Volume 0 (2012), Number 0 pp. 1–14 COMPUTER GRAPHICS *forum*

Interactive Ray Tracing of Large Models Using Voxel Hierarchies

Attila T. Áfra[1,2]

[1]Budapest University of Technology and Economics, Hungary
[2]Babeş-Bolyai University, Cluj-Napoca, Romania
attila.afra@gmail.com

Abstract

We propose an efficient approach for interactive visualization of massive models with CPU ray tracing. A voxel-based hierarchical level-of-detail (LOD) framework is employed to minimize rendering time and required system memory. In a preprocessing phase, a compressed out-of-core data structure is constructed, which contains the original primitives of the model and the LOD voxels, organized into a kd-tree. During rendering, data is loaded asynchronously to ensure a smooth inspection of the model regardless of the available I/O bandwidth. With our technique, we are able to explore data sets consisting of hundreds of millions of triangles in real-time on a desktop PC with a quad-core CPU.

Categories and Subject Descriptors (according to ACM CCS): Three-Dimensional Graphics and Realism [I.3.7]: Raytracing—Methodology and Techniques [I.3.6]: Graphics data structures and data types—

1. Introduction

Real-time visualization and inspection of highly complex 3D models are required in many scientific, engineering, and entertainment domains. Examples of such domains include, among others, computer-aided design (CAD), 3D scanning, numerical simulation, virtual reality, and video games. Many massive models consist of hundreds of millions of primitives (e.g., triangles), occupying tens of gigabytes of space. Even though processor performance and memory capacity are rapidly increasing, the exploration of such immense data sets can still be problematic on a single commodity PC.

Real-time ray tracing has become an active research area in the past few years, mostly thanks to the emergence of affordable, high-performance parallel processor architectures like multi-core CPUs and programmable GPUs. This rendering technique can be easily parallelized and enables the precise simulation of optical phenomena such as shadows, reflections, and refraction. It is a significantly more versatile approach than rasterization, the currently most popular real-time rendering algorithm.

This paper presents a new massive model rendering method based on ray tracing, efficiently combining the advantages and techniques of different existing approaches. The most notable of these are the *R-LODs* [YLM06] and *Far Voxels* [GM05] methods. Some of the details of the system were introduced in [Áfr10].

Several testing examples demonstrate that our method works effectively for different types of complex models, achieving interactive frame rates on a quad-core desktop PC. It supports a wide variety of ray traced shading algorithms, which include direct lighting with shadows, ambient occlusion, and global illumination (see Figures 1 and 3).

2. Previous Work

In this section, we briefly discuss the methods most closely related to our work. For a comprehensive overview of the topic of large model rendering, we refer the reader to [YGKM08].

QSplat [RL00, RL01] is a point splatting technique which uses a bounding sphere hierarchy for level-of-detail (LOD) rendering with visibility culling. The model representation is compressed by quantizing the nodes of the hierarchy. The QSplat algorithm works well for laser-scanned models, but is not suitable for architectural or CAD models.

Layered Point Clouds [GM04] handle large point-based

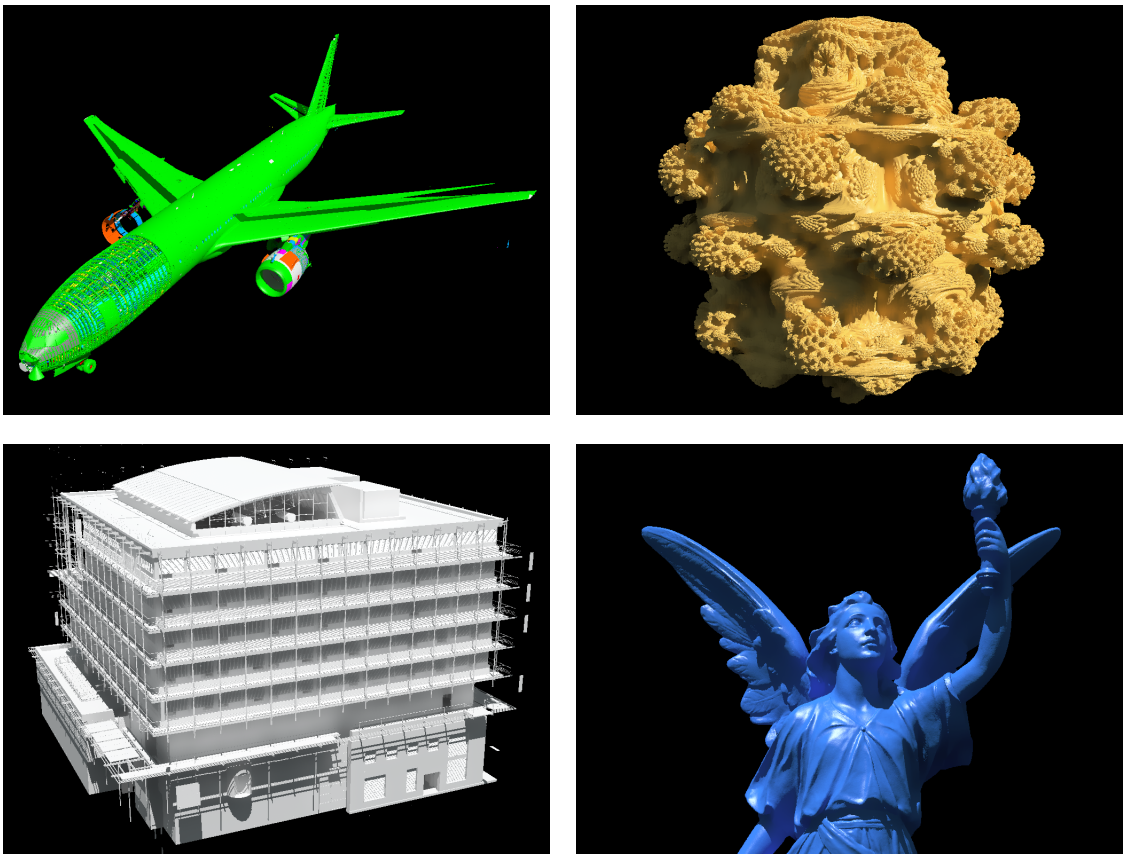


Figure 1: *Large models rendered with the proposed method: Boeing 777 (337M triangles), Mandelbulb (354M triangles), MPI v1.0 (73M triangles), and Lucy (28M triangles). All images were rendered interactively with shadows and one bounce of indirect illumination at* 1024×768 *resolution and 3 pixels of error. The processor used was an Intel Core i7-2600 quad-core desktop CPU. The scenes are lit with two light sources: a point light and a hemispherical environment light.*

models by organizing the point primitives into a multiresolution hierarchy. The points are grouped into clusters of approximately constant size to improve the efficiency of CPU/GPU communication, which is crucial for optimal rendering performance.

