octree* (see figure 3). We use octrees because they are the optimal *p*-grids regarding storage and rendering efficiency if no intermediate samples are stored (Duguet & Drettakis 2004). An octree has a compact representation as byte stream requiring less than three bit to encode a three-dimensional position of a point. To reconstruct the location of the encoded points from the stream of byte codes, the octree can be linearly traversed (see algorithm 1). During reconstruction only addition of well scaled numbers are required. For scaling or rotating the point set costly matrix vector operations need only be applied on eight points (Botsch et al. 2002).

Algorithm 1: decode(x, y, z, l, octreeIndex)

```
if l = 0 then
    setPixel(x,y,z);
else
    octreeByte ← octree_octreeIndex ;
    octreeIndex ← octreeIndex+1;
    for k ← 0, …, 7 do
        if octreeByte_k then
            vec ← displacementVectors_l-k;
            octreeIndex ← decode(x+vec_x, y+vec_y, z+vec_z, l-1, octreeIndex);
return octreeIndex;
```

3 Interactive Mobile PBR in Java

Based on the restrictions of mobile environments we adhere to the following design decisions: We limit spacial resolution of the 3D model to 256^3 grid cells. During rotation we reduce resolution further to 128^3. The renderer supports 256 colors, sufficient for representing faces. The implementation uses fixed point arithmetic and only Java 1.3 features.

Smartphones have a typical display size of 320x200. A depth of eight (256^3 resolution), is sufficient for a high quality representation. In order to achieve high frame rates even on older hardware, we use a lower resolution, a depth of seven (128^3) during rotation. This gives high frame rates while maintaining a good user experience. As the user focuses on adjusting the position while rotating, performance is more important than details. The resulting artefacts of the lower resolution are perceived more as blur or ghosting (Siegel 2000).

We restrict ourselves to the fixed lighting unaltered from the scan process, which eases computation considerably. A face with a constant light source in front of the face independent of the rotation provides a good experience for the watcher. The regions, that are relevant for cognition (eyes, nose, and mouth) are emphasized. The used 3D representation is most detailed in frontal view, but tends to exhibit holes when rotated and looked at from the side. Facial skin does not come with a wide range of different colors (Jones & Rehg 1999) and face recognition works equally well on gray scale images as used in the examples. We restrict ourselves to 256 arbitrary colors from a look up table.

The implementation uses only Java 1.3 features found on all JavaME smartphones. We use fixed point arithmetic, which is faster and makes the application deployable on devices without floating point support, which are popular among business users providing longer battery life. We favor Java runtimes that provide hotspot techniques (Sun 2002), such as Symbian OS phones (Symbian OS 8.x), but do not exclude other JavaME smartphones.

4 Preparation and Finishing

The initial input data from the scan process consists of several hundred thousand points. This amount of data can not be rendered interactively on most common mobile devices. Hence, the number of points given by the original range images has to be reduced to a suitable size. This is automatically performed by the uniform clustering nature of the octree approach. To further reduce the number of points, we eliminate all cells that are never visible. To this end, we merge all cells that are visible from at least one of several viewpoints (see figure 5, left). This preparation step results in a more compact octree representation of the 3D face models. However, this smaller model still produces an identical frontal view compared to the unaltered model. As this step is performed on the server it is not time-critical.

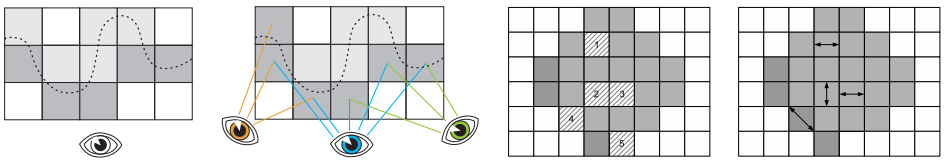


Figure 4: Hidden points removal (left) and hole filling (right).

Due to performance reasons, we render at most one pixel while projecting an octree cell. This is not a problem for the initial frontal view, but when the model is rotated or scaled. In

this case, the limited amount of data can result in holes impairing the image quality. Thus we fill these holes in a 2D finishing step following the rendering of the octree. We use a simple scanline algorithm based on a neighborhood criterion that closes holes of type 1-4 (see figure 4, right). Elimination of potential holes at the border (type 5) is not attempted. For the visual effect of the described hole filling step see figures 6 and 7.

5 Interactive Rendering

Quickly creating 2D images of the 3D model from changing perspectives is the main goal in the rendering phase. Thus, an efficient implementation of the rendering process is necessary. Still, image creation and display would consume all available computing resources. Without further measures, the user interface would be sluggish resulting in a bad interaction experience. First, we describe how we achieve a good user interaction perception, then we detail some optimizations for the implementation of the rendering phase in the Java environment.

