The Randomized Sample Tree: A Data Structure for Externally Stored 3D-Scenes

Jan Klein Jens Krokowski Matthias Fischer Rolf Wanka Friedhelm Meyer auf der Heide

University of Paderborn *

Abstract

We present a new algorithm for rendering highly complex 3D scenes of arbitrary topology. The specific feature of our method is that it makes possible an interactive navigation of scenes consisting of more than 16 GB of polygon data (95 million polygons) that cannot be stored in the main memory, but only on a local or remote hard disk.

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 uses only space 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. The quality of our images is comparable to images computed by the conventional z-buffer algorithm regardless of the scene topology.

CR Categories: 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, Monte Carlo Techniques, Spatial Data Structures, Point Sample Rendering

1 Introduction

Despite of advanced graphics hardware, real-time navigation in highly complex scenes is one of the fundamental problems 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 the main memory are still very difficult to handle.

In this paper, we present a new randomized rendering algorithm that allows an interactive walkthrough in highly complex scenes stored on local or remote hard disk. To generate an approximate image of a scene we render only samples of polygons in order to reduce the scene complexity. The samples are chosen in such a way that the quality of the rendered images is comparable to that of images computed by the conventional z-buffer algorithm. The scene is stored in the sample tree, an octree-like data structure. In the preprocessing phase, every node stores a sample of polygons that approximates the corresponding cell. For rendering an image, the sample tree is traversed according to the viewpoint and polygons from the visited nodes are sent to the graphics subsystem. The traversal stops at a node if the corresponding cell has a projected area of at most one pixel. Every polygon is stored exactly once in the tree so that we need space linear in the number of polygons in practice.

There are two main topics in our work: 1. We want to achieve interactive rendering of complex scenes in output-sensitive time, i.e., the rendering time is sublinear in the number of polygons. 2. We want to navigate through scenes that cannot be stored completely in the main memory but only on hard disk. The number of polygons of a scene should only be restricted by the size of the hard disk.

In the following, we assume that the samples of polygons needed for the computation of one frame can be stored in the main memory. This is no real restriction in practice because the samples are very small even for our most complex scenes.

1. Interactive rendering in complex scenes: For interactive rendering, the running time must not be linear in the number of polygons, but rather sublinear. To obtain this, we reduce the scene complexity by rendering only 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.

2. Complex scenes stored on hard disk: Often scenes are very complex and consist of many different types of models. That's why one cannot use instantiation schemes [24] to keep the whole scene in the main memory. This is a problem as rendering algorithms must send polygons as fast as possible to the graphics subsystem to achieve interactive rendering. If the data cannot be managed in the main memory, it must be stored on disk resulting in expensive hard disk accesses.

We solve this problem 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. With the help of a caching mechanism, only very few nodes have to be loaded from one frame to the next (we assume that viewpoint does not move over great distances in short 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 section 7 we show that an interactive navigation in scenes consisting of more than 95 million polygons (more than 16 GB polygon data) that cannot be stored in the main memory is possible.

In the following, further aspects of our algorithm are explained.

Network-based rendering: Based on the idea that the scenes are highly complex so that they cannot be stored on the local hard

*Dept. of Mathematics & Computer Science and Heinz Nixdorf Institute, University of Paderborn, D-33095 Paderborn, Germany. Email:{janklein, kroko, mafi, wanka, fmadh}@uni-paderborn.de

disk, we implemented our algorithm in a client-server network: The scene is stored on a remote server connected to the client via TCP/IP. Then the client loads the polygon data from the remote hard disk into its main memory. In section 7, we show that for networks with a good bandwidth good frame rates can be achieved.

Arbitrary scene topology: The rendering times and the quality of the images computed by our algorithm are independent of the topology and the position of the camera. Holes or disconnectivities in the models cause no problems. A typical scene for our rendering technique may consist of different models (e.g., landscapes with trees, flowers, buildings and cars).