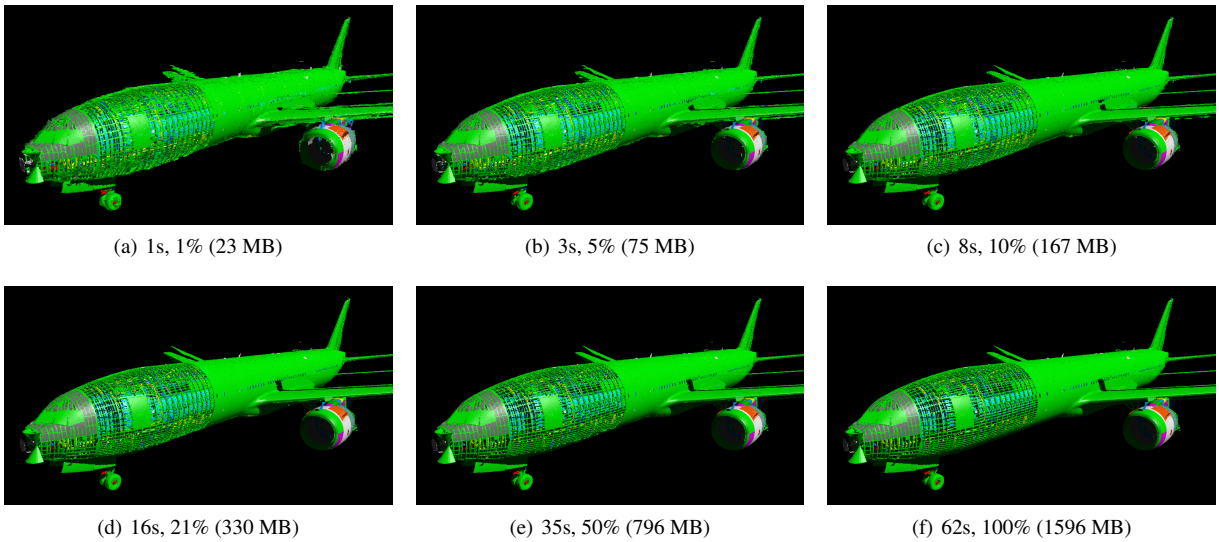
The Far Voxels [GM05] algorithm renders massive polygonal models by employing a LOD framework based on cubical view-dependent voxels, uses asynchronous I/O, and is optimized for GPUs. It renders voxels by splatting, which has the disadvantage that the approximation quality for complex CAD models can be low if the needed data has not been entirely loaded yet. The achieved frame rates are high, but only relatively simple shading is supported.

Quick-VDR [YSGM04] uses a clustered hierarchy of progressive meshes for out-of-core rendering and occlusion culling. Without geomorphing, popping artifacts may occur when switching between different LODs.

Wald *et al.* [WDS04] presented a ray tracing method to visualize the Boeing 777 model in real-time. The I/O is asynchronous to avoid data access latencies, and the unavailable geometry is represented with a small number of volumetric proxies. This is not a full LOD solution, therefore, the ray traversal depth and the working set size are not reduced.

The ray tracing based algorithm proposed by Yoon *et al.* [YLM06] uses drastic simplifications, called R-LODs, to improve rendering performance and to minimize the amount of required memory. The R-LODs are tightly integrated with a kd-tree used as an acceleration structure, and they consist of simple planes bounded by tree nodes, which have limited approximation capability. The latency caused by the data loading is not hidden, because the approach uses *memory mapping* to access the out-of-core data structure.

Other approaches use LOD-less compact in-core representations [LYTM08, SE10], which have reduced memory requirements, but the size of the scene is still limited by the amount of available memory. Also, ray coherency tech-



(a) 1s, 1% (23 MB) (b) 3s, 5% (75 MB) (c) 8s, 10% (167 MB)

(d) 16s, 21% (330 MB) (e) 35s, 50% (796 MB) (f) 62s, 100% (1596 MB)

Figure 2: *The model data is loaded asynchronously in the background. Although this may cause temporary blocking artifacts, the exploration remains fluid. This figure shows this effect in case of the Boeing 777 model. The labels indicate the elapsed time from the beginning of the loading process, the loading progress percentage, and the size of the working set.*

niques [Wal04, RSH05, ORM08, GR08] have low applicability in such cases because of the lack of simplification.

Although several massive model visualization techniques exist, many of them perform adequately only for specific types of models. This is one of the remarkable deficiencies of previous approaches, which are addressed by our proposed method.

3. Method Overview

The main goal is to seamlessly explore massive models, consisting of possibly hundreds of millions of triangles, on a single commodity PC. In order to achieve this, we first construct a *hierarchical out-of-core data structure* (Section 4), which contains, in a compressed format, the original triangles and several LOD levels consisting of voxels. These levels correspond to simplified versions of the data set at different resolutions. The reason we have decided to use voxels for model simplification is because they are suitable for geometry with very complex topology, comprising of many detailed, loosely connected, interweaving parts (e.g., pipes and wires) [GM05, CNLE09, LK10].

