The Randomized Sample Tree: A Data Structure for Interactive Walkthroughs in Externally Stored Virtual Environments

Jan Klein* Jens Krokowski* Matthias Fischer* Michael Wand† Rolf Wanka*
Friedhelm Meyer auf der Heide*

*University of Paderborn †University of Tübingen

ABSTRACT

We present a new data structure for rendering highly complex virtual environments of arbitrary topology. The special feature of our approach is that it allows an interactive navigation in very large scenes (30 GB/400 million polygons in our benchmark scenes) that cannot be stored in main memory, but only on a local or remote hard disk. Furthermore, it allows interactive rendering of substantially more complex scenes by instantiating objects.

For the computation of an approximate image of the scene, a sampling technique is used. In the preprocessing, a so-called sample tree is built whose nodes contain randomly selected polygons from the scene. This tree only uses space that is linear in the number of polygons. In order to produce an image of the scene, the tree is traversed and polygons stored in the visited nodes are rendered. During the interactive walkthrough, parts of the sample tree are loaded from local or remote hard disk.

We implemented our algorithm in a prototypical walkthrough system. Analysis and experiments show that the quality of our images is comparable to images computed by the conventional z-buffer algorithm regardless of the scene topology.

Categories and Subject Descriptors

I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Display Algorithms*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Graphics data structures and data types*; G.3 [**Mathematics of Computing**]: Probability and Statistics—*Probabilistic algorithms*

Keywords

Rendering Systems, Level of Detail Algorithms, Spatial Data Structures, Point Sample Rendering, Out-Of-Core Rendering, Monte Carlo Techniques

*Dept. of Mathematics & Computer Science and Heinz Nixdorf Institute, University of Paderborn, 33095 Paderborn, Germany. Email:{janklein, kroko, mafi, wanka, fmadh}@uni-paderborn.de
†University of Tübingen, WSI/GRIS, Sand 14, 72076 Tübingen, Germany. Email: wand@gris.uni-tuebingen.de

VRST'02, November 11–13, 2002, Hong Kong.

1. INTRODUCTION

Despite the use of advanced graphics hardware, real-time navigation in highly complex virtual environments is still a fundamental problem in computer graphics because the demands on image quality and details increase exceptionally fast. There are several sophisticated algorithms to overcome this problem but scenes of arbitrary topology that cannot be stored in main memory are still very difficult to handle.

A very interesting application field for our proposed approach is navigation in virtual environments consisting of many different detailed objects, e.g., of CAD data that cannot all be stored in main memory but only on hard disk. E. g., such a navigation would be very useful in the area of plant layout. On the one hand, one wants to get an overview of all models by viewing from far away, and on the other hand one wants to examine single objects very closely. Therefore, it is desirable to store only the polygons and not to produce additional data as, e. g., textures or prefiltered points: Then, the viewer can operate with the original data and images of high quality can be rendered independently of the scene topology and the camera position. Furthermore, the polygons of such highly complex scenes require a lot of hard disk space so that the additional data could exceed the available capacities. For these requirements, an appropriate data structure – the randomized sample tree – has to be employed with a memory consumption that is linear in the number of polygons. Our work is motivated by the following two main requirements:

1. Interactive rendering of complex scenes: We want to achieve interactive rendering of complex scenes in output-sensitive time, i.e., the rendering time should be sublinear in the number of polygons. To achieve this, we reduce the scene complexity by only rendering samples of polygons that approximate the scene. Our measurements show that the rendered samples of polygons are small enough to generate interactive frame rates and large enough to generate images of a good quality. We assume that the samples of polygons needed for the computation of one frame can be stored in main memory. In practice, this is no real restriction because the samples are very small even for our most complex scenes.

2. Complex scenes stored on hard disk: We focus on navigation through highly complex scenes that cannot be completely stored in main memory. The number of polygons in a scene should only be restricted by the size of the hard disk. This is of special interest for scenes consisting of many different types of models for which instantiation schemes [32] cannot be used. Generally, storing the data causes expensive hard disk accesses which have to be minimized in order to achieve interactive rendering.

We solve this as follows: The samples are computed in the preprocessing and are stored in a sample tree on hard disk. During navigation, needed nodes and their polygon data must be loaded.

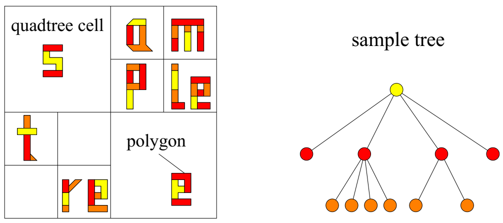

Figure 1: Sample tree for a 2D scene with a quadtree. (This figure is reproduced in color on page 000.)

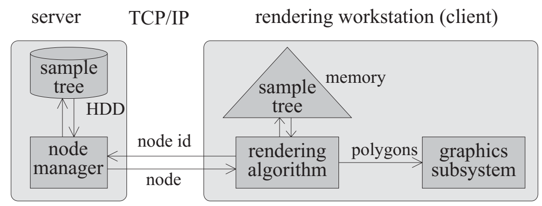

Figure 2: Walkthrough system: client and server connected via TCP/IP, the polygon data are stored on hard disk.

To exploit temporal coherence, we use a caching mechanism so that only very few nodes have to be loaded from one frame to the next (we assume that the viewpoint does not move over great distances within a short period of time). Furthermore, the samples that are stored in the nodes are small so that there is little polygon data to load from hard disk for each image. In the following, we give a short overview of further aspects and features of our algorithm.

Network-based rendering: Based on the idea that the local hard disk is too small to store a highly complex scene, we implemented our algorithm in a client-server network: The sample tree is stored on a remote server connected to the client via TCP/IP. During a walkthrough the client loads parts of the polygon data from the remote hard disk into its main memory and sends them to its graphics subsystem. In Section 7, we show that for networks with a high bandwidth, good frame rates can be achieved.

Arbitrary scene topology: The quality of the images computed by our algorithm is independent of the scene topology and the camera position. Holes or disconnectivities in the models cause no problems. The input for our algorithm is the polygonal description of the virtual environment, e.g., triangles, normals, material colors. A typical scene for our approach may consist of many different models (e.g., landscapes with trees, buildings and cars).

Efficiency: We need neither complicated level-of-detail computation of the polygons nor memory-expensive replacement by textures or prefiltered points. The space for storing our sample tree is only linear in the number of polygons: Every polygon is stored exactly once in the tree and in practice – our measurements confirm this –, the number of nodes is at most the number of all n polygons because every node contains at least one polygon. Therefore, storing the tree costs only $O(n)$ space. This is a great advantage because our scenes are highly complex and have to be stored on hard disk. The test results with our prototypical walkthrough system show that our approach works with scenes consisting of more than 400 million polygons, whereby every polygon is separately stored on hard disk. Our results show that an approximate image with 640×480 pixels can be computed within 326 msec if the sample tree is stored on a remote hard disk.

2. OUTLINE OF OUR APPROACH

The sample tree: The rendering algorithm uses a so-called sample tree which is an octree-like data structure for the spatial organization [4] of the polygons. In every node a sample of polygons is stored that approximates the corresponding cell if the viewpoint is *far away* from the cell. An exact definition of *far away* is given in Section 4. The leaf nodes store all polygons that were not chosen for the samples in any ancestor nodes.

