Pseudo-Immersive Real-Time Display of 3D Scenes on Mobile Devices

Ming Li, Arne Schmitz, Leif Kobbelt
Computer Graphics Group
RWTH Aachen University
Aachen, Germany
http://www.graphics.rwth-aachen.de

***Abstract*—The display of complex 3D scenes in real-time on mobile devices is difficult due to the insufficient data throughput and a relatively weak graphics performance. Hence, we propose a client-server system, where the processing of the complex scene is performed on a server and the resulting data is streamed to the mobile device. In order to cope with low transmission bitrates, the server sends new data only with a framerate of about 2 Hz. However, instead of sending plain framebuffers, the server decomposes the geometry represented by the current view's depth profile into a small set of textured polygons. This processing does not require the knowledge of geometries in the scene, i.e. the outputs of Time-of-flight camera can be handled as well. The 2.5D representation of the current frame allows the mobile device to render plausibly distorted views of the scene at high frame rates as long as the viewing direction does not change too much before the next frame arrives from the server. In order to further augment the visual experience, we use the mobile device's built-in camera or gyroscope to detect the spatial relation between the user's face and the device, so that the camera view can be changed accordingly. This produces a pseudo-immersive visual effect. Besides designing the overall system with a render-server, 3D display client, and real-time face/pose detection, our main technical contribution is a highly efficient algorithm that decomposes a frame buffer with per-pixel depth and normal information into a small set of planar regions which can be textured with the current frame. This representation is simple enough for realtime display on today's mobile devices.**

***Keywords*-mobile; rendering; 2.5D textured polygon; transmission; client-system; interaction**

I. Introduction

Current off-the-shelf mobile devices are equipped with powerful processors, graphic processing units, multiple sensors, and high speed wireless connections. It has already been a common feature for these devices to render 3D geometry. However, at the same time, the advances in 3D games, navigation, mobile GIS, and other applications raise great demands and expectations for real-time 3D display on mobile platforms. The typical data sets in these applications often exceed the available memory size and rendering capabilities even on state-of-the-art high-end handheld devices. Therefore it is still a challenging task to display a complex, textured 3D scene smoothly on such devices.

Various algorithms have been proposed to bridge the gap between hardware limitation and complexity of geometry data. One of them is progressive meshes. Progressive models can be constructed by mesh decimation, which generates a stream of meshes all representing the same object with different levels of detail. Depending on the distance between the viewer and the object, a suitable level of detail is chosen to be rendered. However, progressive models actually require more storage in memory compared to plain polygon meshes. Moreover, some 3D scenes cannot be decimated efficiently, since the individual objects already consist of primitive geometries. In addition, the decimation of certain 3D objects is not straightforward, e.g. trees and leaves. The performance depends on the number of faces as well. Other researchers suggest to use imposters and/or a 3D image cache to improve rendering performance. But they require the knowledge of the 3D scene for segmentation or partition of the space. For range images from a time-of-flight (TOF) camera, these methods can not be used directly.

In this work, we present a client-server system which can automatically simplify complex 3D scenes in real-time and display them interactively on handheld devices. On the server side, the depth and normal information are taken either from a 3D rendering server or from a TOF camera. By using these information of the current view, we extract a coarse 2D mesh using feature edges in the image as constraints. We further exploit the depth value of each vertex to get a 2.5D mesh representation, which approximates the current view of the original 3D scene. This textured mesh is then streamed to the mobile client for rendering. In order to further augment the visual experience, we utilize the mobile device's built-in sensors to detect the spatial relation between the user's face and the device, using the front-facing camera to detect the face position or using the gyroscope to detect the device pose. We adjust the view frustum according to this position/pose.

For the interaction we designed two use cases: One is a "tour guide" mode, in which a pre-defined camera path leads through the virtual scene. The other one is a "free walk" mode, where the user can change the viewing position by using touch gestures, and adjust the viewing orientation by changing the relative position of his face to the front-facing camera.

Besides designing the overall system with a render-server, 3D display client, and real-time face/pose detection, our main technical contribution is a highly efficient algorithm

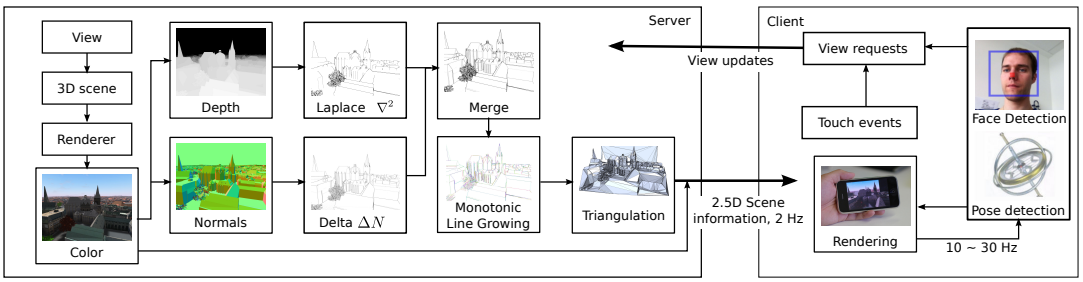


Figure 1. Architecture of our client-server system. Geometry updates for the client are sent at 2 Hz, interaction via face/pose detection on the client is done at 10 to 30 Hz (using face detection or gyroscope).

that decomposes a frame buffer (or a range image) with per-pixel depth and normal information into a sparse set of planar regions which can be textured with the current color framebuffer (or RGB image). Compared to mesh decimation algorithms, our approach performs much faster and can decimate any image rendered by a system providing depth and normal information, such as OpenGL or a TOF camera. Our resulting coarse mesh representation is simple enough for real-time display on today's mobile devices.