Thanks to the hierarchical LOD mechanism, it is possible to render huge data sets that cannot be completely loaded into the system memory. During rendering, we load the necessary details *asynchronously*, thus, there is no stuttering due to insufficient available data (see Figure 2). Also, the exploration starts immediately, without any loading time.

The choice of the ray tracing acceleration structure is very important, as it directly affects the rendering speed and the size of the working set. There are many factors that should be considered, including the hardware architecture, the theoretical capabilities of the construction and traversal algorithms, the type of rays, and the characteristics of the scenes [SKHBS02].

We organize all primitives (i.e., the triangles and voxels) into a *kd-tree*, a simple and efficient acceleration structure. The kd-tree is essentially a binary space partitioning (BSP) tree in which every non-leaf node divides the space into two subspaces with an axis-aligned plane, and the leaf nodes contain references to primitives. It is considered one of the best performing acceleration structures for ray tracing static scenes [Hav00], especially massive models [YGKM08], on the CPU.

This out-of-core kd-tree has a dual purpose in our approach: it speeds up the ray intersections with the triangles, and stores the voxel hierarchy. The kd-tree is a data structure that yields good performance for *both* tasks.

A subset of the kd-tree nodes contain a single *LOD voxel* (Section 4.2), which is a primitive rendered as an axis-aligned box. It roughly approximates the original primitives (i.e., triangles) stored in the subtree of the corresponding node and holds *shading attributes* (e.g., normal, color) per box face. The voxel completely fills the space subtended by the node, and thus it is not necessarily cubic in shape.

The entire kd-tree is decomposed into *treelets* (Section 4.3), which are grouped into equally sized *blocks*. In order

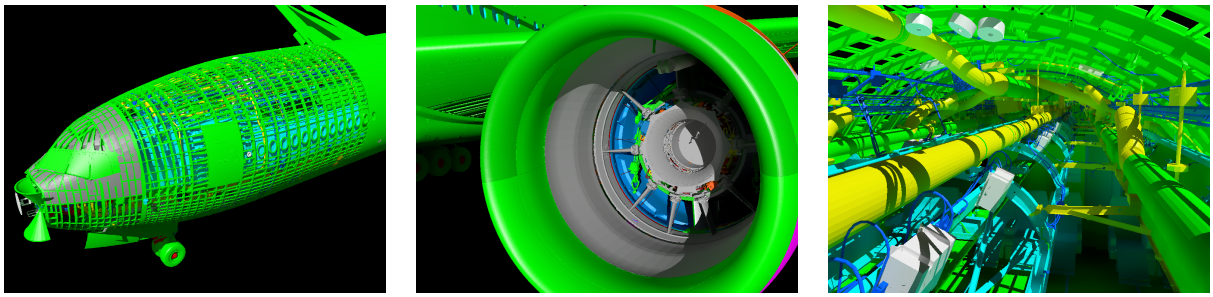


Figure 3: *The Boeing 777 model (337M triangles) rendered in real-time with shadows and one-bounce indirect illumination.*

to reduce storage requirements, the blocks are encoded using a lossless data compression algorithm (Section 4.7).

We employ a custom, purely software-based *memory manager* (Section 5), which is responsible for the loading of the blocks required by the renderer. We have designed the out-of-core data structure with a software solution in mind, and as a consequence, the software address translation overhead is greatly reduced. On the other hand, the application of a custom memory management mechanism enables us to perform superior block caching and on-the-fly decompression.

The tight integration of the LOD levels with the acceleration structure enables an efficient model representation and ray traversal algorithm. One of the advantages of our approach is that existing traversal algorithms can be elegantly extended to support the proposed voxel hierarchy with small overhead (Section 6.1). To exploit the high coherency of primary and shadow rays, we trace them in packets.

By using LOD voxels, significantly higher frame rates can be achieved, with minimal loss of image quality, because ray traversals are less deep, memory accesses are more coherent, and intersections with voxels are free, contrary to triangles (the voxel fills its parent node, therefore, the intersection is equal to the already computed intersection with the node). Furthermore, the LOD framework can also reduce the amount of aliasing artifacts, especially in case of highly tessellated models. We provide fast LOD error metrics for primary, shadow, ambient occlusion, and diffuse interreflection rays (Section 6.2).

4. Out-of-Core Data Structure

In this section we present the format of our compressed data structure used for LOD-based ray tracing, and an out-of-core algorithm for its construction from the full resolution model.

The entire data structure is divided into 64 KB blocks, which allows simple and efficient memory management. These are similar to memory pages handled by the CPU and the operating system, which commonly have a size of 4 KB.

An important benefit of custom memory management is the possibility to store the blocks on the hard disk in a compressed form and decompress them in real-time. This not only reduces storage requirements, but may also increase loading performance because less disk operations are necessary.

4.1. Kd-tree building

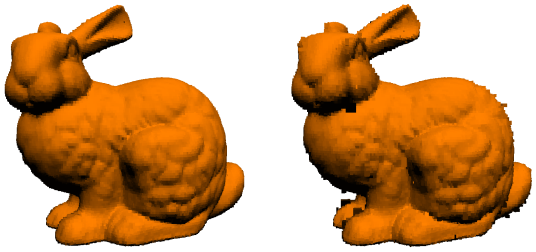
The main part of the construction process is the building of the kd-tree, which consists of the recursive spatial partitioning of the scene using axis-aligned planes. The inner nodes of the kd-tree contain an axis-aligned splitting plane, and the leaf nodes refer to one or more (up to 128) triangles.

We assume that the input for the construction algorithm is stored as a *triangle soup* (i.e., a simple list of triangles without shared vertex data) in a file. First, we calculate the axis-aligned bounding box of the model, then we proceed with the building of the kd-tree nodes, in depth-first order.

If all primitives belonging to a node can fit into the memory, we determine the splitting plane using the well-known *surface area heuristic* (SAH), which produces high quality results [Hav00]. Otherwise, we split the node in the middle (i.e., the spatial median), in an out-of-core fashion. This operation takes as input a triangle list stored in a file stream, and produces two new streams corresponding to the children of the respective node.