The user directs rotation with the navigation keys of the device and expects a fixed rotation speed, which can easily be achieved by separating model view transformation and rendering. On slow devices the frame rate drops, but rotation speed remains constant. This is common in interactive simulations and supported by the Java Game API, which is used. However, always rendering the high definition model (256^3 resolution) is too time consuming for many older smartphones. But a resolution of 128^3, which is about eight times faster, is still acceptable during rotation. Therefore, the interactive rendering algorithm displays 2D images derived from the 128^3 resolution model during rotation and immediately after the user has released the keys, renders a 2D image derived from the 256^3 resolution model displaying a high definition freeze image. In addition, we apply the 2D finishing as described in section 4.

Figure 5: Unscaled 8 bit depth, 8 bit depth finishing, 7 bit depth, 7 bit depth finishing.

In figure 5 we see first a plain image generated out of the 256^3 model slightly rotated. The visible artefacts are eliminated by the finishing process. The second image is the one the user will see after rotation stops. During rotation the user sees the fourth image, which is generated from a 128^3 model (third image) plus finishing. The effects of the finishing process are significant. Examining the image series with a slightly scaled image in figure 6, we see the effects of the finishing process in the high resolution freeze image. With seven bit, the image quality is acceptable for rotation. The finishing step helps, but visible artefacts remain.

Figure 6: Scaled 8 bit depth, 8 bit depth finishing, 7 bit depth, 7 bit depth finishing.

Thus, even on older smartphones, the rendering and display of the high definition model is achieved within at most 200-300 ms. Typically, users do not realize that the image is rendered twice. Reactions to user input below 300 ms after the action (releasing a key) are not perceived as wait time. The rendering algorithm uses two threads, one for the high definition rendering (H) for freeze images and one for the low definition rendering (L) used while rotating. H starts as soon as the next frame is to be displayed (every 100 ms) and no key is pressed, which is necessary to reduce perceived delay. If no key is pressed after H has finished the last image is kept and both threads wait, not wasting precious power resources.

Unfortunately, a common user interaction pattern is changing the rotation direction quickly in order to achieve both a horizontal and a vertical rotation as shown in figure 7. For example, the user holds the vertical navigation key and then instantly, moves the thumb to the horizontal navigation key. In effect, the user releases the navigation key for a short time period, which is sufficient to start H. However, in less than 300 ms, a new key is pressed. Thus rotation is started, which triggers L, while H still runs. To sustain an agile user experience L must starts instantly and display its results to avoid ”sluggishness”. In order to not waste computation and thus not hinder L, we need to stop H as soon as a key is pressed.

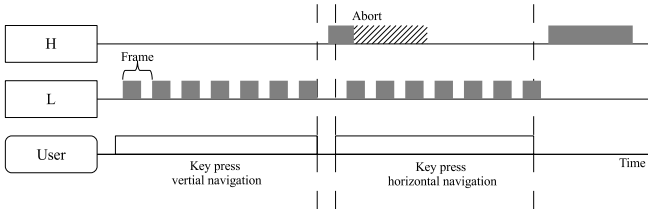


Figure 7: User interaction changing position and rendering threads.

Stopping is requested via a shared variable available to H. In order to achieve both high performance and quick response, the check points of that shared variable need to be carefully chosen. Fortunately, this can be easily achieved by checking at a fixed tree depth whether computation needs to be aborted. We chose depth 4 which is reached around 512 times per rendering phase meaning computation can be aborted within a millisecond, while the test itself is mostly executed 512 times and thus not hindering performance of the rendering.

Having reduced computation time by reducing computation needs, we now focus on the implementation of the rendering algorithm. Modern Java runtime environments can execute well written code almost as efficient as similar implementations in C. In the implementation of the rendering algorithm, we avoid common performance pitfalls, such as useless object creation (Shirazi 2002). In addition, most mobile devices have either weak floating point performance or no floating point support at all. Thus, we implemented fixed point arithmetic.

The 3D model is represented as array of characters to the decoding algorithm. However, packaging and delivery of the 3D model to the viewer is, contrary to expectation, not simple. The obvious approach is to bundle data and viewer in an installable package (JAR) and then extract the data during runtime. While this is supported by JavaME, some smartphones do not adequately support reading data files. In addition, the amount of memory needed to read the data is more than twice as large as the data itself, which tends to exceed available heap memory on some devices. It appears that regardless of the data used the entire contents of the JAR file are read into main memory. Thus, we have chosen to compile the data stream to Java code and bundle the resulting class files containing the data and the viewer. The data stream is split into small chunks and distributed over several classes as static members. The split into several classes was necessary because the Java virtual machine limits classes to contain at most 65536 bytes (Lindholm & Yellin 1999). During runtime one sufficiently large array is allocated and filled. The size of the installable package is not increased compared to storing the byte array as file. On the heap only the size of the final array representing the data stream is occupied. This approach worked on all tested devices (see section 6).