Figure 1 demonstrates the idea of our sample tree. In order to pick a sample for the root node of the sample tree, our tree construction algorithm chooses polygons from the whole scene depending on their areas and stores them in the root node. These polygons are marked in yellow in our example. They give a good approximation of the scene if the viewpoint is *far away* from the cell; in this case the viewpoint would lie far outside of the whole scene. In order to get a sample for the upper left quadtree cell, we randomly choose polygons from the corresponding cell, the probabilities depending on their areas (we do not consider polygons that are already stored in the root node). The chosen polygons, marked in red, lying in the cell together with the yellow polygons stored in the ancestor node give a good approximation of the upper left quadtree cell if the viewpoint is *far away* from it.

Rendering of the scene: In order to produce an image of the scene, the sample tree is traversed by a depth-first search and all polygons lying in visited nodes are rendered. The traversal stops at a node u if the corresponding cell is *far away* from the viewpoint. Then the distance from the viewpoint to the cell of u is so large that the sample stored in u gives a good approximation of the cell. If the traversal stops at a leaf node v, all polygons lying in the corresponding cell are rendered. If, in this case, we rendered only a sample of polygons, the approximation would result in low image quality because the viewpoint is close to the cell. This kind of traversal is well-known in the context of LOD systems (e. g., see [18]).

Figure 12 shows how our algorithm works. We zoomed into two parts of an image of a landscape scene rendered by our method. One can see that the objects are incomplete and that only samples of polygons are rendered. Below the two zoomed pictures, the corresponding complete parts of the original scene rendered by the z-buffer algorithm are presented for comparison. Observe that for nodes lying in the part closer to the viewpoint the tree traversal is aborted later than for nodes in the part far away. One can see, e. g., that more polygons of a house are rendered in the front part than for houses lying in the other part.

Client-server walkthrough system: For highly complex scenes, the sample tree can only be stored on a local or remote hard disk if the scene does not fit into main memory. The walkthrough system consists of a rendering workstation (client) and a node manager (server) that handles the sample tree stored on its local disk (see Figure 2). The rendering algorithm (client) traverses the sample tree during the walkthrough and sends the polygons stored in the visited nodes to the graphics subsystem. If a node of the sample tree is not available in main memory, the rendering workstation loads it via TCP/IP from the node manager.

We produce images within a very short time because we load only a few nodes for one frame. Furthermore, the samples of polygons stored in the nodes are very small. This client-server concept is similar to the concept of a scene manager used in [25].

3. RELATED WORK

Several approaches have been proposed for accelerating the rendering of complex 3D scenes. In the following, we summarize related work in the area of point sampling, out-of-core rendering and network-based walkthroughs. Furthermore, we look at the possibility of adapting occlusion culling to our approach.

Point sample rendering: Our rendering technique has the most in common with point sample rendering. Like these methods, we draw samples of the scene objects and render them in order to reduce time-expensive rendering of all polygons.

In the 1970's and 1980's several researchers had already proposed to use points as display primitives [2, 16] because their rendering and rasterization is simple and cheap. Additionally, point sample rendering is independent of the scene topology in contrast to mesh-based multi-resolution models. Recently, research has focused again on point sampling. Grossman and Dally [13] use points both as rendering and as modelling primitives. QSplat [22] and Surfels [21] apply prefiltering methods to convert graphics primitives into a uniform point representation. The points are selected from a LOD-like representation according to their screen projection. This results in a low rendering time and high image quality.

QSplat [22] works best for large, dense models containing relatively regular, uniformly spaced points. The problems are sharp corners and subtle curves. A further problem occurs if objects are viewed very closely and if points are projected to more than one pixel, it results in blocky images. A solution to overcome this problem is the interpolation between adjacent points which sometimes results in artifacts [21]. A further solution for the problem of close views is given by [1, 27, 31], who use locally adapted sample densities for the rendering of complex geometry. Chen and Nguyen [3] use triangles and points for the rendering of large mesh models with the ability of close views.

In contrast, our approach generally uses polygons for rendering. Points or splats can additionally be used in order to get coarser approximations and to improve the rendering times. There are some advantages of mainly using polygons: First of all we produce no additional data like, e.g., textures so that we have a memory consumption that is linear in the number of polygons. This cannot always be fulfilled by point-based methods. Moreover, the viewer can operate with the original polygon data which could be useful for some applications, for example, in the area of plant layout. Another point is that our method is able to render images of high quality independent of the scene topology and the camera position and therefore we have no problems with close views.

Wand et al. [31] proposed a technique for randomized point sampling. For each frame they have to choose new sample points by using hierarchical concatenated distribution lists. In combination with a caching mechanism they are able to render complex scenes at interactive frame rates. But their technique can only handle scenes which can be completely stored in main memory. Furthermore, their method produces images of bad quality if some polygons are parallel and lie close to each other. Our proposed approach overcomes these two drawbacks: In Section 4 we prove that images of good quality are produced by our algorithm whereby no assumption about the position of the polygons is made. The other problem – the limitation to main memory – is solved by our sample tree: We have developed a sampling technique that can be done in the preprocessing whereby the polygons can easily be stored on hard disk. Because we do not use any distribution lists like Wand et al. [31], where expensive dynamic updates have to be done for each frame, our randomized sampling method saves enough time that can be used for the data transfer from hard disk. So in contrast to Wand et al. we are able to handle scenes consisting of many different models that cannot all be stored in main memory but only on hard disk. Note that although we sample only once in the preprocessing instead for each frame we can prove that our images are always of good quality.

Out-of-core rendering: Recently, Lindstrom and Pascucci [17] present a framework for out-of-core visualization of terrain surfaces. Their results show that this approach works well with large terrain meshes consisting of about 5 GB polygon data but it does not really fit to scenes of arbitrary topology as our approach does. The system presented by Sobierajski Avila and Schroeder [26] allows an interactive navigation through large polygonal environments produced by CAD systems. Thereby, they combine view frustum culling and LOD modelling. In comparison, our method does not use a limited number of different LOD representations but every polygon is only stored once in the sample tree. Furthermore, our approach adjusts the approximation quality within a single object during the navigation. So we can achieve a smooth navigation without the effect of toggling between different LOD representations.

Wald et al. [30, 29] propose distributed ray tracing as a technique for interactive rendering. In [29] it is shown that they are able to generate very high quality images of scenes consisting of 50 million polygons stored on hard disk. They achieve interactive frame rates by using seven dual Pentium PCs. It is obvious that ray tracing produces images of higher quality than conventional rendering methods but generally ray tracing has a worse performance than rasterization algorithms.

Network-based walkthroughs: Transmitting 3D data across a network has become an interesting field of research. As our approach allows not only rendering of highly complex scenes stored on a local hard disk but also stored on a remote hard disk, we survey shortly related work in the area of network-based rendering.

In the context of occlusion-culling, the PVS approach presented by Teller and Séquin [28] is suitable for a client-server system. Thereby, the server computes the PVS (Potentially Visible Sets) and sends them to the client to avoid the transmission of invisible geometry [10, 11].