The sifting of the triangles into two sublists can be achieved in only a single sweep over the input file stream. This is an important property of the method because disk I/O operations are very time-consuming. Although splitting in the middle is almost always inferior to the surface area heuristic, only a relatively small amount of nodes close to the root are required to be built this way.

We stop subdividing and create a leaf node if the SAH cannot find a beneficial split, or if the number of triangles in the node is one. However, if the SAH fails to choose a splitting plane, and the triangle count is above a predefined threshold (e.g., 128), we split the node in the middle and continue the subdivision.



(a) *With* empty space cutting (b) *Without* empty space cutting

Figure 4: *The highest resolution voxel-based LOD level of the Bunny model constructed with and without empty space cutting. The image on the right shows that using solely the SAH does not result in satisfactory approximation quality.*

To produce a tight voxel-based approximation of the original geometry, the kd-tree cells, which bound the voxels, must not contain large empty spaces. The SAH alone is capable of cutting off empty space, but it can still produce many unnecessarily large cells (see Figure 4). We solve this problem by additionally employing an aggressive *empty space cutting* strategy [Hav00, RSH05]. Before selecting a splitting plane using one of the mentioned methods or making the node a leaf, we check whether we can create an empty child node that has a volume respective to the original cell larger then a specified threshold (we used values between 10–20%). This approach not only improves LOD quality, but also increases ray tracing performance. We use it for both SAH-based and spatial median splits.

4.2. LOD voxels

The size of a LOD voxel is identical to the size of the kd-tree node in which it is located. This means that the only data which must be stored to represent a voxel are the shading attributes.

In our implementation, we use two types of shading attributes: *normal* and *color*. A set of shading attributes constitute a *shading attribute sample*. When creating a voxel, we compute shading attribute samples for each of its six sides. We average these samples, and if the maximum absolute difference between the samples and the average is below a threshold, we store only the average sample in the voxel (*1-sample voxel*). Otherwise, we store separate samples for the six sides (*6-sample voxel*).

Creating and storing voxels in every kd-tree node is very expensive, therefore, we compute voxels only for a subset of the nodes. We significantly reduce the amount of voxels by storing them only in inner nodes. Since a kd-tree node partitions space only in one dimension, it is adequate to select only every third inner node on the path from the root to a leaf. This not only reduces memory requirements, but also improves ray traversal performance because the LOD error metric must be evaluated only for nodes that contain voxels. Note that it is not essential to enforce having splits along all three axes between two consecutive voxels on a path.

If we have to build a node out-of-core, we combine the computation of the voxel shading attribute samples with the triangle sifting process. We rasterize the triangles onto six image planes corresponding to the sides of the respective voxel. Depth testing is performed to take into account occlusion. After processing all triangles, we average the unoccluded samples. When building a subtree in-core, we employ adaptive Monte Carlo ray sampling. This can be implemented very efficiently by using the prebuilt kd-subtree rooted at the current node to accelerate the ray traversal.

We lossily compress the normals into 15 bits by storing them as points on a cube [LK10]. We encode the face of the cube (1-bit *sign* and 2-bit *axis*) and two coordinates on the face (6-bit signed integers u and v). The colors are stored in 15-bit RGB format (5 bits per channel). Using these encodings, a shading attribute sample can be packed into a 32-bit integer (Table 1). One of the 2 remaining bits is used to indicate whether the sample belongs to a 1-sample or a 6-sample voxel.

The total size of a 1-sample voxel is 4 bytes, and thus a 6-sample voxel is 24 bytes.

Bits	Value
0–4	color red
5–9	color green
10–14	color blue
15	normal sign
16–17	normal axis
18–23	normal u
24–29	normal v
30	*unused*
31	1-sample voxel flag

Table 1: *The memory layout of 4-byte voxel shading attribute samples.*

4.3. Treelets

We decompose the kd-tree and all related data into treelets which are small subtrees with a fixed maximum height of 3. See Figure 5 for an example. We store the treelets in breadth-first order and the nodes inside the treelets in preorder. The treelets are packed into blocks, a block containing multiple consecutive treelets from the same level (Figure 6).

As there are voxels only in every third tree level (with the exception of leaf nodes), treelets that have roots at the same depth constitute a LOD level. According to this treelet layout, the LOD levels are stored continuously and consecutively, starting with the lowest level-of-detail.