Easy implementation: Our rendering algorithm is simple and easy to implement. The input is the polygonal description of the scene, e.g., triangles, normals, material colors. We need neither complicated level-of-detail computation of the polygons nor memory-expensive replacement by textures.

Efficiency: The space for storing our sample tree is only linear in the number of polygons because we store every polygon exactly once in the tree. 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 95 millions of polygons. Thereby we compute an image within 290 msec if the sample tree is stored on a remote hard disk.

This paper is organized as follows: In section 2, we give an outline of our algorithm. In section 3, related work is discussed and similarities and differences to our method are described. In the next section, the structure of the sample tree and the rendering algorithm are presented. We determine the size of the samples stored in each node to guarantee a correct image with arbitrarily high probability. Section 5 describes the construction of the sample tree on the hard disk. In the following section, we discuss the details of rendering scenes stored on local or remote hard disk. Our test results are presented in section 7. Section 8 concludes our work and gives an outlook on future work.

2 Outline of the rendering algorithm

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. *Far away* means that the projected area of the cell is at most one pixel. In the leaf nodes, all polygons that were not chosen for the samples in the ancestor nodes are stored. Note that every polygon occurs exactly once in the tree.

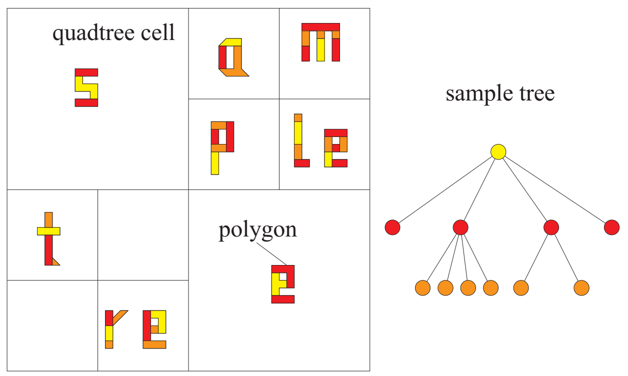

Figure 1: Schematic example of a sample tree for a 2-dimensional scene with a quadtree.

Figure 1 demonstrates the idea of our sample tree. For simpler drawing, we take a 2D scene and a quadtree instead of an octree. Note that the polygons are marked in different colors only for demonstration. Their real colors could be different from the colors in the figure. 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 store them in the root node. These polygons are marked 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 depending on their areas from the corresponding cell (we do not consider polygons that are already stored in the root node). The chosen polygons, marked 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 projected area of the corresponding cell is at most one pixel. 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 of the corresponding cell are rendered. If we rendered only a sample of polygons, the approximation would result in a low image quality because the viewpoint is close to the cell. This kind of traversal is well-known in the context of level-of-detail systems (e.g., see [13]).

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 the main memory. The walkthrough system consists of a rendering workstation (client) and a node manager (server) which handles the sample tree stored on its local disk (see Figure 2). Server and client are connected via TCP/IP and can run on one single computer to store the scene on the local disk of the rendering system or on different ones in order to store the scene on a remote hard disk.

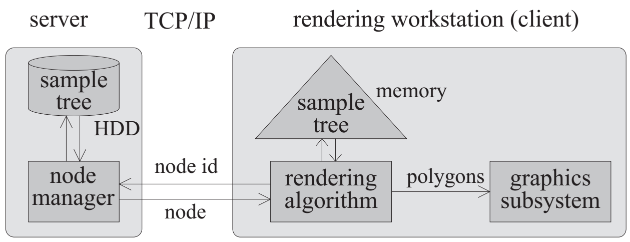

Figure 2: Walkthrough system: client and server connected via TCP/IP, the polygon data are stored on hard disk.

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 the main memory the rendering workstation loads it via TCP/IP from the node manager (server). Therefore, the client sends a unique identifier to the server. If the viewpoint changes, some new nodes of the sample tree are loaded and replace some of the current nodes stored in the main memory.

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 [20].

3 Previous Work and Related Areas

We will summarize related work from point sampling and network-based walkthroughs.