The main part of the renderer is the recursive decoding algorithm computing all 2D points of a scaled and rotated 3D model (algorithm 1). Note, that the last recursion step reaches a leaf of the tree, where only a point is rendered. Thus, computation and copying all information, as is necessary for the inner nodes, is wasted. In addition, the number of recursive calls of a leaf dominates the number of all recursive calls. In a full octree at depth k we have about $8^k/7$ (sum of 8^i for $1 \leq i < k$) inner recursive calls and 8^k recursive calls for leafs. Hence, over 85 percent of all recursive calls may only set a pixel. As for a surface on average only half of the child cells contain voxels, we have for depth k=8 approximately $4^k = 65536$ calls for leafs. Still over 66 percent of all recursive calls are leaf calls. Thus, eliminating the last recursive call has the most impact and gives algorithm 2. Further recursion elimination was not attempted, as there is no useless computation or copying of data involved. Finally, we have unrolled the fixed size **for**-loop iterating over the eight different sub cells.

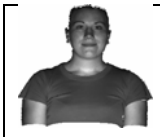
Algorithm 2: decode(x, y, z, l, octreeIndex)

```
octreeByte ← octree_octreeIndex ;
octreeIndex ← octreeIndex+1;
for k ← 0, …, 7 do
    if octreeByte_k then
        vec ← displacementVectors_l-k;
    if l = 1 then
        setPixel(x + vec_x, y + vec_y, z + vec_z);
    else
        octreeIndex ← decode(x+vec_x, y+vec_y, z+vec_z, l-1, octreeIndex);
return octreeIndex;
```

As many devices do not support floating point hardware, we use fixed point arithmetic for all non integer computation. In order to compute the x, y, and z coordinates of a 3D point of a rotated and scaled model a series of precomputed amounts are successively added (Botsch et al. 2002). As typical views always depict the entire face, scaling up is limited to a small factor. Artefacts up to a factor of two can almost completely be eliminated (see figure 6). The model fits into a cube with edge length 512 requiring 9 bits in original size. Providing 17 bits for the integer parts leaves 15 Bits precision, this seems to be more than adequate. With fixed point arithmetic with 32 Bit integer numbers and only addition as mathematical operation during tree traversal we do not expect numerical issues.

6 Evaluation

For all tests we use the 3D model from figure 1. The tests were run on different devices as shown in figure 2 ranging from an older Panasonic X700 to a modern Nokia N95.



	original	depth 8	depth 7
spacial resolution	-	256^3	128^3
number of points	443163	67741	19076
size (incl. color)	-	91.5 Kbyte	25.4 Kbyte

Figure 8: Model used for evaluation

	Operating System	Processor	Display resolution	number of colors
Panasonic X700	Symbian OS 7.0s	ARM-920T, 104 MHz	176x208	65536
T-Com MDA Pro	Windows Mobile 5	XScale PXA270, 520 MHz	640x480	65536
Nokia N70	Symbian OS 8.1a	ARM-926, 220 MHz	176x208	262144
Nokia N95	Symbian OS 9.2	ARM-11, 330 MHz	240x320	16777216

Figure 9: Devices used for evaluation

In figure 10 we show the effects of the different implementation enhancements to the decoding algorithm. We have only measured the time to compute all image points of the frontal view without actually displaying the image (as displaying an image is provided by the Java library and not affected by our implementation).

	Floating Point			Fixed Point		
	baseline	recursion elimination	loop unrolling	baseline	recursion elimination	Loop unrolling
Panasonic X700	-	-	-	361	283	134
T-Com MDA Pro	344	286	283	90	67	60
Nokia N70	355	306	278	160	116	91
Nokia N95	221	196	598	96	75	473

Figure 10: Time (msec) per image, 256^3, frontal view, different implementations

The baseline implementation of the rendering algorithm corresponds to algorithm 4. Recursion elimination corresponds to algorithm 9. As expected hardware floating point support on mobile devices cannot compete with fixed point arithmetic and, if used, is the limiting factor. Careful optimization, as detailed in section 5, brings significant performance improvements and allow to reach five frames per second (including display of the image) even on the Panasonic X700. By reducing the resolution to 128^3 during rotation, we easily reach ten frames per second on the Panasonic X700 and even slower devices. Rendering instantly the high resolution model and displaying it, as explained in section 5 is done within 200 msec. Thus, we sustain perceived high quality by the user.