In order to use the network capacity consistently despite of sharply changing viewpoints, Cohen-Or and Zadicario [6] propose a prefetching of an ε-neighborhood to ensure that off-screen objects are loaded into memory. Mann and Cohen-Or [19] present another system of collaborating client and server. They assume that web-based virtual environments are dominated by huge amounts of texture data which are stored on the server, while the models are locally available at the client. The network requirements are reduced by sending only the difference image needed for the current view.

Schmalstieg and Gervautz [24] employ the notion of “Area Of Interest” (AOI) in the context of multi-user distributed virtual environments where AOI is a circular area around the viewpoint. Only the objects in this area are sent from the server and stored in main memory of the client in order to reduce the network transmission. In contrast to their approach, we also render objects that are far-off so that we do not have the effect of objects popping into an image.

Streaming transmission of data sets has become interesting because of growing network speed. Sophisticated algorithms transmit low-resolution data first, so the user sees a coarse approximation and can interact with the scene. Then, a progressive stream of high-resolution data improves the quality. There are some polygon-based approaches that are suitable for this kind of multi-resolution rendering: Hoppe's progressive mesh approach [15] represents a model by a base mesh and refinements to this mesh. The mesh correction can take place by vertex split operations [15] or by wavelet encoded operations [8].

Rusinkiewicz and Levoy [23] employ a multi-resolution point rendering system for the streaming of large scenes. A splat-based bounding box hierarchy is used for a view-dependent progressive transmission of data across a network of limited bandwidth.

Occlusion culling: In the last few years many occlusion culling algorithms have been developed, see the survey of Cohen-Or et al. [5] for a detailed overview. We have not focused on any occlusion culling yet, because point sample rendering and occlusion culling are two orthogonal problems. We only perform view frustum culling and backface culling. The running time of our approach depends linearly on the projected area a of s sampled polygons, including hidden polygons. Since we use the z-buffer algorithms for rendering the sampled polygons, the running time is $O(s+a)$ [14]. Thus, our algorithm becomes inefficient if there are many polygons near to the viewer because these polygons have a great projected area. But we think, our sample tree fits well to known occlusion culling algorithms [9, 12] because the polygons are managed in a spatial and hierarchical way that can be used for the visibility tests.

4. THE RANDOMIZED SAMPLE TREE

In this section, we describe the idea and the structure of the sample tree and how an image of the scene is rendered. Furthermore, we show how the samples of polygons have to be chosen so that, with arbitrarily high probability, a correct image is produced. In the last two subsections, we describe extensions of our algorithm to improve the image quality and to accelerate the rendering time.

In order to explain the main idea of our sample tree, we need the notion of *correct image*.

Correct image: A part of an image with size ≤ 1 pixel is called correct if at least one visible polygon r from this part is rendered. The whole image is correct if all of its parts are correct. A discussion of this definition follows at the end of section 4.3.

A *correct image* may still show aliasing and noise artifacts. Wand et al. [31] deal with these problems by averaging over several independent renderings resulting in a slow rendering time. This technique is not possible in our case as we employ fixed, precomputed sample sets. However, the same effect can be achieved by supersampling, i.e., rendering the image at a higher resolution and sampling density and then downsampling the image.

4.1 The Data Structure

In the following, we explain the hierarchical structure of our sample tree, which corresponds to a modified octree, and the insertion of the polygons into its nodes.

The octree cell of a node u is called $B(u)$. Let $A(r)$ denote the area of the polygon r. Furthermore, let $P(u)$ denote all polygons of a scene lying in $B(u)$ and let $A(P(u))$ denote the sum of all areas of polygons from $P(u)$, i.e., $A(P(u)) = \sum_{r\in P(u)} A(r)$. The idea of our sample tree is that in every inner node u, a sample of the polygons $P(u)$ is stored. We store sufficiently many polygons in a node u, so that these polygons, together with the polygons stored in the ancestor nodes of u, *approximate well* $P(u)$ if the viewpoint is *far away* from $B(u)$. *Approximate well* means that the projection of the polygons produces a correct part of the image with high probability p. The probability p depends on the sample size m_u, that, in turn, depends on $A(P(u))$. For details, see Section 4.3. To pick a sample stored in a node u we look at all polygons $P(u)$ and add a polygon r to the sample set with a probability proportional to its area, i.e., with probability $A(r)/A(P(u))$, because on average larger polygons contribute more to an image than smaller ones. *Far away* means that the projected bounding box of $B(u)$ is at most one pixel. So, in every inner node u, a sample of polygons is stored that approximates $P(u)$. A leaf node v contains all polygons from $P(v)$ except those polygons that are already stored in any direct or indirect ancestor node of v. Analogously, a polygon is not stored in any inner node u if it is already stored in an ancestor node of u. That means that every polygon is stored exactly once in the tree.

Figure 3 presents the algorithm that computes a sample tree. Before starting it, an octree has to be constructed where polygons are inserted in leaf nodes or recursively in the ancestor nodes if they do not fit completely into a bounding sphere of a leaf node. The algorithm has to be started with the root node of the octree. The sample size m_u can be computed from Theorem 1 (Section 4.3).

One has to consider that many other very small polygons would be inserted into high-level nodes (nodes near to the root node), if they lie on edges of any octree cells. As a consequence, these polygons would be rendered for nearly every image. To avoid this behavior, the algorithm can easily be modified: Instead of choosing polygons from $P(u)$ we choose them from all polygons lying in the bounding sphere of $B(u)$.

```
createSample ( node u )
    if u is not a leaf node
        for every child d of u do createSample ( d )
        M = ∅
        compute m_u
        for i = 1 to m_u do
            choose polygon r from P(u) with probability
                A(r)/A(P(u))
            store the chosen polygon in set M
            remove the polygon from any child node of u
        store all polygons from M in node u
```

Figure 3: The algorithm constructs a sample tree in bottom-up manner. Its input is the root of a precomputed octree.

4.2 Rendering an Image

In order to render an image, the sample tree is traversed by a depth-first search and all polygons stored in visited nodes are rendered. The traversal is stopped at a node u if the projected bounding box of $B(u)$ is at most one pixel. Then the polygons stored in u are rendered and produce a correct part of the image with arbitrarily high probability that depends on the sample size m_u. If the traversal is not aborted, the search ends at a leaf node v. Consequently, all polygons of the scene lying in $B(v)$ are rendered.

Note that all polygons stored in visited nodes have to be rendered. It is not sufficient to render only polygons stored in nodes where the traversal stops since $P(u)$ is only approximated by the polygons in u and the ancestor nodes together.

Our method renders all polygons having a projected area of more than one pixel because the traversal stops at a node u if its projection is at most one pixel, and polygons that do not fit completely into the bounding sphere of $B(u)$ are stored in ancestor nodes.

4.3 Sample Size

In this section, we explain how many polygons have to be chosen for each sample. Furthermore, we prove that enough polygons are stored in every node to produce correct images with arbitrarily high probability. We choose polygons randomly with a probability proportional to their areas. That means we add a polygon r to the sample with probability $A(r)/A(P(u))$ because on average larger polygons contribute more to an image than smaller ones. It is obvious that the larger the sum of all polygon areas in a cell is, the larger the sample should be in order to compute a correct image. It is important to properly estimate the size of the sample, as too many polygons result in slow rendering and too few polygons result in a low image quality. Therefore, Theorem 1 shows how many polygons have to be stored in a node u.