Point sample rendering: Our rendering technique has the most similarities 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.

Several researchers have proposed to use points as display primitives [12, 2]. Their rendering is simple and rasterization is cheaper than that of polygons. 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 sample rendering. [9] uses points both as rendering and as modelling primitives. QSplat [17] and Surfels [16] use prefiltering methods to convert graphics primitives into a uniform point representation. The points are selected from a level-of-detail-like representation according to their screen projection. This results in a low rendering time and a high image quality.

QSplat [17] 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 [16]. A further solution for the problem of close views is given by [1, 21, 23], 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.

We do not use any conversion of the rendering primitives into points as the algorithms mentioned above, but we only render polygons. This results in aliasing artifacts that can be reduced by precomputing color values for the nodes of our sample tree, and in a slightly more expensive rendering time. Our method is quite simple and allows a correct rendering of scenes of arbitrary topology and avoids the problem of a fixed resolution if the viewpoint is near to an object. Furthermore, an advantage of our method is that we only need space linear in the number of polygons. This cannot always be guaranteed by point-based methods.

As the approach of Wand et al. [23] our method renders all polygons having a projected area of more than one pixel. The power of our method lies in rendering highly complex scenes where many polygons are projected only onto a part of a pixel. Our algorithm has the advantage that the samples are only computed once in the preprocessing and do not have to be chosen for each frame during the walkthrough as in [23]. The developed data structure — the randomized sample tree — constitutes the basis for storing the scenes on a hard disk and loading needed polygons during the walkthrough. As the approach of Wand et al. [23], our methods caches samples and reuses them for the next frames.

Network-based walkthroughs: Transmitting 3D data across a network has become an interesting field of research.

In the context of occlusion-culling, the PVS approach presented by Teller and Séquin [22] 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 [7, 8].

In order to use the network capacity consistently despite of sharply changing viewpoints, Cohen-Or and Zadicario [5] perform a prefetching of an ε-neighborhood to ensure that off-screen objects are loaded into memory

Mann and Cohen-Or [14] 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 [19] 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 the main memory of the client in order to reduce the network transmission.

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 [11] represents a model by a base mesh and refinements of this mesh. The mesh correction can take place by vertex split operations [11] or by wavelet encoded operations [6].

Rusinkiewicz and Levoy [18] 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.

Our algorithm does not yet perform any occlusion culling on the server, as the described PVS methods above, nor on the client. It performs view frustum culling only. The running time depends linearly on the projected area a of sampled polygons s, including hidden polygons. Since we use the z-buffer algorithms for rendering of the sampled polygons, the running time is $O(s+a)$ [10]. Thus, our algorithm becomes inefficient if there are many polygons near to the viewer because these polygons have a great projected area.

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 many samples of polygons have to be chosen so that, with arbitrarily high probability, a correct image is produced. In the last two subsections, extensions of our algorithm to improve the image quality and to accelerate the rendering time are described.

In order to explain the main idea of our sample tree, let us first define the notation *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 so that the corresponding pixel gets the color of r. The whole image is correct if all parts of the image are correct.

4.1 The Data Structure

In this section, we describe the hierarchical structure of our sample tree which corresponds to a modified octree. Polygons are only stored in a node u, if they lie completely in the bounding sphere of the corresponding octree cell of u. Furthermore, every polygon is stored exactly once in the sample tree. The insertion of the polygons into the nodes is described in the following.

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 polygons is stored. These samples together with the samples stored in the ancestor nodes approximate $P(u)$. To pick a sample stored in node u we look at all polygons $P(u)$ and choose a polygon r for the sample set with a probability proportional to its area, i.e., with probability $A(r)/A(P(u))$. We have to store sufficiently many polygons in u so that their projection *approximates well* the polygons $P(u)$ if the viewpoint is *far away* from $B(u)$. *Approximates well* means that the projection of all polygons stored in u produces a correct part of the image, with high probability p, which depends on the sample size. The size of the sample in turn depends on $A(P(u))$. Its choice is described in detail in section 4.3. *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. In practice, 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.