We have observed an interesting side effect of the loop unrolling of the fixed size **for**-loop in algorithm 9. Figure 10 shows varying performance improvements depending on processor and runtime infrastructure. We see a significant improvement of this optimization on the Panasonic X700 and a smaller, but still measurable improvement on the MDA Pro and the N70. However, performance terribly degrades on the N95 by almost an order of magnitude. We suspect, that the hotspot compiler on the N95 decides to *not* compile the unrolled loop to native code based on the increased code size, if the loop is unrolled. This shows the importance of careful engineering and rigorous testing on different devices.

We tested usability of the interaction with selected computer science students, all experienced in using computers and mobile phones. All of them quickly identified and intuitively understood the meaning and effect of the navigation keys. The agile and simple user interface requires no explanation for rotating and scaling the model. Some users did not realize that the model can be rotated at all. Based on this feedback, the application now initially rotates the model until the first key is pressed. It is planned to distribute the application commercially to consumers as ready to install package with integrated model and viewer. Due to the platform-independent nature of Java it was straight forward to use the very same rendering implementation and precompiled model to build applets for viewing in a Web browser and build a stand alone desktop viewer. As performance and size is no issue on even older Desktop PCs we typically use a model at higher spatial resolution, e.g. 512^3.

7 Conclusion

We have adapted a point-based rendering algorithm for interactive rendering of high definition face models on common mobile devices. We set out with the clear goal to maintain agile interaction at 10 frames per second for the user.

It is possible to achieve this goal with a preprocessed 3D model of over sixty thousand points in Java on *modern* smartphones. Taking into account user perception of rotating models, it is even possible to achieve this frame rate on *common* smart phones while sustaining perceived high quality through reduction of the resolution while rotating. To this end, the rendering algorithm had to be carefully implemented and adopted to allow for rendering of different depths and aborting of the rendering at external events while maintaining high performance. This was only possible with rigorous performance and functionality testing on different

target devices and continuously verifying deployability and responsiveness during interaction. Although Java fulfills its promise of platform independence, performance critical sections need to be validated per platform, at least on mobile devices. With increasing hardware capabilities of mobile devices additional quality improvements, such as advanced shading and higher resolution models for high resolution displays can be supported in the future. However, this would limit deployment to these modern devices.

Acknowledgment

We thank LOOXIS, a company specializing on 3D face scans lasered into high quality glass, for providing a 3D face scanner and supporting this applied research in a joint project.

Literature

Botsch, M. & Wiratanaya, A. & Kobbelt, L. Efficient high quality rendering of point sampled geometry. *Eurographics workshop on Rendering*, Switzerland, 2002. Eurographics Association.

Bruce, V. & Young, A. Understanding face recognition. *The British Journal of Psychology*, 77(3):305-327, 1986.

Chang, C. & Ger, S.H. Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering. *PCM '02: IEEE Conference on Multimedia*, London, UK, 2002. Springer-Verlag.

Duguet, F. & Drettakis, G. Flexible Point-Based Rendering on Mobile Devices. *IEEE Computer Graphics and Applications*, 24(4):57-63, 2004.

He, Z. & Liang, X. A Point-Based Rendering Approach for Mobile Devices. *International Conference on Artificial Reality and Telexistence – Workshops, 2006. ICAT '06.*, 2006.

Jones, M.J. & Rehg, J.M. Statistical color models with application to skin detection. *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, 1:274-280, 1999.

Lindholm, T. & Yellin, F. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

Lluch, J. & Gaitán, R. & Camahort, E. & Vivó, R. Interactive three-dimensional rendering on mobile computer devices. *ACE '05: ACM SIGCHI,* pages 254-257, New York, NY, USA, 2005. ACM.

Pulli, K. & Aarnio, T. & Roimela, K. & Vaarala, J. Designing graphics programming interfaces for mobile devices. *Computer Graphics and Applications, IEEE*, 25(6):66-75, Nov.-Dec. 2005.

Pulli, K. & Vaarala, J. & Miettinen, V. & Simpson, R. & Aarnio, T. & Callow, M. The mobile 3D ecosystem. *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 1, 2007. ACM.

Shirazi, J. *Java Performance Tuning*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

Siegel, M. "Just enough reality", microstereopsis, and the prospect of zoneless autostereoscopic displays. *Image Processing, 2000. Proceedings. International Conference on*, 1:9-12 vol.1, 2000.

Sun Microsystems. *The CLDC HotSpot Implementation Virtual Machine*. whitepaper, 2002.

Williams, C. & Burge, M. MIDP 2.0 changing the face of J2ME gaming. *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 37-41, New York, NY, USA, 2004. ACM.