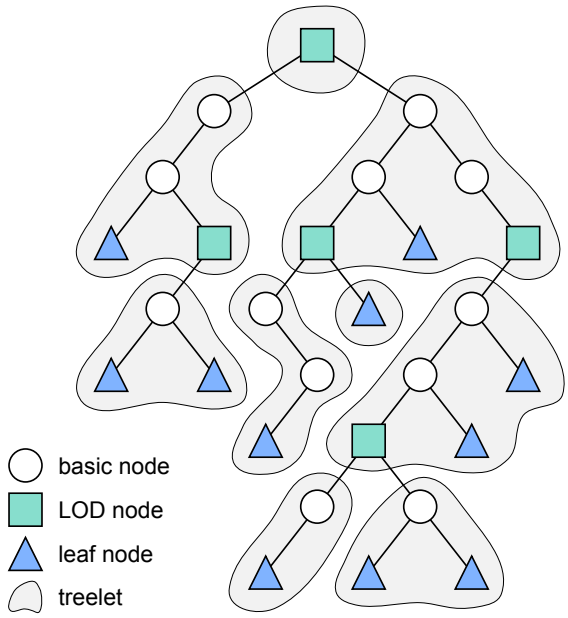


Figure 5: *The treelet decomposition of an out-of-core kd-tree.*

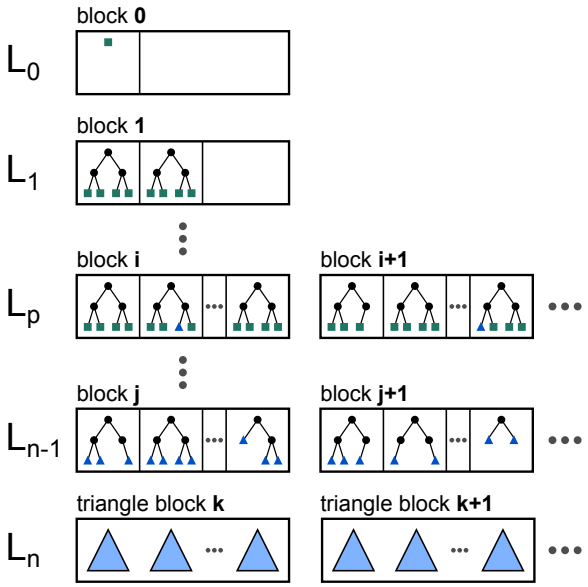


Figure 6: *The block layout of the out-of-core data structure. The blocks are grouped into LOD levels (L). The treelets, including the voxels, are stored in breadth-first order in levels* 0 *to* $n-1$*, whereas the triangles are in level n, the last level.*

A treelet consists of different types of kd-tree nodes, which may refer to additional data. Every treelet must have at least one inner node containing a LOD voxel (called a *LOD node*) or a leaf node containing triangles.

To encode the nodes in a highly compact way, variable size treelet nodes are used. The size of a node is either 8 or 16 bytes, so a cache-friendly memory alignment is possible. Also, the bit representation is carefully designed to minimize the number of operations required to unpack the node data.

A notable property of the ordering of the nodes is that a node is always stored at an address lower than that of its left child, which in turn has an address lower than the right one. This enables us to reference child nodes with strictly positive offsets.

4.4. Basic nodes

We call inner nodes that do not contain voxels *basic nodes*. These have children located in the same treelet. In addition to the type of the node and the splitting plane, child addresses are also encoded. The amount of required bits is reduced by storing the offset of the left child from the parent and the offset of the right child from the left one. These offsets are strictly positive and are multiples of 4.

A significant amount of the child nodes are empty leaves, which should not be stored to save memory. An efficient solution is to encode bits in the inner nodes that indicate whether a child node is empty or not.

Basic nodes can be fit into 8 bytes as shown in Table 2. Note that this is also the size of the compact in-core node representation proposed by Wald [Wal04].

Bits	Value
0–1	split axis
2–15	left child offset from this node
16	left child empty flag
17–20	right child offset from left child
21	right child empty flag
22–28	*unused*
29	voxel flag (0)
30	*unused*
31	inner node flag (1)
32–63	split position

Table 2: *The memory layout of 8-byte basic nodes.*

4.5. LOD nodes

An inner node that incorporates a voxel is called a *LOD node*. It is a treelet leaf, which points to roots of other treelets that are located in a different block. We encode a child address as a 32-bit *block ID* and an offset. This way, the size of the addressable memory space is 256 TB. In order to store only one block ID, the two child treelets are restricted to be in the same block.

During rendering, voxels are significantly less frequently accessed than nodes. A cache-efficient layout is to store the voxels separately from the nodes. In this case, a node must also contain an offset to its voxel.

When the ray tracer accesses a LOD node, it has to compute a LOD error value to decide whether to continue with the traversal or not. This error value depends, among others, on the size of the voxel, the on-the-fly calculation of which is costly. Therefore, the size is stored in the tree, encoded as the radius of the sphere enclosing the respective voxel. This radius is kept in the node data structure because it is a frequently accessed value. Thus, the size of LOD nodes excluding the voxel data is 16 bytes (see Table 3).

Bits	Value
0–1	split axis
2–15	left child offset from this node
16	left child empty flag
17–20	right child offset from left child
21	right child empty flag
22–28	voxel offset from this node
29	voxel flag (1)
30	*unused*
31	inner node flag (1)
32–63	split position
64–95	enclosing sphere radius
96–127	children block ID

Table 3: *The memory layout of 16-byte LOD nodes.*

4.6. Leaf nodes

The leaf nodes in a kd-tree contain a subset of the original triangles, and, usually, many of these triangles are shared by multiple nodes. Because of this, it is not efficient to store the triangles themselves in each node.

In case of simple in-core kd-tree representations, it is common practice to store each triangle only once in a global array, and create triangle lists that belong to the leaf nodes which contain pointers to the triangles. The nodes, triangles lists, and triangles are put into different buffers.

For our out-of-core data structure, we take a similar approach. However, instead of global buffers, we create *triangle blocks*, each divided into a *triangle list section*, a *triangle section*, and a *vertex section*. These blocks represent the highest level-of-detail, and we output them after the other blocks (see Figure 6).

The triangle list section contains a series of 16-bit offsets that point to 12-byte triangle structures (with color). Usually, the model is a mesh, so a vertex may belong to more than one triangle. To further minimize the storage requirements, the unique vertices are stored in the vertex section, and the triangle structures contain three offsets to one of these vertices. The shared vertices are determined by using a hash table. If the vertices have only a position attribute, they are encoded in 12 bytes.

A leaf node is a simple 8 byte structure (see Table 4), which contains a reference to a triangle list and the number of triangles in that list. The list reference is expressed as a block ID and an offset from the beginning of the block.

Bits	Value
0–15	triangle list offset
16–23	triangle count
24–30	*unused*
31	inner node flag (0)
32–63	triangle block ID

Table 4: *The memory layout of 8-byte leaf nodes.*

4.7. Compression

On-the-fly decompression should not degrade significantly the rendering performance, thus we apply an LZ77 family algorithm [ZL77], which provides very fast decompression speed. It compresses data losslessly by replacing previously encountered strings with offset and length pairs. Therefore, decompression basically consists of a series of string copying operations. We use a byte-aligned encoding method to minimize decompression complexity.

The decompression speed on a single core of an Intel Core i7-2600 processor is more than 700 MB/s.

5. Memory Management

The previously described out-of-core model data structure is accessed by the renderer through a custom memory manager, which can be implemented entirely in software. This design enables us to tailor and fine tune the data loading and caching for our needs and, at the same time, hide the complexity of the memory management from the ray tracer.