Figure 3 shows the algorithm that computes our sample tree. Before starting the algorithm, an octree has to be constructed where polygons are inserted in the 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 is determined in section 4.3.

```
createSample ( node u )
  {
    if u is not a leaf node
    {
      for every child c of u do createSample ( c )
      M = ∅
      for i = 1 to m 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 a bottom-up manner. It has to be started with the root node of a precomputed octree.

4.2 Rendering an Image

In order to render an image of a scene, 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 sampled polygons stored in u are rendered and produce (with arbitrarily high probability) a correct part of the image according to the construction of our tree. 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. As already mentioned, every polygon is stored once in the tree.

Obviously, 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. We choose polygons randomly with a probability proportional to their areas (that means we add a polygon r into the sample with probability $A(r)/A(P(u))$ because 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 estimate the size of the sample well, as too many polygons result in slow rendering and too few polygons result in a low image quality.

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 follows:

$$q_u = \left\lceil \frac{A(P(u))}{A(B(u))} \right\rceil$$

Let c be an arbitrary constant. If $m = q_u \cdot \ln q_u + c \cdot q_u$ independent drawings from $P(u)$ are done and the chosen polygons are stored in 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 node u. 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).

We consider an arbitrary viewpoint and viewing direction onto $B(u)$ so that the projection of $B(u)$ is at most one pixel. Let f_{max} be the maximum area of a surface orthogonal to the viewing ray 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 $P(u)$ shows no holes. If some visible polygons are not orthogonal to the viewing ray, the area of all visible polygons or parts of polygons will be greater than f_{max}.

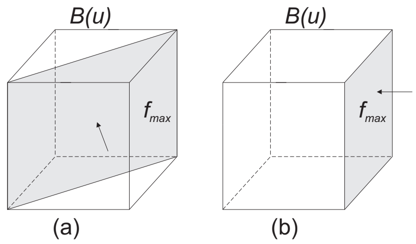

Figure 4: (a) The grey marked surface is orthogonal to the viewing ray. This surface with an area of f_{max} lies completely in $B(u)$. (b) If the viewing ray is orthogonal to a side face of $B(u)$ then f_{max} will be minimal.

For the sake of argument, let us partition the polygons from $P(u)$ into $\widetilde{q_u}$ groups so that all polygons of a 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 follows:

$$\widetilde{q_u} = \left\lceil \frac{A(P(u))}{f_{max}} \right\rceil .$$

All visible polygons or parts of 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 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, seen from an arbitrary but fixed viewpoint, m polygons have to be stored in u so that a visible polygon from $P(u)$ is between these m polygons. When all m 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 [15]) 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 a probability p which depends on c. Consequently, $m = \widetilde{q_u} \cdot \ln \widetilde{q_u} + c \cdot \widetilde{q_u}$ 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 [15]. They showed that the probability p for $X \leq m$ is equal to $p = e^{-e^{-c}}$ for a sufficient large number of bins.

Until now, viewpoint and viewing direction are arbitrary but fixed and satisfy the condition that the projected bounding box 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 a viewing direction is chosen so that the viewing ray 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 u can be estimated by $m = q_u \cdot \ln q_u + c \cdot q_u$.

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 $1 - 10^{-10}$. Therefore, $c \approx -\ln 10^{-10} \approx 23$ has to be set. If the factor j is about 10^6 — a dimension we have never reached — the probability p^j is still greater than 0.9999.

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 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 few 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 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 Splatting