Theorem 1 (Sample size): Let $A(P(u))$ denote the sum of all areas of polygons from $P(u)$, and let $A(B(u))$ denote the area of one side face of the bounding box of $B(u)$. The factor q_u is defined as $q_u = \lceil A(P(u))/A(B(u)) \rceil$. Let c be an arbitrary constant. If $m_u = q_u \cdot \ln q_u + c \cdot q_u$ independent drawings from $P(u)$ are done and the chosen polygons are stored in the node u, then with probability $p \approx e^{-e^{-c}}$ their projection gives a correct part in the image for an arbitrary viewpoint if the projection of $B(u)$ is at most one pixel and if the area of the projected polygons from $P(u)$ shows no holes from any viewpoint.

Proof: We have to guarantee with probability p that from an arbitrary viewpoint, there is always a visible polygon in the sample stored in the node u. Then, as the projection of $B(u)$ is at most one pixel and as we render a visible polygon with probability p, the corresponding part of the image is correct (with probability p). Note that the following described operations neither have to be done during the preprocessing phase nor during the navigation. They are only used for proving the theorem.

Let V_u denote the viewing ray through the center of $B(u)$. We consider an arbitrary, but fixed viewpoint in the scene so that the projection of $B(u)$ is at most one pixel. Let f_{max} be the maximum area of a surface that is orthogonal to the viewing ray V_u and that completely lies in $B(u)$ (see Figure 4 (a)). The minimal sum of the area of all visible polygons from $P(u)$ amounts to f_{max} because we assume that from any viewpoint the area of the projected polygons from $B(u)$ shows no holes. If some visible polygons are not orthogonal to the viewing ray V_u, the area of all visible polygons or parts of polygons will be greater than f_{max}.

For the sake of argument, let us partition the polygons from $P(u)$ into $\widetilde{q_u}$ groups so that all polygons of each group (except the last one) have a total area of f_{max}. For this, polygons might have to be subdivided and to be put into two groups. $\widetilde{q_u}$ can be computed as $\widetilde{q_u} = \lceil A(P(u))/f_{max} \rceil$. All visible polygons or parts of visible polygons lying in $B(u)$ are put into one group. They fill at least one group because they have a minimum area of f_{max} as described above.

Now the question is how many polygons m_u have to be chosen in such a way that with probability p from each group at least one polygon is taken. By choosing at least one polygon from each group, you would have chosen at least one polygon from the group of the visible ones. Consequently, if m_u polygons are stored in u, a visible polygon from $P(u)$ is between these m_u polygons, seen from an arbitrary but fixed viewpoint. When all m_u polygons are rendered, the visible polygon is rendered too. As $B(u)$ is at most one pixel, the corresponding part of the image is correct.

The question raised above can be answered with the help of the reduction to a simple urn model: Given a bins, how many balls have to be thrown randomly and independently with the same probability into the bins so that every bin gets at least one ball? The number a of the bins corresponds to the number $\widetilde{q_u}$ of the groups and the number of balls corresponds to the number of necessary drawings of polygons. Let X denote the number of drawings required to put at least one ball into each bin. It is well known (e.g., see [20, p. 57f]) that the expectation value of X is $\widetilde{q_u} \cdot H_{\widetilde{q_u}}$ where $H_{\widetilde{q_u}}$ is the $\widetilde{q_u}$-th harmonic number.

Let c be an arbitrary constant. The $\widetilde{q_u}$-th harmonic number is about $\ln \widetilde{q_u} \pm 1$ which is asymptotically sharp, and so $c \cdot \widetilde{q_u}$ additional balls are enough to fill each bin with probability p which depends on c. Consequently, $m_u = \widetilde{q_u} \cdot \ln \widetilde{q_u} + c \cdot \widetilde{q_u}$ polygons from $P(u)$ have to be chosen and stored in u. To compute the dependence of p on c, we refer to the proof in the textbook of Motwani and Raghavan [20, p. 61ff]. They showed that the probability p of $X \leq m_u$ is equal to $p = e^{-e^{-c}}$ for a sufficient large number of bins.

Until now we considered an arbitrary, but fixed viewpoint so that the projection of $B(u)$ is at most one pixel. The size of the sample depends on $\widetilde{q_u}$ and $\widetilde{q_u}$ depends on the viewpoint: If the viewpoint is chosen so that f_{max} is minimal, $\widetilde{q_u}$ — the ratio of $A(P(u))$ and f_{max} — is maximal. It is obvious that the minimal size of f_{max} corresponds to the area of a side face s of the bounding box of $B(u)$. Then the viewpoint is chosen so that the viewing ray V_u is orthogonal to the side face s (see Figure 4 (b)). All other viewpoints result in a greater area f_{max}. So $\widetilde{q_u}$ is maximal and equal to q_u if f_{max} is equal to $A(B(u))$. Therefore, the maximal number of polygons which have to be chosen from $P(u)$ and stored in the node u can be estimated by $m_u = q_u \cdot \ln q_u + c \cdot q_u$. ■

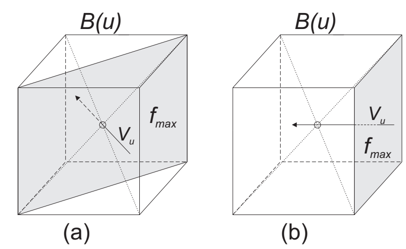

Figure 4: (a) The grey surface is orthogonal to the viewing ray V_u. This surface with an area of f_{max} lies completely in $B(u)$. (b) If the viewing ray V_u is orthogonal to a side face of $B(u)$ then f_{max} will be minimal.

Corollary 1 (Correctness of an image): Let c be an arbitrary constant and let j denote the number of nodes where the traversal did not continue to child nodes. The sample size in every node u is chosen so that with probability $p \approx e^{-e^{-c}}$ the polygons of the sample produce a correct part of the image. Then the probability of a correct image is p^j.

Proof: We produce a correct image of the scene with probability p^j because for all j nodes where the traversal stops, the corresponding area of the image is correct with arbitrarily high probability p. All other areas in the image are correct anyway because all polygons in these areas are rendered. ■

In practice, it is no problem to choose a sample size so that p is always greater than $1 - 10^{-6}$. Therefore, $c \approx -\ln 10^{-6} \approx 13.8$ has to be set. If the factor j is about 10^4, this is a realistic value in practice, and if we store $m_u = q_u \cdot \ln q_u + 13.8 \cdot q_u$ polygons in every node u, we generate a correct image with probability $p^j > 0.99$.

Remark: The definition of a correct image differs from Wand et al. [31] in that it has not to be guaranteed that always one of the selected visible polygons is nearer to the viewpoint than all selected invisible polygons. E. g., errors can occur due to situations as shown in Figure 5. We assume that only polygons A and C are selected for rendering. Thus the pixel gets the color of the invisible polygon C instead of the color of the selected visible polygon A. In our experiments we observe only few errors at object boundaries.