The memory manager operates on the granularity of blocks, which means that the smallest unit of data that can be loaded and cached is a block. This is very similar to the way paged virtual memory works, which is implemented by the operating system and requires hardware support, typically in the form a memory management unit (MMU) built into the CPU. If there are no page faults, virtual memory accesses have very small overhead thanks to the hardware address translation performed by the MMU.

Most operating systems featuring virtual memory (e.g., Windows, Linux) have support for *memory-mapped files*, which enable efficient file I/O entirely through memory accesses. Unfortunately, this facility has a few significant limitations that negatively impact some usage scenarios.

From our point of view, the most significant deficiencies of memory mapping are the lack of native, cross-platform

asynchronous I/O and on-the-fly decompression. Note that the Linux kernel has special memory mapping functions (*mincore* and *madvise*) that can be used to manually implement a limited form of asynchronous data access [WDS04], but this approach does not support compression and is heavily operating system dependent. Thus, we have opted for a purely software memory management system.

Since we want to exploit the power of multi-core and multi-processor systems, the memory manager must support multiple concurrent renderer threads. The amount of synchronization between the threads should be minimized. We accomplish this by synchronizing only between rendering two consecutive frames.

Blocks required by the renderer threads are loaded and decompressed into a fixed-size cache by one or more separate *fetcher* threads. The cache consists of *slots*, and a simple table is used to store the mapping between block IDs and cache slot IDs.

We associate with each cache slot a *timestamp*, which indicates when the block stored in the respective slot was last accessed. This information is used to determine which blocks should be evicted from the cache if there are not enough free cache slots to load new blocks. The current timestamp is incremented after each synchronization.

5.1. Block requests

If a renderer thread wants to access data residing in a block, it requests the address of the block from the memory manager. First, the memory manager checks whether the block is loaded into the cache. If so, it updates the timestamp of the proper cache slot by replacing the old value with the present timestamp, and performs address translation by computing and returning the block pointer. Otherwise, it adds an entry to the *request list* of the renderer thread and returns a null pointer. In this case, the renderer should also specify a *data ID*, a value that identifies the data inside the block it wants to access. This value can be used by the memory manager to assign a better priority value to the block.

Updating a timestamp must be done carefully because multiple threads may attempt to change that value at the same time. For this, we exploit the *atomic store* operation of the CPU, which provides a safe and efficient *lock-free* [Fra04] solution.

The address translation, not being accelerated by an MMU, is a relatively costly operation, therefore, the renderer should not invoke it at every memory access. The ray traversal algorithm does not require frequent block requests, so the overhead is minimal.

5.2. Fetcher threads

The fetcher threads continuously load the blocks referenced in a *fetch list* into preallocated cache slots. After loading and decompressing a block, we do not immediately mark it as available. As a consequence, the set of available blocks is constant throughout the computation of a frame.

By running more than one fetcher thread, the disk read operations can be executed more efficiently. This is especially true if the data is compressed. While one fetcher thread is waiting for a compressed block to be read from the file, another thread can decompress an already loaded block. However, we must be careful not to run too many fetcher threads, which can degrade the rendering performance. Fortunately, a small amount of fetcher threads (e.g., 4) is usually sufficient to achieve satisfying loading speeds.

5.3. Cache update

After finishing the rendering of a frame, the renderer should request a cache update, which is a major thread synchronization point. During this entire process, the renderer threads may not request blocks.

As a first step, we stop all fetcher threads. Then, we mark all freshly loaded blocks as available and update a *least recently used* (LRU) list based on the timestamps of the cache slots. We gather the requests from the request lists and assign priority values to them. The higher a priority value is, the sooner the block will be loaded.

The priority of a block is the maximum of the number of requests per data ID per block. By taking into account the data IDs too, we can achieve that a block with even a single piece of outstandingly important data will be loaded sooner than a block with several less important ones.

In the next step, we generate block fetch jobs, sort them by their priority, and put them into the fetch list, discarding its previous contents. We limit the length of the fetch list based on the estimated maximum I/O bandwidth. This fetching approach is similar to the one incorporated into the Streaming QSplat algorithm [RL01].

Because blocks must not be removed from the cache while rendering, we preallocate cache slots for the blocks to be fetched. If the cache is full, we use the LRU list to evict blocks. We do not remove blocks that are part of the current working set to prevent cache thrashing.

Finally, we resume the fetcher threads in order to start loading and decompressing the blocks specified in the newly populated fetch list.

6. Ray Tracing

Once we have constructed the out-of-core data structure for a scene, we can render it using LOD-based ray tracing. As already mentioned, the ray tracer accesses the data structure through the custom memory manager.

To compute the color of a pixel, we cast a ray from the

camera viewpoint through the pixel and find its nearest intersection with the geometry. Then, we perform shading at the intersection point while optionally casting secondary rays.

Ray casting consists of two main operations: ray traversal and primitive intersection. For our rendering approach, the acceleration structure corresponds to the out-of-core kd-tree, and the primitives are of two types: triangles and voxels.

6.1. Ray traversal

We extend the kd-tree traversal algorithm proposed by Wald [Wal04] to support LOD voxels and asynchronous loading, as shown in Algorithm 1. During traversal, we additionally maintain the reference to the last encountered voxel on the current path. When we reach a LOD node, we update this reference and compute a LOD error value. If the error does not exceed a specified threshold, we return the intersection of the ray with the voxel and stop the traversal.

It may happen that we cannot continue with the traversal from the current node because the required data (child node or triangle list) is not yet loaded into memory. In this case, we return the intersection with the last encountered voxel. We always return the intersection distance to the current node, even if it does not contain a voxel. This means that we clip the last encountered voxel to the bounds of the current node.

Determining the distance to the intersection of the ray with a voxel is trivial. Since voxels are bounded by kd-tree nodes, the intersection distance is given by the lower bound of the current ray segment. However, getting the shading attributes at the intersection point may involve additional computations. If the voxel has only one sample, the solution is simple because the shading attributes are constant throughout its surface. But this is not the case if there are separate samples for the six sides of the voxel. Simple nearest-neighbor filtering is sufficient, therefore, we have to determine on which side of the voxel the intersection point is.