II. Related work

There are various methods to extract feature edges and detect line segments from 2D images. Isenberg et al. [1] provided a survey on algorithms for computing polygonal model silhouettes. Von Gioi et al. [2] proposed a parameter-less linear-time line segment detector which gave accurate results. Compared to state-of-the-art line segment detection algorithms, their approach is fast and can reduce the false detection of edges for natural images. But the method is not suitable for depth images, especially for organic objects that have smooth surfaces, because no line segments are detected on the surface.

When trying to construct a coarse mesh, one can use mesh decimation to go from a dense sampling to a less dense one. There has been an enormous amount of research in this area of computer graphics. Gotsman et al. [3] presented a survey onf simplification methods and 3D mesh compression techniques. A good overview of state-of-the-art algorithms are given in the book by Botsch et al. [4]. Many of these methods are not fast enough for our application. One fast stochastic decimation model has been introduced by Wu and Kobbelt [5], but it still does not achieve the performance needed for real-time streaming. Decimation methods start from a complex model and simplify it, whereas our method constructs a sparse mesh incrementally from visible feature edges. Hence we never construct a complex model and can achieve near real-time processing speed.

Some researchers reconstructed the mesh representation from depth images. Pulli and Pietikäinen [6] suggested a range image segmentation method based on depth information and surface normals. Pajarola et al. [7] proposed a fast depth-image meshing algorithm based on restricted quadtree triangulation [8]. In their results, they only show single objects or simple 3D scenes consisting of a few objects. For complex geometry scenes, e.g. urban scenery, this method could result in a dense mesh for transmission and visualization.

A problem of the range-image based mesh reconstruction is the "rubber sheets" at silhouette boundaries between foreground and background objects due to depth discontinuity. Mark et al. [9] avoided this occlusion artifacts by warping 2 different reference frames and compositing the results. Pajarola et al. [7] blended several reference depth-meshes to synthesize new views.

For visualization of urban scenery, imposter based approaches were introduced. Sillion et al. [10] segmented the urban scene into 2 parts, the local 3D model and a set of imposters used to represent the distant scenery. The segmentation is based on the city-block-structure. To extend cache life of imposters, the depth information of the distant landscape is used to build a sparse textured mesh, which is similar to our approach. But their augmented imposter only considers depth discontinuity, while our method also takes normal discontinuity into account, which gives a more accurate representation of building facades. Similarly, Decoret et al. [11] improved the imposters by using multi-layered imposters to reduce the occlusion error. Imposter based approaches are good solutions for urban scenery visualization, but they are not suitable for general 3D scenes or range images. First, the knowledge about 3D scenes and city blocks is not always available for the segmentation. Second, the viewpoint is restricted to the ground level for city walk-through.

To reduce the workload of static 3D scene rendering, Schaufler et al. [12] combined concepts of imposters, hierarchical scene subdivision and levels of detail in order to cache three dimensional images of a virtual environment. A similar approach is also introduced by Shade et al. [13]. These algorithms need knowledge of the 3D scene for partitioning, which is not known for TOF camera input.

A system that uses view dependent simplification for rendering on mobile devices was developed by Lluch et al. [14]. The authors implement this as a client-server rendering architecture. Their system is useful, but limited to triangle

based geometry, whereas our approach can take any input, as long as depth and normal buffers are supplied.

Lee [15] implemented an on-screen view into a 3D world using face tracking. He used a Wii remote to track the user's head movements. This is not practical for our scenario, where the user has only a small handheld device, without any extra input devices. This is why we opted to use face/pose detection instead.

A time-of-flight camera (TOF camera) is a camera system which can capture distance information using the time-of-flight principle [16]. Hirschmuller and Scharstein [17] built a dataset of stereo images including color and depth images. Microsoft released Kinect for its game console Xbox 360 [18], which uses Light Coding technique [19] to generate depth images. OpenKinect [20] provides open source libraries for using Kinect. OpenNI [21] recently released their skeleton tracking sdk for kinect development.

III. System Architecture

The system we present in this paper is based on a client-server architecture. The server renders a complex 3D scene using OpenGL and takes the per-pixel depth and normal information to detect feature edges. After that we perform a line growing algorithm to extract monotonic feature segments. These are converted from a pixel representation to a set of continuous line segments using an algorithm we call Inverse Bresenham. The resulting edges are used as constraints in a Delaunay triangulation to build the final 2.5D mesh. The geometry is transmitted to the mobile client together with the color image of the current view. See Fig. 1 for an overview of the system.