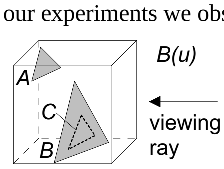

Figure 5: The octree cell contains visible polygons *A* and *B*. Polygon *C* is occluded by *B*. *B* is nearest and *A* is farthest w.r.t. the viewpoint.

4.4 Precomputation of Color Values

To summarize, one can say our method renders all polygons having a projected area of more than one pixel and renders only a representative sample of polygons for smaller ones. The traversal stops

Figure 6: The Happy Buddha rendered by the z-buffer algorithm (leftmost) and by our approach with different splat sizes: From left to right: splat size 1, 3 and 5. (This figure is reproduced in color on page 000.)

at a node u if its projected cell $B(u)$ is at most one pixel. All polygons in such a node u are projected onto an area of the image with a size of at most one pixel and cover only parts of a small neighborhood of pixels. Which color value is attached to the pixel eventually depends on the depth of the z-buffer, the sequence of the projected polygons and the graphics library that is used. In order to improve this procedure with the aim to get a better color value for the pixel, a color value that is representative for all polygons stored in a node u can be determined. This color value is rendered with the size of one pixel instead of the corresponding polygons, if the projected bounding box of $B(u)$ is at most one pixel. In our implementation the color value is determined as the median of all polygon colors depending on the polygon areas. It is also conceivable to calculate a better color value by using Gaussian filters and considering adjacent color values.

4.5 Far-Off-Splatting

In the context of point sample algorithms the notion of splatting has become popular in the last few years. Instead of rendering points or single pixels many approaches render splats in order to reduce rendering time and to guarantee that the images show no holes. Based on this idea we developed the far-off-splatting. Thereby, objects far-off are mainly represented by splats and only the larger polygons are rendered for these objects. For objects near to the viewer it is the other way round: Generally, polygons are used for drawing them and splats are only used sparsely so that the image quality of these objects is good. With this technique the rendering times can be improved while the images are of good quality.

Specifically the far-off-splatting works as follows: Instead of traversing the tree until the projected bounding box of a node is at most one pixel, the traversal can be stopped earlier at a node u, if its projected bounding box is at most i^2 pixels, $i \in \{1,2,3,\ldots\}$. For such a node u, a color value computed in the preprocessing is rendered as a splat of size i^2 pixels instead of the polygons stored in u. We denote i as the splat size. Thereby, we ensure that there are no holes in the rendered images. Note that the projection of bounding boxes near to the viewer is mostly larger than i^2 pixels (for small i) so that the corresponding polygons are rendered and a high image quality can be achieved.

5. EXTERNAL CONSTRUCTION OF THE SAMPLE TREE

If the scenes are highly complex and cannot be completely stored in main memory, the sample tree cannot be constructed in an easy top-down manner. The problem is that for *every* node u one has to draw a polygon m_u times from *all* polygons $P(u)$. This cannot be done within an acceptable amount of time in the preprocessing phase because only a small part of polygons can be stored in main memory. Many hard disk accesses would be necessary.

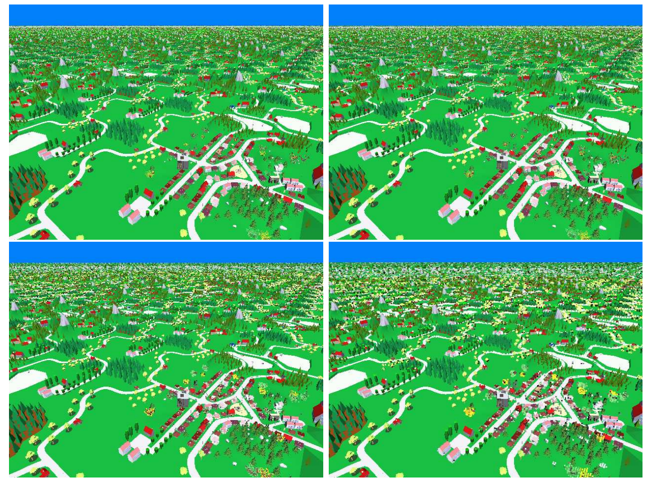

Figure 7: Scene *Landscape 2* (more than 2 billion polygons) rendered with different splat sizes in comparison to the z-buffer algorithm (upper left). Upper right: splat size 1, lower left: splat size 3, lower right: splat size 5. (This figure is reproduced in color on page 000.)

To overcome this problem we construct the sample tree in a bottom-up manner that differs a little from the algorithm described in Section 4.1 (see Figure 3). For this purpose, we have to construct an octree from which afterwards our sample tree is built and stored on hard disk. The octree has to be precomputed because we have to know which polygons lie in each leaf node of the sample tree at the beginning of its construction.

5.1 Octree Construction

One problem to be solved is that one cannot store every node of the octree or of the sample tree in a single file. During the construction, as well as during the walkthrough, too many files would have to be opened and closed which would result in a long preprocessing time or in a high response time of the node server. Another problem to overcome is how to store all nodes with their polygon data. This cannot be done in a single file because its size is restricted by most operating systems. Furthermore, it is important to store nodes of neighboring cells in a single file because often these nodes are requested immediately one after another.

Because of these problems, an octree with a fixed depth is first constructed in two steps. The final depth – by which it can be guaranteed that every leaf node does not store more than a fixed number of polygons – cannot be determined in advance because of the unknown spatial distribution of the polygons. In the first step, the structure of the octree is determined and for every node, the number of polygons lying in the corresponding octree cell is counted. With this information we can identify the nodes (with their polygon data) which should be stored together. In the second step the polygons are stored in the file of the corresponding node. After this construction, leaf nodes are subdivided into subnodes if they contain more than a fixed number of polygons.

5.2 Sample Tree Construction

In the following, we construct a sample tree for a node u, that differs a little from the definition in Section 4, but still satisfies the desired sample tree property. We have to modify the bottom-up construction (see Figure 3) because we have to achieve that as few polygons as possible are loaded from hard disk into main memory

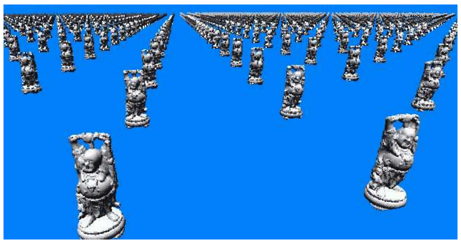

Figure 8: Happy Buddha (instantiation): More than 10,000,000,000 polygons are rendered in 380 msec (splat size 2). (This figure is reproduced in color on page 000.)

Scene	polygons	preprocessing	polygons per node (max/avg)	disk usage (total / polygon data)
Landscape 1 (LS 1)	$642,847$	$52sec$	$529/11$	$54.7MB/46.6MB$
Landscape 2 (inst. of LS 1)	2.01 billion	$(52sec)$	$(529/11)$	$(54.7MB/46.6MB)$
Landscape 3 (out-of-core)	$406,149,559$	$30.5h$	$584/21$	$29.68GB/28.58GB$
Happy Buddha (single)	$1,087,474$	$86sec$	$38/3$	$102.8MB/78.8MB$
Happy Buddha (inst.)	10.87 billion	$(86sec)$	$(38/3)$	$(102.8MB/78.8MB)$
Chessboard (small)	$80,000$	$7sec$	$35/8$	$6.97MB/5.80MB$
Chessboard (huge)	$320,000,000$	$23.3h$	$47/10$	$24.44GB/22.65GB$

Figure 9: Statistical overview of the sample trees for different scenes. *Preprocessing* denotes the time for constructing the sample tree. *The total disk usage* includes the polygon data and the additional data for storing the sample tree. By instantiating objects, neither the preprocessing time nor the sample tree structure is changed (the corresponding values are in parentheses).

to reduce expensive hard disk accesses. So the sample tree can be built up in an acceptable preprocessing time.