A side can be identified by an axis and a sign. The axis of an intersected voxel side cannot be determined implicitly, but the sign is always the opposite of the sign of the ray direction along the axis of the side. We solve this problem by maintaining in each traversal step the axis of the plane through which the ray enters the node, the *entry axis*. If we traverse the front child node, the entry axis does not change. However, if we traverse the back child, the entry axis becomes the axis of the splitting plane of the parent node.

We exploit the coherency of primary and shadow rays by tracing them in packets of 4×4 [Wal04, Ben06]. Ray coherency techniques like packet tracing have limited benefit for large models rendered at full resolution because coherence decreases with geometric complexity. However, LOD-based ray tracing does not have this downside, because traversals are terminated before they substantially diverge.

Algorithm 1 Single ray traversal

```
1:  hit.t ← ∞
2:  (ray.tMin, ray.tMax, entryAxis) ← IntersectAABB(ray, root)
3:  if IsRayInvalid(ray) then
4:      return hit
5:  node ← root
6:  voxel ← NULL
7:  loop
8:      loop
9:          if IsLeaf(node) then
10:             block ← GetBlockAddress(node.blockID)
11:             if block = NULL then
12:                 RequestBlock(node.blockID)
13:                 hit ← IntersectVoxel(ray, hit, voxel, entryAxis)
14:                 return hit
15:             triangles ← block + node.trianglesOffset
16:             hit ← IntersectTriangles(ray, hit, triangles)
17:             if hit.t ≤ ray.tMax then
18:                 return hit
19:             break
20:         if IsLOD(node) then
21:             voxel ← node + node.voxelOffset
22:             block ← GetBlockAddress(node.blockID)
23:             if block = NULL then
24:                 RequestBlock(node.blockID)
25:                 hit ← IntersectVoxel(ray, hit, voxel, entryAxis)
26:                 return hit
27:             if node.voxelRadius ≤ ray.tMin · C then
28:                 hit ← IntersectVoxel(ray, hit, voxel, entryAxis)
29:                 return hit
30:             nodeBase ← block
31:         else
32:             nodeBase ← node
33:         t ← node.splitPosition − ray.origin[node.splitAxis]
34:         t ← t / ray.direction[node.splitAxis]
35:         front ← Sign(ray.direction[node.splitAxis])
36:         back ← 1 - front
37:         if t > ray.tMax then
38:             if not node.childFlags[front] then
39:                 break
40:             node ← nodeBase + node.childOffsets[front]
41:         else if t < ray.tMin then
42:             if not node.childFlags[back] then
43:                 break
44:             node ← nodeBase + node.childOffsets[back]
45:         else
46:             if node.childFlags[back] then
47:                 backNode ← nodeBase + node.childOffsets[back]
48:                 PushState(backNode, voxel, t, ray.tMax, splitAxis)
49:             if not node.childFlags[front] then
50:                 break
51:             node ← nodeBase + node.childOffsets[front]
52:             ray.tMax ← t
53:     if IsStateStackEmpty() then
54:         return hit
55:     (node, voxel, ray.tMin, ray.tMax, entryAxis) ← PopState()
```

Model	Tris	Primary rays		Primary, shadow rays		Primary, shadow, diffuse rays	
		Speed	Working set	Speed	Working set	Speed	Working set
Asian Dragon	7M	47.6 fps	108 MB	33.4 fps	118 MB	6.4 fps	119 MB
Power Plant	12M	45 fps	62 MB	23.1 fps	74 MB	3.2 fps	98 MB
Lucy	28M	62 fps	143 MB	41.4 fps	145 MB	7.7 fps	148 MB
MPI v1.0	73M	39.8 fps	165 MB	26 fps	174 MB	4.1 fps	224 MB
Boeing 777	337M	36.6 fps	834 MB	24.5 fps	945 MB	6.6 fps	1019 MB
Mandelbulb	354M	28.2 fps	598 MB	16.4 fps	667 MB	3.5 fps	693 MB

Table 5: *Rendering speed and working set size measurements for the test models at* 1024×768 *resolution and 3 pixels of error. The listed values are averages from 4 representative views. The benchmarks have been performed on an Intel Core i7-2600 processor with three different rendering methods: direct illumination without shadows (i.e., ray casting), direct illumination with point-light shadows, and both direct and indirect illumination. We cast one shadow ray per primary or diffuse ray, and two random diffuse rays per primary ray. The diffuse rays are used to compute both one bounce of indirect irradiance and environment irradiance, which are processed with a bilateral filter [TM98] to eliminate noise.*

6.2. LOD error metric

For primary rays, we employ a simple and low-cost projected screen-space error metric. The error threshold is expressed as the maximum tolerated screen-space area of a projected voxel, which is sometimes known as *pixels of error* (PoE) [YSGM04, YLM06]. The higher the PoE value is, the less detailed but faster is the rendering. The quality degradation can be seen on Figure 9.

To simplify the calculations, we approximate the voxel with its enclosing sphere and compare its screen-space radius with a threshold. The value of this threshold is simply the radius ($\hat{R}_{\max}$) of the circle which area is equal to the PoE.

The estimated perspective projected screen-space radius ($\hat{R}$) of a voxel enclosed by a sphere with radius R can be written as:

$$\begin{aligned} \hat{R} &= \lambda \frac{R}{t_{\min}}, \\ \lambda &= \frac{w/2}{\tan(\phi/2)}, \end{aligned} \tag{1}$$

where $t_{\min}$ is the distance from the ray origin to the closest intersection with the voxel, and λ is a screen-dependent constant determined by the number of pixels w along the field of view ϕ. To efficiently check whether the error metric is satisfied, we evaluate the following rearranged inequality:

$$R \leq t_{\min} C, \tag{2}$$

where C is a global for all primary rays. This can be precomputed the following way:

$$C = \frac{\hat{R}_{\max}}{\lambda}. \tag{3}$$

Thus, only a multiplication and a comparison operation must be executed for each encountered voxel.