The client uses two threads, one for rendering the textured mesh using OpenGL ES, the other for image capture and face detection (or pose detection). Once a face/pose is detected, the rendering thread will receive a notification about the face position (or device pose) and the virtual camera view will change accordingly. The viewpoint will also change when the user navigates using touch gestures. With our textured 2.5D representation we can easily achieve frame-rates > 10 Hz on a consumer smart phone. If the viewpoint changes too much, this movement will be sent as a request to the server to generate and stream a new 2.5D mesh.

Using this kind of client-server architecture and a 2.5D mesh instead of a simple video stream has several advantages. First, at a lower frame rate (2 Hz) we can spend more bits on each individual frame, which results in higher visual quality. Second, whenever the connection to the server deteriorates, the user still has an immersive experience, due to the locally available geometry information. Third, and most important, we reduce the lag that the user experiences, since navigation is evaluated locally. Compared to a streamed video, where the user input has to be transmitted to the server, and a video stream has to be sent back, we can hence reduce the system latency significantly.

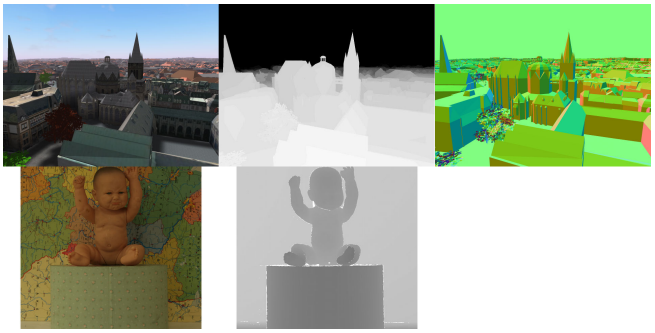


Figure 2. Input to our system. Top: synthetic images (color, depth, normals), Bottom: TOF images (color, depth) from [17]

Since we strive for real-time with a guaranteed 2 Hz update on the mesh data for the client, we have to limit the time budget for each step in our pipeline. We assume that it takes around 150ms to transmit all data to the client and incorporate the feedback from the client. Furthermore we allocate 250ms for the rendering of the framebuffers and compression of the raw image data. That leaves 100ms for the 2.5D mesh generation.

The mesh generation time budget is further divided into four parts: the edge detection (5ms), pixel sequence growing (50ms), inverse Bresenham algorithm (5ms), and constrained Delaunay triangulation (40ms).

IV. Coarse 2.5D Mesh Generation

The key aspect of our proposed method is an efficient approximation of the rendered 3D scene by a coarse, single layer depth mesh. In the following paragraphs, we will explain how our algorithm generates this mesh.

A. Rendering Server

The input to our algorithm are three framebuffers of the same image, see Fig. 2. The first is simply the color-buffer of the rendering result. The second is a linear depth buffer of the scene, which gives us information about depth discontinuities. The third framebuffer is the normal buffer, containing the surface normals per pixel in world coordinates. This buffer helps to detect feature edges of the geometry and is easy to generate for synthesized images. The input is not limited to 3D scene rendering results. Alternatively, our system can also take inputs (depth image and RGB image) from a TOF camera. For images where we only have the depth information, the normals can be approximated by computing the gradients of the smoothed depth image.

B. Image Segmentation

We use a Cut-Off distance to segment the image into foreground and background, based on the pixel depth values. There are two reasons for this. First, the user is more

interested in objects that are near, than the ones which are far away, hence it is sufficient to compute the correct parallax for nearby objects. Second, due to perspective projection the displacement of a point $P(x, y, z)$ in the world space will be hardly visible, if P is far away from the camera. By applying a symmetric frustum and viewport transformation, we project P from the world space to the window space:

$$P_{win} = \left(\frac{w}{2}\left(1 - \frac{nx}{rz}\right) \quad \frac{h}{2}\left(1 - \frac{ny}{tz}\right) \right)^T, \quad (1)$$

where w and h are the width and height of the window. n, r, t is near, right, and top clipping plane. If P moves to P' in the world space, its displacement in window space is:

$$|P_{win}P'_{win}| = (2|z|)^{-1}\sqrt{A(x'-x)^2 + B(y'-y)^2} \ , \quad (2)$$

where A equals $\frac{w^2n^2}{r^2}$ and B equals $\frac{h^2n^2}{t^2}$, which are constants. Here we assume $|z' - z|$ is much smaller than $|z|$. As described in III, we set thresholds T_x, T_y for viewpoint updating. Therefore we have $|x'-x| < T_x$ and $|y'-y| < T_y$. To make the on-screen disparity of P and P' less than 1 pixel, we only need to make $|P_{win}P'_{win}| < 1$, which equals:

$$|z| > 0.5\sqrt{AT_x^2 + BT_y^2} \ . \quad (3)$$

Our Cut-Off distance is computed using 3. In the following steps, only the foreground information is processed.