The sample tree is constructed bottom-up from the octree. Let the subtrees of any node u be sample trees. The child nodes d_i, for $i = 1 \ldots 8$, store exactly the polygons that give a good approximation of $B(d_i)$ if the projected size of $B(d_i)$ is at most one pixel. Therefore, it is sufficient to sample only from polygons that are stored in the direct child nodes d_i, and store the sampled polygons in u. Of course, one has to remove the chosen polygons from the child nodes. This construction has the effect that polygons stored in the leaf nodes of the octree are shifted up successively.

6. NAVIGATION IN EXTERNALLY STORED SCENES

To allow an interactive navigation in scenes stored on hard disk, the following requirements have to be considered: 1. To compute a new image, only a few nodes should be loaded. 2. Loading a node may take only very short time. An appropriate algorithm is described in this section. Thereby, we assume that the viewer moves only slightly between the computation of two sucessive frames. Also, we assume that the polygon samples needed for the computation of one frame can be stored in main memory. In practice, this is not a restriction because the samples are very small even for our most complex scenes.

6.1 Client-Server Model

In order to store scenes on hard disk and to load data during navigation, we have implemented a client-server structure (see Figure 2). The sample tree is stored on the hard disk of the server and is managed by the node manager. The client is connected to the server via TCP/IP and serves as a rendering workstation.

The traversal of the sample tree and the rendering of polygons are executed on the client. At the beginning, there is no sample tree on the client so that the client sends a request to our node manager with the unique node identifier of the root node. The received root node is stored in main memory. While the traversal has not stopped at a node u, and the required child node is not stored in main memory, the client sends requests to the server and stores the received nodes in memory. Of course, one cannot manage all nodes in main memory at the same time. For this, we developed a caching mechanism which deletes nodes in main memory but keeps nodes that are needed for the next frames.

6.2 Caching and Deleting Nodes

We use a very simple but efficient method for caching nodes: If the traversal stops at a node u stored in the k-th level of the sample tree, all child nodes of the next j levels are kept in main memory. j is arbitrary but fixed and measurements show that $j \in \{1,2\}$ provides the best results. All nodes of the next levels having a depth of more than $k + j$ are deleted from memory. This method can be improved by implementing the data transfer of the nodes via TCP/IP and the rendering and the traversal in the sample tree to be asynchronous. That means if a request is sent to the server, the traversal is not stopped until the requested node is transferred. The traversal and rendering of the polygons are executed simultaneously with the data transfer. The measurements in section 7 show that with this caching mechanism, only very few nodes have to be loaded from one frame to the next. Additionally, only small samples of polygons are stored in the nodes, resulting in a low data transfer.

7. IMPLEMENTATION AND RESULTS

We implemented our proposed approach in a prototypical walkthrough system. Our C++ code has not been optimized. The precomputed octree has to be kept on hard disk during the sample tree construction. So we need double memory capacity which means that for computing the sample tree of our largest scene (30 GB) we need 60 GB on hard disk. All tests were performed on a Linux-based system with a 2GHz Pentium 4 using 1GB of main memory, nVIDIA GeForce3 graphics card and a 80 GB IDE hard disk. For all benchmarks we chose a resolution of 640×480 pixels.

In order to show the characteristics and advantages of our method, we modelled some scenes which will be described in detail later. The scenes are different with respect to the scene complexity, the complexity of each 3D model, the dimension and the structure of the scene. In every benchmark in this section, another aspect of our approach is examined. The aim is to show that the idea of drawing the samples during the preprocessing and of loading them from hard disk during the navigation allows an interactive walkthrough as described in the introduction. Additionally, it is demonstrated that the precomputation of the samples in a preprocessing phase does not lead to a poor quality in comparison to other methods as, e. g., the approach proposed by Wand et al. [31]. In order to get a first overview of the scenes and the corresponding sample trees one should look at the table shown in Figure 9. There, one can also see the necessary preprocessing times for constructing the sample trees.

7.1 Image Quality

The theoretical analysis concerning the image quality that is given in Section 4 should now be verified in practice. Figures 6 and 7 compare the quality of images computed by our method with an image computed by the z-buffer algorithm, depending on different splat sizes (see Section 4.5).

First, we examine the quality of the scene *Landscape 2* (more than 2 billion polygons), where many objects are far-off in the distance. In Figure 7 the upper left picture is rendered by the z-buffer algorithm. In comparison, the upper right picture rendered by our approach (splat size 1) is of the same quality. In the two pictures

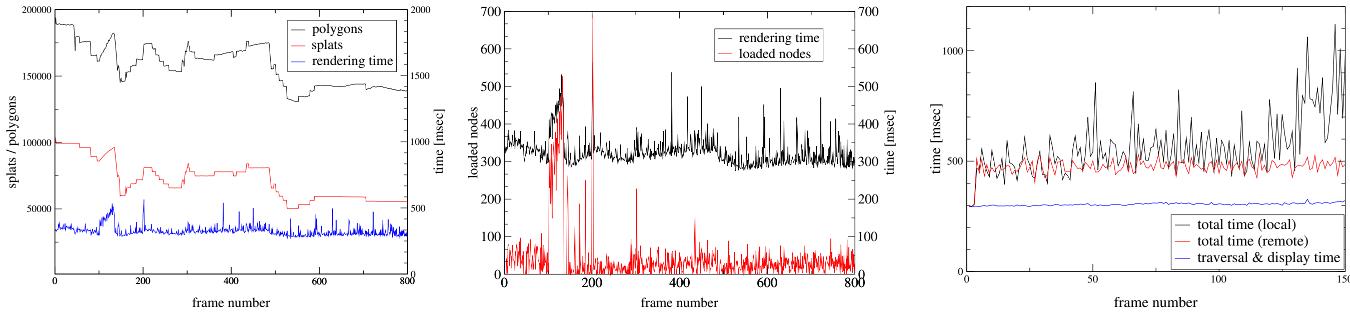

Figure 10: Measurements for *Landscape 2*. Left: Number of rendered polygons and splats. Center: Number of nodes to be loaded during the navigation. Right: Rendering time for storing the scene on a local and remote hard disk. The total time includes the communication between server and client.