To increase the efficiency of our rendering algorithm, one can do the following: Until now the traversal stops at a node u if the projected bounding box of $B(u)$ is at most one pixel. Alternatively, the traversal can be stopped earlier at a node u, if the projected bounding box of $B(u)$ is at most i^2 pixels, $i \in \{1,2,3,...\}$. Analogously to the described method, all polygons stored in u can be rendered or in the preprocessing phase, averaged color values can be computed. This color values are rendered as splats of size i^2 pixels instead of the polygons. We denote i as the splat size. Thereby we ensure that there are no holes in the rendered images. The size of the samples has to be adjusted accordingly in order to get a correct image as described above. Now $m' = i^2 q_u \cdot \ln(i^2 q_u) + c \cdot i^2 q_u$ polygons have to be chosen for one node u. The number q_u of groups of polygons (see section 4.3 for more detail) has to be increased by i^2 because i^2 visible polygons from every group of the q_u groups have to be chosen instead of only one polygon as before.

Indeed experimental evaluation also shows that without increasing the sample size the image is not correct but the quality is still good, if the traversal stops one or two levels earlier in the tree. The consequences are better frame rates with an acceptable rendering quality.

5 External Construction of the Sample Tree

If our scenes are highly complex so that they cannot be stored completely in the main memory, the sample tree cannot be constructed in a top-down manner. The problem is the calculation of the necessary factor q_u (see section 4.3) for every node u. For this, the whole surface area $A(P(u))$ of all polygons lying in $B(u)$ has to be computed for every node u. This cannot be done within acceptable time in the preprocessing phase because only a small part of polygons can be stored in the main memory. Many hard disk accesses would be necessary. In order to construct the sample tree in a bottom-up manner, for every node u the set $P(u)$ has to be known. This information can be obtained by constructing an octree. In a second step we build up the sample tree from this precomputed octree and store it on hard disk. This procedure has to be performed only once in a preprocessing phase.

5.1 Octree Construction

Because of the unknown spatial distribution of the polygons, the depth of the octree cannot be determined in advance in order to guarantee that every leaf node stores not more than a fixed number of polygons.

The main problem is that on the one hand one cannot store every node of the sample tree (and of the octree) in a single file. During the walkthrough too many files would have to be opened and closed which would result in a high response time of the node server. On the other hand one cannot store all nodes with their polygon data 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 after each other.

Because of this problem, first of all an octree with a fixed depth is constructed in two steps: 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.

Before constructing the octree, one can define the maximal number of polygons to be stored in the leaf nodes. If after the described construction of the octree a leaf node v contains more polygons than this maximal number, the node v is subdivided into subnodes until all leaf nodes do not exceed the fixed number.

5.2 Sample Tree Construction