C. Edge Detection

The goal of edge detection is to find out all high frequency changes, i.e. visible edges, of the current view. We compute the second-order derivative (Laplacian operator) of the depth image, search for zero-crossing points and perform a binary thresholding θ, so that only the edges are highlighted. The result image is:

$$I_{lap}(x, y) = \text{Binary}_\theta(\nabla^2 I(x, y)) \quad (4)$$

For the normal image, we compute the normal change between neighboring pixels along horizontal and vertical direction, scale the results to the same range as that of the depth image, and store the maximum normal change for every pixel:

$$\Delta n_x = \langle n_{i,j}, n_{i+1,j} \rangle \quad (5)$$

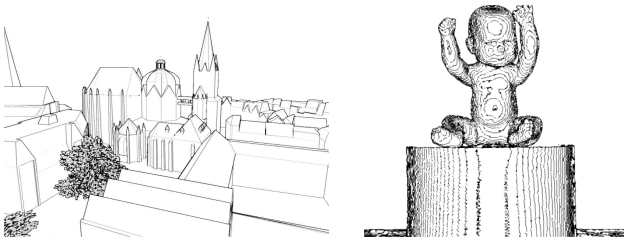
$$\Delta n_y = \langle n_{i,j}, n_{i,j+1} \rangle \quad (6)$$

$$I_{dot}(x, y) = s \cdot \max\left(1 - \Delta n_x, 1 - \Delta n_y\right) \quad (7)$$

The output of the edge detection is the combination of I_{lap} and I_{dot}, see Fig. 3(a):

$$I_{ED}(x, y) = \max(I_{lap}(x, y), I_{dot}(x, y)) \quad (8)$$

A real smooth surface has no depth discontinuity, but in a TOF camera image, the depth value quantized, which will result in small depth disparity on that surface. By lowering the threshold of the Laplacian filter, we can detect slight depth discontinuities on the surfaces of organic objects, see Fig. 3(b).



(a) Synthetic data. (b) TOF image.

Figure 3. Result of Edge Detection.

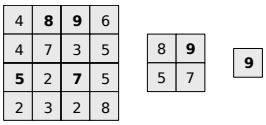


Figure 4. The maximum quad-tree stores in each inner node the maximum of its four children. Deletion updates take at most $O(\log n)$, if the update needs to be propagated to the root node. In the best case the update takes $O(1)$.

D. Monotonic Line Growing

The output of the edge detection is a grayscale image where all feature edges are highlighted. In order to generate a linked list of pixels for the line segmentation in the next step, we follow a greedy strategy to find monotonic pixel sequences in the intensity image. I.e. from the starting position the x- and y-coordinate of the pixel must monotonically increase or decrease.

We use a maximum-quad-tree to efficiently find and update the edge pixel with the highest intensity. The maximum-quad-tree is a standard quad-tree, but augmented to become a heap-like structure. The root node of the tree contains the coordinate and value of the pixel with the highest intensity (see Fig. 4). When we take a pixel out of the tree, the update propagates from the leaves to the root node. At most $O(\log n)$ changes to the tree have to be performed per pixel deletion, where n is the number of pixels in the tree. Since we use a quad-tree, no re-balancing is needed, as is the case for a traditional heap. Obviously this comes at the expense of higher memory consumption, since our quad-tree always has maximum fan-out.

The line growing procedure picks the maximum pixel in the tree, and monotonically grows the pixel sequence by continuing with the strongest edge pixel in the direct 8-neighborhood. This is done until either the image border is reached, or no more non-zero pixels can be found, or the growing direction is changed. Then the next pixel sequence is started. Pixel sequences that are shorter than a certain number of pixels are discarded. In our experiments we used ten pixels as the threshold, see Fig. 5(a),5(b).

E. Inverse Bresenham Algorithm

To segment a linked list of points into straight line segments, Lowe [22] introduced an algorithm which recursively

subdivides a curve at the point of maximum deviation. Thus he can generate a good approximation of line segments even from a noisy image. But this approach needs to compute a point-line distance for every linked point, which is expensive. To accelerate this process, we identify sub-sequences that correspond to the rasterization of a line segment by utilizing the monotonic pixel sequences obtained in the last section.

Let $\{\mathbf{p}_i\}$ be such a sequence of pixels. Due to the monotonicity of the sequence we can always flip the signs of the x and y coordinates such that only three different direction vectors $\mathbf{v}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$ can occur, namely

$$\mathbf{x} = \begin{pmatrix} 1 & 0 \end{pmatrix}^T, \mathbf{y} = \begin{pmatrix} 0 & 1 \end{pmatrix}^T, \mathbf{d} = \begin{pmatrix} 1 & 1 \end{pmatrix}^T. \quad (9)$$

We can further simplify this sequence of direction vectors by replacing each occurence of the vector $\mathbf{d}$ with the two vectors $\mathbf{x}$ and $\mathbf{y}$ which corresponds to adding one auxiliary pixel to the sequence.