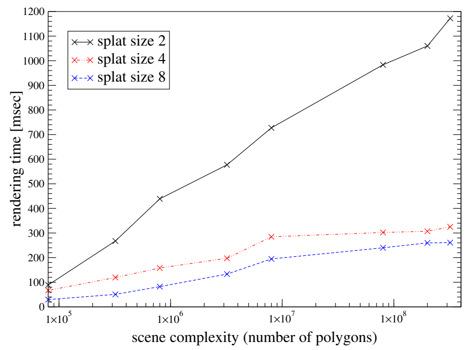

Figure 11: Rendering time for chessboard scenes depending on the scene complexity and the splat size. The time grows roughly logarithmically with the polygon number. Note that the x-axis is logarithmically scaled.

below (left: splat size 3, right: splat size 5) the quality is a little bit worse, as one can see some wrong pixels. In contrast to this landscape scene, Figure 6 shows the happy buddha model [7] (scene *Happy Buddha (single)*) which is quite complex and consists of more than one million polygons. These polygons are lying in a very small volume and – compared to the landscape scene – many polygons are very close to the viewer. This model is used by other researchers so that a comparison is easily possible (we have taken the frequently used resolution of 640×480 pixels). Note that even for splat size 5 the image quality is remarkably good.

7.2 Complexity of Scenes

In this section we explain why the rendering time depends sublinearly on the polygon number. We chose chessboard models (consisting of black and white polygons) with fixed dimensions but increasing number of polygons from 80,000 (*Chessboard (small)*) to maximal 320 million (*Chessboard (huge)*) as our benchmark. Thereby, constant conditions concerning the projected area, the illumination and other geometric parameters can be guaranteed. Note that every polygon is stored separately on a remote hard disk. The viewer moves over the chessboard at a very low altitude so that many polygons are close to the viewer and nearly all polygons of the scene lie in the view frustum. If the viewer were farther away, computing an image would take shorter time since the traversal in the sample tree would be stopped earlier.

In Figure 11 one can see that the rendering time is sublinear in the number of polygons: The computation of an image for a scene consisting of 8 million polygons amounts to 284 msec, whereby the rendering of a scene consisting of 320 million polygons takes about 324 msec per image (splat size 4). These values are averaged over a path consisting of 100 steps. For a splat size of 8, an interactive navigation in the most complex scene (320 million polygons) is possible with about 4 fps. Thereby, the image quality is quite good and the quality of the area close to the viewer is almost equal to that of images computed by the z-buffer algorithm.

7.3 Out-Of-Core Storage

We want to show that our developed data structure – the randomized sample tree – is very well suited to render scenes that cannot completely be stored in main memory. Therefore we consider our largest scene *Landscape 3* consisting of more than 400 million polygons (30 GB). The organization and the design of the 3D scene objects are comparable to those shown in Figure 7. The viewer moves in 800 steps over the scene at a low altitude whereby the way is not only straight on: So, on the one hand most of the polygons near to the viewer have to be rendered, and by changing the viewing direction more new nodes have to be loaded.

Of course, the rendering times are very interesting but one has to take into account that they depend highly on code optimization and the system used. Therefore, we also show how many nodes and polygons are loaded from hard disk and we look at the size of rendered polygons and splats. First of all one should examine the table shown in Figure 9. There, one can see that the average number of polygons per node as well as the maximum number is small for all scenes. Moreover, we only load a few nodes from hard disk during the navigation as one can see in Figure 10 (center): On average, no more than 100 nodes are loaded per frame. In the same figure the rendering times are shown whereby the average time for rendering an image takes 326 msec (splat size 5).

The Figure 10 (left) depicts the number of polygons and splats that have to be rendered on the chosen path through our scene. As one can see, the ratio between the polygons and splats is roughly the same on the whole path. About 157,000 polygons and about 70,000 splats are drawn for each frame on average, and always twice as many polygons are rendered than splats. Clearly, the ratio is roughly the same: For complex objects the sample tree is deeper than for less complex objects. So, if the viewer moves to a complex object, the sample tree is traversed deeper and more polygons are rendered. Furthermore, there are more nodes where the traversal is stopped and for which splats have to be drawn.

To summarize, one can say that only few nodes with few polygons have to be loaded for each frame resulting in a low transfer time. The transfer time that depends on storing the sample tree on a local or remote hard disk is examined in the next section.

7.4 Local versus Remote Rendering

Now we compare the rendering times for our largest scene *Landscape 3* (400 million polygons, 30 GB) depending on whether the sample tree is stored on a remote disk or whether it is stored on a local hard disk. See Figure 10 (right). We chose a splat size of 5 in order to get good quality images. A path through the scene was fixed so that as many polygons as possible are in the viewing cone. Furthermore, we violated our assumption that the viewpoint does not move over great distances within a short period of time: thus, many nodes had to be loaded from hard disk for each frame. This makes sense since we want to examine the time for loading the polygons and not the rendering time for a smooth walkthrough. The time for the traversal of our sample tree plus the display time (time used to send the polygons to the graphics card) amounts to 300 msec in the average case, regardless of storing the scene on a local or remote hard disk. If the scene is stored on local hard disk, the total time for the computation of an image amounts to 570 msec in the average case. The difference between these two time values is the communication time between client and server.

If the same scene is stored on a remote hard disk that is connected to the client by a fast network (100 MBit), the total time only amounts to 470 msec in the average case. This can be explained by the fact that if client and server run on the same system, they compete with each other for computing time and main memory. Furthermore, if the server runs on a remote system it can store more files in its cache.

It is obvious that rendering the first image takes longer than rendering an image during the navigation: At the beginning no polygons are stored in main memory and therefore all needed polygons have to be loaded from hard disk. Our measurements show that 36 seconds are necessary for the first image of our largest scene.

7.5 Instantiation of Objects

Our approach requires no instantiation schemes for managing highly complex scenes. Every polygon is stored separately on hard disk. Nevertheless, we have implemented the ability to instantiate objects in order to show the power of our method applied to scenes that exceed our hard disk capacities.

Figure 7 shows an instantiated scene (*Landscape 2*) consisting of more than 2 billion polygons that can completely be stored in main memory. The z-buffer algorithm needs about 50 minutes for the computation of one image while our algorithm (splat size 1) needs only 2.5 sec for an image of the same quality. With a splat size of 2, the rendering time amounts to 960 msec and with splat size 5 rendering takes 269 msec for images of acceptable quality which means we have a frame rate of more than 3.7 fps. The rendering times are average values for a path of 1000 steps through the scene. During the navigation camera positions are chosen very close to some objects as well as far-off positions with nearly all polygons in the view frustum. For splat size 2 we render about 301,750 polygons and 110,400 splats, and for splat size 5 we have about 87,860 polygons and 32,110 splats per image.

To show that our approach also works with scenes consisting of many very complex and smooth models we created the scene *Happy Buddha (instantiation)* with 10,000 instances of a single happy buddha model (more than 10^{10} polygons, see Figure 8). We chose a path through the scene of the same kind as described above. With a splat size 2, images can be computed within 334 msec and with splat size 4, frame rates of about 5.9 fps can be achieved on average. Thereby, about 40,130 polygons and 15,330 splats are rendered per image, whereby with splat size 2 about 65,900 polygons and 65,230 splats are drawn.