The sample tree is constructed bottom-up from the octree. Let the subtrees of any node u be sample trees. In order to construct a sample tree for u, which differs a little from the definition of section 4, but still satisfies the desired sample tree property, one can consider the following: The child nodes v_i for $i = 1 \ldots 8$ store exactly the polygons that give a good approximation of $B(v_i)$ if the projected size of $B(v_i)$ is at most one pixel. Therefore, it is sufficient to sample only from polygons that are stored in the direct child nodes v_i and store the sampled polygons in u in order to get a good approximation for u if its projected bounding box is at most one pixel. 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 highly complex scenes externally stored on hard disk, the following requirements have to be considered: 1. To compute a new image, only a few nodes have to be loaded. 2. Loading a node may take only a very short time. The appropriate algorithm is described in this section. In the following, we assume that the viewpoint moves only a little bit from one frame to the next. Also, we assume that the samples of polygons needed for the computation of one frame can be stored in the main memory. This is not a restriction in practice 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 managed by the node manager. The client is connected to the server via TCP/IP and serves as a rendering workstation. Server and client can either be started on a single computer to store the scenes on local hard disk or on different computers to store the scenes remotely.

The traversal of the sample tree and the rendering of polygons are executed on the client. At the beginning, obviously 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.

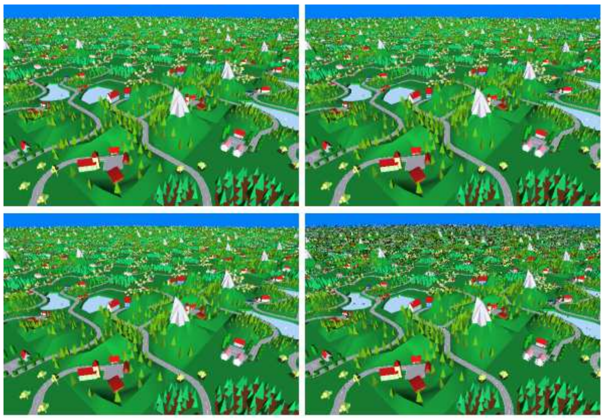

Figure 5: Images rendered by our method with different splat sizes in comparison to the z-buffer algorithm (upper left). Upper right: splat size 1, lower left: splat size 5, lower right: splat size 10.

The received root node is stored in the main memory. While the traversal has not stopped at a node u, and the required child node is not stored in the main memory, the client sends requests to the server and stores the received nodes in the memory. Of course, one cannot manage all nodes in the main memory at the same time. For this, we developed a caching mechanism which deletes nodes from the 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 for a node u stored in the k-th level of the sample tree, then all child nodes of the next j levels are kept in the 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 the memory if they are present. 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 by this caching mechanism, only very few nodes have to be loaded from one frame to the next. Additionally, only samples of polygons are stored in the nodes resulting in 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 the capacity on hard disk. All tests were performed on a linux based system with a Pentium 4 using 1GB of main memory, nVIDIA GeForce3 graphics card and a 40 GB hard disk.

Basic idea of our approach: First of all we present how our algorithm works. Figure 7 shows an image of a scene rendered by our method using a splat size of 4. We use no precomputed color values for better illustration. We zoomed into two parts of the image, which show that only samples of polygons are rendered. One can see that the objects are incomplete. Below the two zoomed pictures,

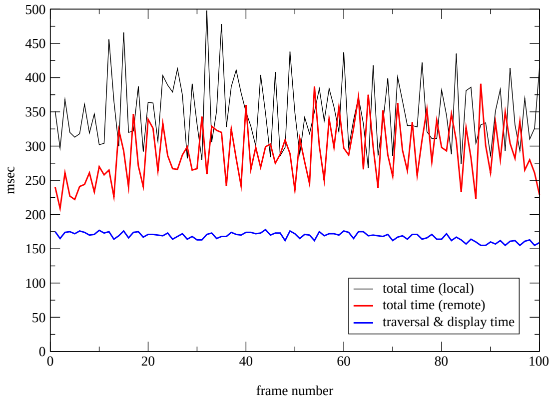

Figure 6: Rendering time for a scene of 95 millions of polygons stored on local and remote hard disk (resolution of $300 \times 300 pixels$). The total time includes the communication between server and client.

the corresponding complete parts of the original scene rendered by the z-buffer algorithm are presented for comparison. Observe that the size of the sample of the area close to the viewpoint is larger than that of the area far away.

Image quality: Figure 5 compares the quality of images computed by our method with an image computed by the z-buffer algorithm depending on different splat sizes. As described in section 4.5 for splats of size $i > 1$, precomputed color values are rendered as splats of size i^2 pixels instead of the polygons lying in nodes where the traversal stops. That reduces the rendering time and ensures that there are no holes in the image. 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 below (left: splat size 5, right: splat size 10) the quality is a little bit lower, one can see some false pixels.