Now we have to characterize sequences of direction vectors $\{\mathbf{d}_i\}$ which emerge from the rasterization of a single line segment. The defining property of such sequences is the constant slope m, which in our case has to lie in the interval $[0, \infty)$ due to monotonicity and optional sign flips. Since the slope m defines the ratio of the number of occurences of the vectors $\mathbf{x}$ and $\mathbf{y}$ we can derive the following regular expression for rasterized line segments:

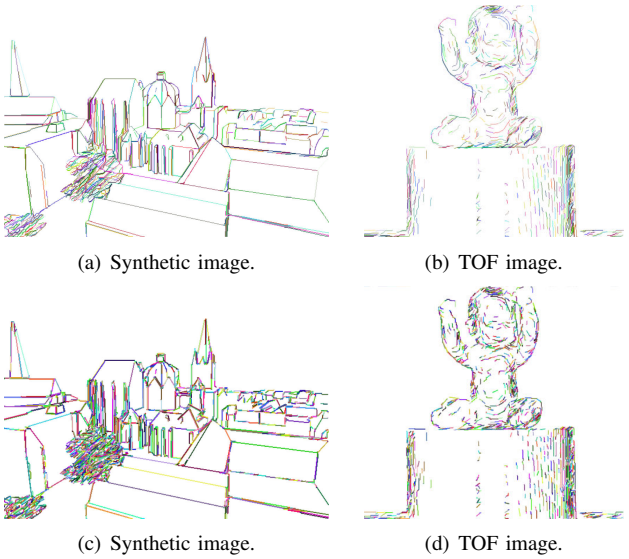
$$a\, b^{n_0}\, a\, b^{n_1}\, a \ \ldots\ a\, b^{n_j}\, a \ \ldots \quad (10)$$

where there exists an integer k such that for the integer run lengths n_j of b-repetitions it holds that $n_j \in \{k, k+1\}$. If the slope m is below 1 then in this expression the symbol a corresponds to the vector $\mathbf{y}$, b corresponds to $\mathbf{x}$ and $k \leq 1/m < k+1$. If $m > 1$, we find a corresponds to $\mathbf{x}$, b corresponds to $\mathbf{y}$ and $k \leq m < k+1$.

If the slope m of the line to be vectorized is known in advance, the above regular expression can be parsed by a simple regular automaton. However, in our case the slope is unknown a priori and hence we have to equip our automaton with a float variable m which stores the currently estimated approximate slope. The following pseudo code implements this automaton, where we assume for simplicity that the input sequence of direction vectors $\{\mathbf{d}_i\}$ has already been converted into a sequence of integer run lengths $\{n_j\}$ by counting repetitions.

```
m = n_0 ;
for j = 1 to ...
{
    if  | m - n_j | > 1  then  stop
    m = (m + n_j) / 2
}
```

The "trick" in this implementation lies in the fact that initially m is set to the integer value n_0 such that both $n_j = n_0 - 1$ and $n_j = n_0 + 1$ can pass the stopping criterion as long as the value of m does not change. Then when the first $n_j \neq n_0$ is parsed, the value of m is adjusted to a non-integer value such that from now on only two possible integer run lengths are accepted. Results are shown in Fig. 5(c),5(d).



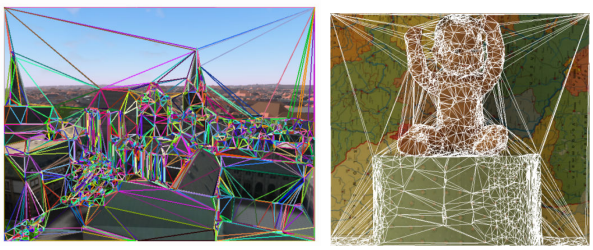
(a) Synthetic image. (b) TOF image.

(c) Synthetic image. (d) TOF image.

Figure 5. Top: Monotonic Line Growing. Bottom: Inverse Bresenham. Each color stands for a pixel sequence (or line segments).

F. Triangulation

The Inverse Bresenham step will result in a set of line segments that are now used as input to a constrained Delaunay triangulation. Vertices and edges are inserted into the mesh, the Z-coordinate being set to the corresponding value from the depth buffer image, see Fig. 6(a),6(b).



(a) Synthetic image. (b) TOF image.

Figure 6. 2D mesh resulting from the Constrained Delaunay Triangulation.

V. Immersive Display

The key point of interaction on mobile platforms is intuitiveness, which requires to map our "natural" actions from the physical world into the virtual world. On the client part of our system, besides using the touch-screen gestures, we further augment the interaction by using the front-facing camera or the gyroscope of the device. The spatial relation between the user's face and the device is detected and then

the view frustum is updated accordingly. This gives the illusion of immersiveness, i.e. as if the screen were a window into the 3D world, creating a realistic illusion of depth and space. This is inspired by Lee [15], who used the infrared camera in the Wii remote and a head mounted sensor bar to track the location of the user's head and render view dependent images on the screen. Lee's method is suitable for desktop display and free body movement several meters away from the infrared camera, while our approach fits the mobile use case better. Since the physical size of the mobile screen is small and we always hold it in our hand to manipulate, it is more natural to tilt the phone than to move our head around to look into the virtual space.