8. CONCLUSION AND FUTURE WORK

We developed a method that allows an interactive walkthrough in virtual environments of arbitrary topology that cannot be stored in main memory, but rather on hard disk. Our experiments illustrate that the running time of our algorithm depends only weakly on the number of polygons and show that our approach is suitable for highly complex scenes. In contrast to recent point sample approaches, our memory consumption is only linear in the number of polygons. Our method computes the samples in a preprocessing phase so that no expensive computations are necessary as in [31] in order to specify the samples during the navigation. Furthermore, our randomized sample tree requires loading only a small amount of polygon data from hard disk if the viewpoint moves only slightly between the computation of two sucessive frames. This means that acceptable rendering times can be achieved. We also showed that with arbitrarily high probability, correct images can be computed.

There are some possibilities to extend our work and to further improve the rendering time.

Anti-aliasing: Our precomputation of color values (see Section 4.4) is only a very simple method for avoiding aliasing artifacts. By using Gaussian filters during the precomputation, one would probably get better anti-aliased images. It is also possible to use randomized visibility tests during the walkthrough for better anti-aliasing.

Occlusion culling: As the running time of our method is linear in the projected area of polygons, it is desirable to implement an occlusion-culling algorithm. Well-known algorithms should fit very well to our sample tree as described in Section 3. It is also conceivable to use randomization for the visibility tests.

Dynamic updates: Dynamic updates of scenes could be very useful for some applications. Therefore, the sample tree has to be modified during the navigation for inserting, moving or deleting single objects.

Network-based rendering: Our method allows interactive navigation in highly complex scenes stored on a remote hard disk only on high bandwidth networks. Integrating the possibility of adjusting the approximation quality to the bandwidth of the network might be a good idea.

Acknowledgements

The authors wish to thank Birgitta Derenthal for valuable discussion and her support of our work. Thanks also to José Luis de los Rios, Marcus Werner, and Eric Torunski for modelling the scenes. This work is partially supported by DFG grant ME 872/8-1 and the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

9. REFERENCES

[1] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *Proc. IEEE Visualization 2001*, pages 21–28, 2001.

[2] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[3] B. Chen and M. X. Nguyen. POP: A hybrid point and polygon rendering system for large data. In *Proc. IEEE Visualization 2001*, pages 45–52, 2001.

[4] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[5] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand. A survey of visibility for walkthrough applications. *Transactions on Visualization and Computer Graphics*, 2002. to appear.

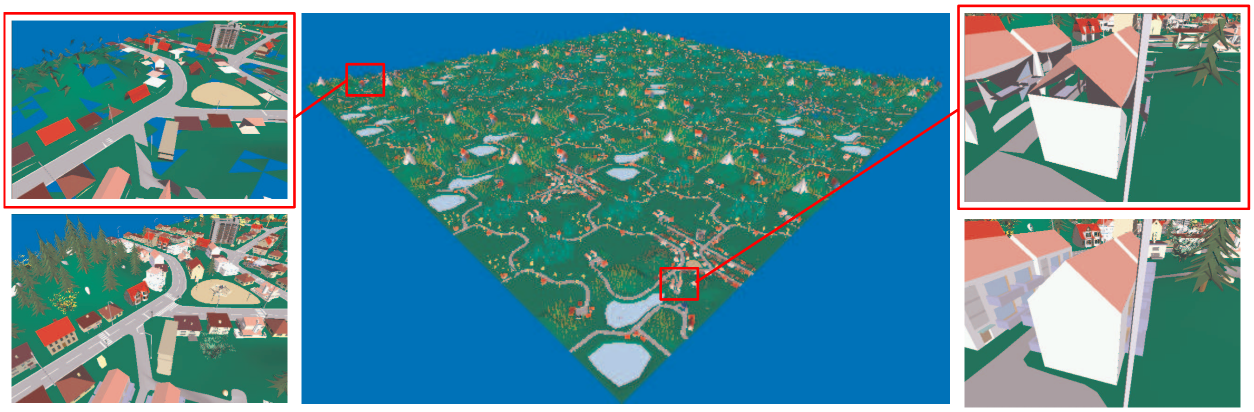

Figure 12: Two parts of the image are zoomed and show that only samples of polygons are rendered. Below, the corresponding parts of the original scene are rendered by the z-buffer algorithm. We used splat size 4 and left out the precomputation of color values for better illustration. (This figure is reproduced in color on page 000.)

[6] D. Cohen-Or and E. Zadicario. Visibility streaming for network-based walkthroughs. In *Graphics Interface 1998*, pages 1–7, 1998.

[7] B. Curless. The happy buddha model. Stanford Computer Graphics Laboratory.

[8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Computer Graphics (SIGGRAPH 1995 Conference Proc.)*, pages 173–182, 1995.

[9] J. El-Sana, N. Sokolovsky, and C. T. Silva. Integrating occlusion culling with view-dependent rendering. In *Proc. IEEE Visualization 2001*, pages 371–378, 2001.

[10] T. A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics 1995*, pages 85–92, 1995.

[11] T. A. Funkhouser. Database management for interactive display of large architectural models. In *Graphics Interface 1996*, pages 1–8, Toronto, 1996.

[12] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Computer Graphics (SIGGRAPH 1993 Conference Proc.)*, pages 231–238, 1993.

[13] J. P. Grossman and W. Dally. Point sample rendering. In *Proc. Rendering Techniques 1998*, pages 181–192, 1998.

[14] P. S. Heckbert and M. Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface 1994*, pages 43–50, 1994.

[15] H. Hoppe. Progressive meshes. In *Computer Graphics (SIGGRAPH 1996 Conference Proc.)*, pages 99–108, 1996.

[16] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report 85-022, Computer Science Department, University of North Carolina, 1985.

[17] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proc. IEEE Visualization 2001*, pages 363–370, 2001.

[18] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics 1995*, pages 95–102, 1995.

[19] Y. Mann and D. Cohen-Or. Selective pixel transmission for navigating in remote virtual environments. In *Computer Graphics Forum (Proc. EUROGRAPHICS 1997)*, 1997.

[20] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[21] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Computer Graphics (SIGGRAPH 2000 Conference Proc.)*, pages 335–342, 2000.

[22] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Computer Graphics (SIGGRAPH 2000 Conference Proc.)*, pages 343–352, 2000.

[23] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics 2001*, 2001.

[24] D. Schmalstieg and M. Gervautz. Demand-driven geometry transmission for distributed virtual environments. In *Computer Graphics Forum (Proc. EUROGRAPHICS 1996)*, number 3, pages 421–433, 1996.

[25] J. M. Sewell. *Managing Complex Models for Computer Graphics*. PhD thesis, University of Cambridge, Queens' College, 1996.

[26] L. Sobierajski Avila and W. Schroeder. Interactive visualization of aircraft and power generation engines. In *Proc. IEEE Visualization 1997*, pages 483–486, 1997.

[27] M. Stamminger and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Proc. Rendering Techniques 2001*, pages 151–162, 2001.

[28] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (SIGGRAPH 1991 Conference Proc.)*, pages 61 – 69, 1991.

[29] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 277–288, 2001.

[30] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum. 20(3)*, pages 153–164, 2001.

[31] M. Wand, M. Fischer, I. Peter, F. Meyer auf der Heide, and W. Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Computer Graphics (SIGGRAPH 2001 Conference Proc.)*, pages 361–370, 2001.

[32] J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison Wesley, second edition, 1994.