In every node of our sample tree we stored a maximal number of $m = q_u \cdot \ln q_u + c \cdot q_u$ polygons, where $c = 5$ is chosen. According to theorem 1 (section 4.3) a correct part of the image is computed for every cell $B(u)$ with probability $p \approx 0.993$ (for splat size 1).

Rendering time: Figure 6 shows the rendering time for rendering our largest scene consisting of 95 millions of polygons (16 GB) stored on a local or remote hard disk. We chose a splat size of 4 so that we get images of a good quality. A path through the scene was fixed so that as many polygons as possible are in the viewing cone. The time for the traversal of our sample tree plus the display time (time used to send the polygons to the graphics card) amount to 168 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 349 msec in the average case. The difference between these two time values is the communication time between client and server.

The total time for remote hard disk connected to the client by a fast network (100 MBit) took only 290 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.

Of course, we do not use an instantiation scheme which would keep many polygons in the main memory.

Instantiation of objects: An instantiation scheme allows us to generate a scene consisting of more than 400 millions of polygons that can be stored completely in the main memory. The higher the splat size is chosen, the fewer nodes are visited in the sample tree and the shorter is the rendering time. The z-buffer algorithm needs more than ten minutes for the computation of one image while our algorithm (splat size 1) needs only 5.4 sec for an image of the same quality. With a splat size of 5, the rendering time amounts to 360 msec and with splat size 10 rendering takes 97 msec for images of acceptable quality. The rendering times are average values for a path of 100 steps through the scene.

8 Conclusion and Future Work

We developed a method that allows an interactive walkthrough in scenes of arbitrary topology that cannot be stored in the main memory, but only 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 [23] in order to specify the samples during the navigation. Furthermore, our randomized sample tree makes it possible to load only very few polygon data from hard disk if the viewpoint moves only a little bit from one frame to the next. So acceptable rendering times can be realized.

We also showed that with arbitrarily high probability, correct images can be computed. The quality of our images is comparable to those of the conventional z-buffer algorithm independent of the viewpoint and the scene topology.

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

Anti-Aliasing: Our precomputation of color values (see section 4.5) 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 useful to implement an occlusion-culling algorithm. It is conceivable to use a randomized visibility test for that.

Network Based Rendering: Our method allows an interactive navigation in highly complex scenes stored on remote hard disk only if the bandwidth of the network is good. Therefore, an adjustment of the approximation quality to the bandwidth of the network might be a good idea.

Acknowledgements

The authors wish to thank Michael Wand, Birgitta Derenthal, and Hanne Gräßle for valuable discussion and their 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).

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, Salt Lake City, 1974.

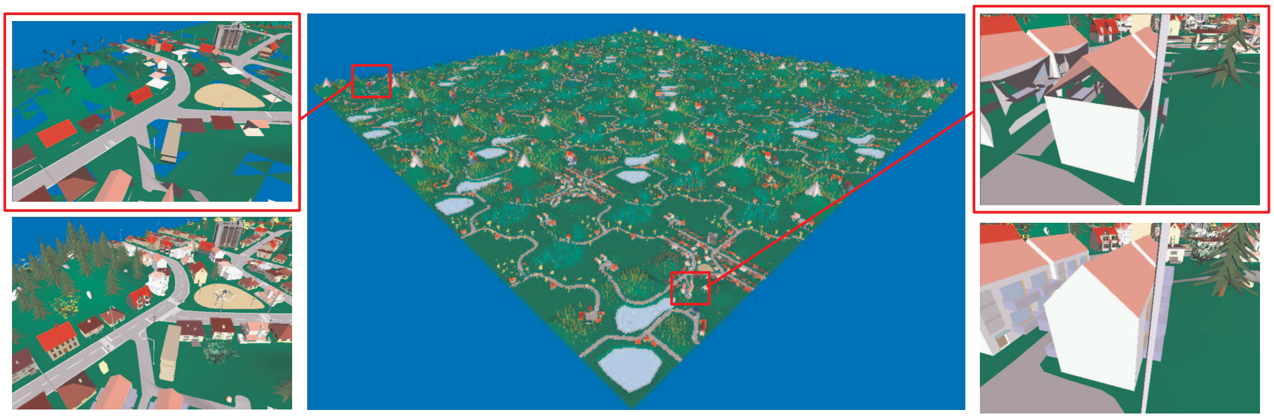

Figure 7: Two parts of the image are zoomed and show that only samples of polygons are rendered. Below, the corresponding parts of the original scene are rendered by the z-buffer algorithm.

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

[6] 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.

[7] 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, 209, 1995.

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

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

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

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

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

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

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