Comparing face detection and gyroscope, they can both achieve a virtual window effect in our use case. But using face detection, both the device and the face can be moved, and this will assure that the illusion works. In addition, using face detection we can roughly compute the distance between the camera and the face. This parameter can be used to modify the view frustum, so that when we get closer to the screen, the user has a wider field of view. This is consistent to the viewing experience in the real world when walking towards a window. Since the gyroscope sensor is provided by the device, here we only explain the face detection.

A. Face Detection

We use the face detection API of OpenCV, which is the implementation of Viola-Jones object detection algorithm [23]. The input is the captured frame from the integrated camera. Since the detection processes frames individually, no inter-frame information is considered. After optimization, the detection rate is around 10 FPS.

In order to augment the virtual window effect, we compute the distance between the mobile screen and the user's face by using the detected face scale. We preset a reference distance and the corresponding face size. By a simple reciprocal mapping we can get the current screen-face distance, which is then used to set the near clipping plane.

B. Interaction

We propose two different modes for interaction: One is "tour guide", and the other one is "free walk".

In the "tour guide" mode, we predefine a camera path flying over the virtual scene, so the trajectory of the virtual camera is fixed. With a rate of two updates per second the server streams the textured mesh of the current view to the client for display. Since the camera trajectory is known, the client can interpolate viewing positions and render the 2.5D mesh at a framerate > 10Hz.

In the "free walk" mode, the user is allowed to explore the scene freely. The user can change his position by touch gestures like panning and pinching. The viewing direction is also changed by face detection. If the user moves or rotates the view by more than a certain threshold, the server will generate a new 2.5D mesh and send it to the client.

VI. Results

We implemented the system in C++. The server runs on a desktop PC with a Dual-Core 2.0 GHz CPU and 4 GiB memory. In the current version, the coarse mesh generation runs mostly on the CPU, with the edge detection being implemented in GLSL on the GPU. The client device is an Apple iPod Touch 4G at 1 GHz with 256 MiB RAM. To communicate with the server, we use a Wi-Fi connection.

For the depth and normal buffer, we use a resolution 640×480 pixels. Each view rendered on the server contained 0.5 to 0.8 million triangles, and there were more than 1.6 million triangles in the whole scene. The render server uses multiple render passes to achieve many special effects, including normal mapping, dynamic shadow maps, interior mapping, and much more. This would not be possible to render efficiently on the mobile device. For the TOF camera input, if the image resolution is higher than 640×480, we scale the width down to 640 and keep the original ratio.

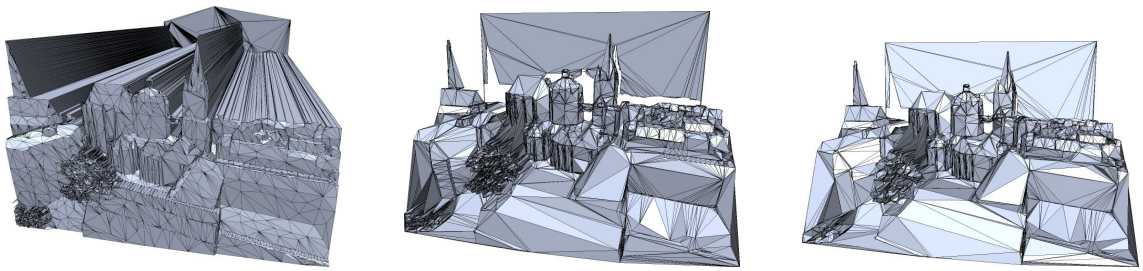
Table I shows the time consumption for each processing step of our Coarse 2.5D Mesh Generation on a city scenery view, see Fig. 7(b). The total time we need to build the mesh is around 100 ms. The processing time is related to the complexity of the scene. For single objects scene it will be faster, see Table III.

Fig. 7(b) shows one of our depth meshes containing 5,000 vertices and 9,700 faces, which would result in 105 KiB mesh data to transmit (vertices, texture coordinates and indices, each value is stored in a ushort or half precision float). Since on a mobile display small details are hardly visible, a sparse mesh suffices. To further reduce complexity we add a threshold to the pixel-sequences of the monotonic line growing algorithm. Hence we would first consider the longer pixel-sequences, which are more likely to represent relevant parts of a contour which separates different planar regions. In our experiment, we found out that when using 50% of the pixel-sequences there were no visible artifacts, and the mesh contained about 3,300 vertices and 6,400 faces (70 KiB), see Fig. 7(c). Together with the mesh data the server needs to send the color buffer of the current view to the client. The textures have a resolution 512×512 and are stored in JPEG format. The image file size is around 40 KiB each. Therefore per textured mesh about 110 KiB are streamed from the server to the client every half second or when the server receives the update request.

We compare our depth mesh to one generated by decimating a dense mesh which represented the contents of the depth buffer (i.e. each pixel in the depth image is treated as a vertex), using an optimized implementation of a standard decimation technique [24]. For the results compare Fig. 7(a) and 7(b). Our algorithm only places vertices on the feature edges, thus producing a more sparse representation of the scene, and is two orders of magnitude faster, see Table II. Our approach works for rendered polygons and TOF camera

Table I
TIME CONSUMPTION FOR DIFFERENT PROCESSING STEPS.

Step	Laplace	Δ Normal	Merge Buffer	MaxTree	Line Growing	Inv. Bresenham	CDT	Sum
Timing(ms)	< 1	< 1	< 1	22	40	2	42	106



(a) A decimated depth mesh

(b) Coarse 2.5D mesh, 75% pixel sequences.

(c) Coarse 2.5D mesh, 50% pixel sequences.

Figure 7. (a) A raw depth mesh (640 by 480. each pixel in the depth image is treated as a vertex) is decimated to 10,000 faces, by a standard mesh decimation algorithm using error quadrics. (b) Our mesh, 9,720 faces. (c) Our mesh, 6,423 faces.

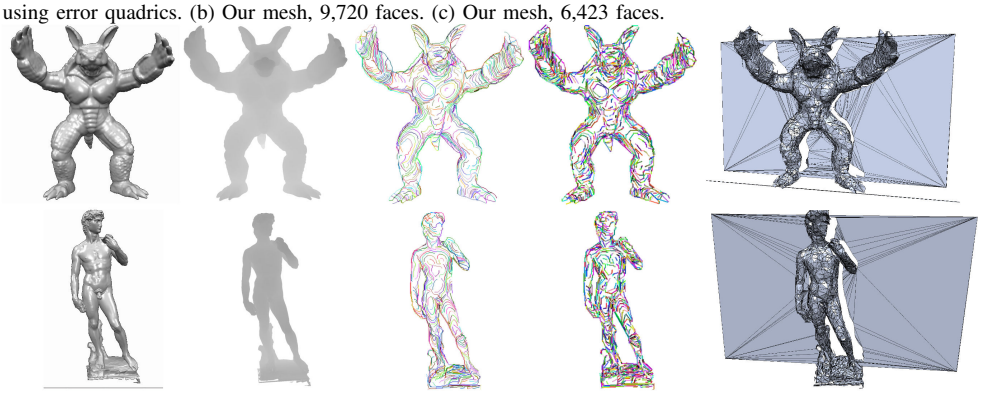


Figure 8. Examples of our 2.5D mesh generation of 3D models. From left to right: rendered polygon model, depth-image, monotonic line growing, inverse bresenham, 2.5D mesh.

Table II
TIME CONSUMPTION FOR MESH GENERATION

Mesh	#Vertex	#Face	Time(sec)
Decimated Mesh	6732	10000	10
Our Mesh 75%	5000	9720	0.10
Our Mesh 50%	3343	6423	0.07

Table III
TIME CONSUMPTION OF DIFFERENT MODELS

Input Data	#Faces Model	#Faces 2.5D	Time(ms)
Model Armadillo	100k	8,951	60
Model David	500k	5,549	47
Kinect Sculpture	-	4,420	56
TOF camera Baby	-	8,082	85

images, see Fig. 8, 9. On the surface of organic objects, by correct thresholding, depth discontinuities are detected as feature edges which can then be used to build the 2.5D mesh. The time consumption for complex scenes is higher than single object due to the increasing amount of feature edges, but it still follows our time budget, see Table III.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we presented a client-server architecture for displaying complex 3D scenes on mobile devices. On the server side, we introduced a highly efficient algorithm to decompose the geometry represented by the current view's depth profile into a small set of textured polygons. The textured mesh is then streamed to the client for rendering. To augment the viewing experience on the mobile platform, we utilized Viola-Jones face detection algorithm to detect the face position. Based on that the view frustum is changed accordingly, so that the user can look into the 2.5D scene through a virtual window. When the face movement is higher than a threshold, the view position is updated to the server to request a new textured mesh.

Currently we use Viola-Jones face detection which is sensitive to light condition, and the detected face positions are not very stable. The future work could be to find a more sophisticated algorithm, e.g. considering optical flow. Second, after foreground/background segmentation there are holes in the background. We could blend meshes of multiple reference frames to avoid disocclusion artifacts. In addition, in our mesh generation pipeline, Monotonic Line Growing is the main bottleneck, such that implementing this step on the GPU would accelerate the conversion even more.

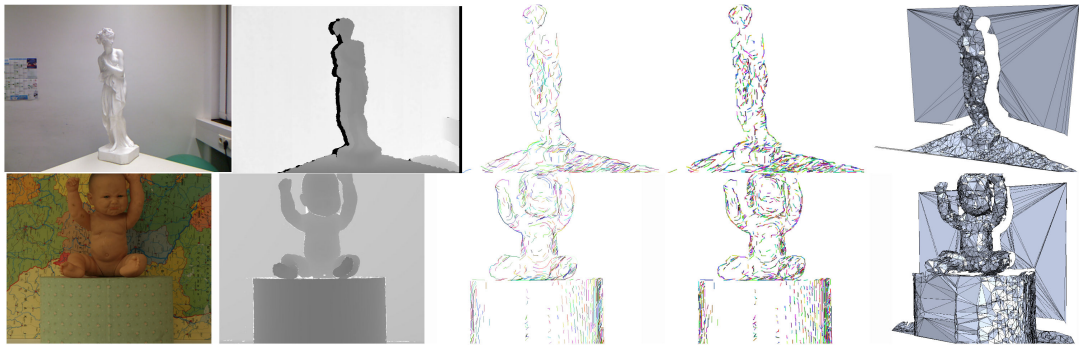


Figure 9. Examples of our 2.5D mesh generation using TOF and structured light camera inputs. Left to right: RGB image, depth-image, Monotonic Line Growing, Inverse Bresenham, 2.5D mesh. Top: images using Kinect, Bottom: dataset of [17].

ACKNOWLEDGMENT

This work was supported in part by NRW State within the B-IT Research School and the UMIC Research Centre, RWTH Aachen.

REFERENCES

[1] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte, "A developer's guide to silhouette algorithms for polygonal models," *IEEE Comput. Graph. Appl.*, vol. 23, pp. 28–37, July 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=858619.858656

[2] R. von Gioi, J. Jakubowicz, J.-M. Morel, and G. Randall, "Lsd: A fast line segment detector with a false detection control," *PAMI*, vol. 32, no. 4, pp. 722 –732, 2010.

[3] C. Gotsman, S. Gumhold, and L. Kobbelt, "Simplification and compression of 3d meshes," in *In Proceedings of the European Summer School on Principles of Multiresolution in Geometric Modelling (PRIMUS*. Springer, 1998, pp. 319–361.

[4] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Levy, *Polygon Mesh Processing*. AK Peters, 2010.

[5] J. Wu and L. Kobbelt, "Fast mesh decimation by multiple-choice techniques," in *VMV*, 2002, pp. 241–248.

[6] K. Pulli and M. Pietikäinen, "Range image segmentation based on decomposition of surface normals," in *SCIA*, 1993.

[7] R. Pajarola, M. Sainz, and Y. Meng, "Dmesh: Fast depth-image meshing and warping," *Int. J. Image Graphics*, vol. 4, no. 4, pp. 653–681, 2004.

[8] R. Pajarola, "Large scale terrain visualization using the restricted quadtree triangulation," in *VIS*, 1998. [Online]. Available: http://portal.acm.org/citation.cfm?id=288216.288219

[9] W. R. Mark, L. McMillan, and G. Bishop, "Post-rendering 3d warping," in *I3D*, 1997, pp. 7–16. [Online]. Available: http://doi.acm.org/10.1145/253284.253292

[10] F. Sillion, G. Drettakis, and B. Bodelet, "Efficient impostor manipulation for real-time visualization of urban scenery," vol. 16, no. 3, pp. 207–218, 1997. [Online]. Available: http://artis.imag.fr/Publications/1997/SDB97

[11] X. Décoret, F. X. Sillion, G. Schaufler, and J. Dorsey, "Multi-layered impostors for accelerated rendering," *Comput. Graph. Forum*, vol. 18, no. 3, pp. 61–73, 1999.

[12] G. Schaufler, W. Stürzlinger, J. Kepler, U. Linz, and A-Linz, "A three dimensional image cache for virtual reality," *Comput. Graph. Forum*, vol. 15, no. 3, pp. 227–236, 1996.

[13] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder, "Hierarchical image caching for accelerated walkthroughs of complex environments," in *SIGGRAPH*, 1996, pp. 75–82. [Online]. Available: http://doi.acm.org/10.1145/237170.237209

[14] J. Lluch, R. Gaitán, E. Camahort, and R. Vivó, "Interactive three-dimensional rendering on mobile computer devices," in *ACE*, 2005, pp. 254–257.

[15] J. C. Lee, "Hacking the Nintendo Wii Remote," *IEEE Pervasive Computing*, vol. 7, pp. 39–45, 2008.

[16] S. B. Gokturk, H. Yalcin, and C. Bamji, "A time-of-flight depth sensor - system description, issues and solutions," in *CVPRW'04*. IEEE Computer Society, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1032634.1032926

[17] H. Hirschmuller and D. Scharstein, "Evaluation of cost functions for stereo matching," in *CVPR*, 2007.

[18] Microsoft, "Kinect," Dec 2010, http://www.xbox.com/en-US/kinect.

[19] PrimeSense, "Light coding," Dec 2010, http://www.primesense.com/.

[20] T. O. Project, "Openkinect," Dec 2010, https://github.com/OpenKinect/libfreenect/.

[21] T. O. Organization, "Openni," Dec 2010, http://www.openni.org/.

[22] D. G. Lowe, "Three-dimensional object recognition from single two-dimensional images," *Artificial Intelligence*, vol. 31, pp. 355–395, 1987.

[23] P. Viola and M. Jones, "Robust real-time object detection," *IJCV*, vol. 57, pp. 137–154, 2004.

[24] M. Garland and P. S. Heckbert, "Surface simplification using quadric error metrics," in *SIGGRAPH '97*, 1997.