This error metric can be easily used for shadow rays too because shadows are projections, similar to perspective or orthogonal camera projections. It can also be adapted with a slight modification to ambient occlusion and diffuse interreflection rays. For such rays we specify the error threshold in steradians instead of pixels. As in [PFHA10], we assume that hemispherically sampled rays subtend on average a solid angle of $2\pi/n$ steradians, where n is the number of rays. An important limitation of our simple LOD error metric is that it does not work with refraction and non-planar reflection rays.

One of the common problems of ray tracing is to avoid self-intersections due to numerical imprecisions when casting secondary rays. This is usually solved by ignoring intersections closer than an epsilon threshold. For LOD-based ray tracing, we must additionally ignore intersections with the voxel selected by the previous ray. This can be achieved by increasing the threshold with the diameter of the voxel.

Model	Tris	Size	Compr. ratio	Build time
Asian Dragon	7M	0.4 GB	70%	5m
Power Plant	12M	0.5 GB	54%	7m
Lucy	28M	1.5 GB	68%	21m
MPI v1.0	73M	4.6 GB	58%	1h 3m
Boeing 777	337M	31.4 GB	66%	7h 43m
Mandelbulb	354M	7.5 GB	59%	1h 44m

Table 6: *Construction statistics for the test models: the number of triangles in the model, the size of the compressed out-of-core data structure, the compression ratio relative to the full uncompressed data structure (which includes the kd-tree, the voxels, and the triangles), and the build time (single-threaded).*

7. Results and Discussion

All benchmarks were performed on a desktop PC with an Intel Core i7-2600 (4 cores, 8 threads, 3.40 GHz) processor,

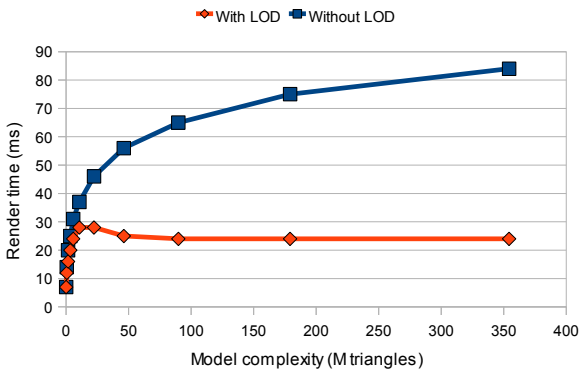


Figure 7: *The scaling of the ray casting performance with the model complexity. We have created simplified versions of the Mandelbulb model (354M triangles) and rendered them from the same viewpoint. We have measured the render time of a single frame with and without using LOD. Note that with LOD, the performance becomes nearly constant after a certain model complexity.*

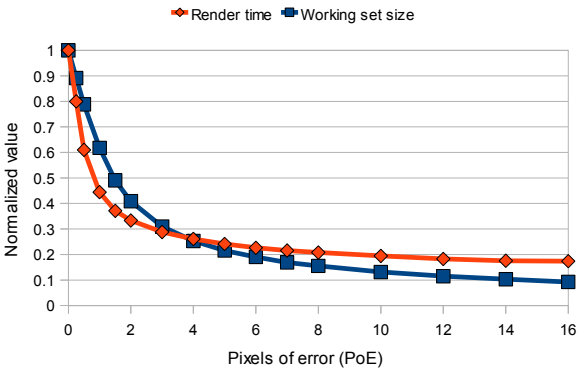


Figure 8: *The scaling of the ray casting performance with the pixels of error. We have rendered the Lucy model with varying error thresholds from the same viewpoint and measured the render time and working set size. The values shown in this figure are relative to the maximum measured values.*

8 GB RAM, and two 7200 RPM hard disks in RAID 0 setup. The system was running Ubuntu Linux 11.04 64-bit.

For our performance evaluations, we have selected test models from different application domains: Power Plant (12M triangles) is a CAD model of a coal-fired power plant; Asian Dragon (7M triangles) and Lucy (28M triangles) are high resolution laser-scanned statues; MPI v1.0 (73M triangles) is a virtual reconstruction of the Max Planck Institute for Informatics building [HZDS09]; Boeing 777 (337M triangles) is a highly complex CAD model of an airplane; Mandelbulb (354M triangles) is a fractal mesh obtained by isosurface extraction. Some of these data sets are shown in Figure 1. The source code for the Mandelbulb data set generator is freely available at `https://bitbucket.org/attila_afra/mandelbulbgen`.

The construction statistics for the models are listed in Table 6. We have designed our prototype implementation of the out-of-core construction phase for simplicity instead of performance. It is single-threaded, which means that it cannot leverage multiple CPU cores.

The majority of the construction involves building sub-kd-trees for in-core geometry chunks and sampling voxels. Both of these tasks can be considerably sped up through the use of multi-threaded algorithms. First, we can parallelize the kd-tree building by building a single tree at a time on multiple threads [CKL*10] or building more than one tree simultaneously, each on a single thread. Secondly, we can sample the voxels independently on parallel threads. However, the out-of-core splitting of the geometry cannot be improved using multi-threading because the performance is bound by the hard disk I/O bandwidth. Fortunately, this operation constitutes less than 5% of the total construction time.

The scaling of the ray casting performance with the number of triangles in the model is illustrated in Figure 7. Notice that without LOD, the render time increases logarithmically, as expected from employing a kd-tree as an acceleration structure. However, if we enable the use of LOD voxels, the performance becomes nearly constant as soon as the maximum projected triangle size drops below the pixel error threshold.

Figure 8 shows that by increasing the pixels of error, the render time and the size of the working set can be dramatically reduced, at the expense of image quality. According to our tests, a good trade-off between performance and quality can be achieved with about 2–3 PoE. For example, rendering with 3 PoE is typically about 50% faster than with 1 PoE, but the subjective image quality is only slightly inferior.

Our ray tracing based rendering approach allows flexible shading and lighting, ranging from simple direct illumination without shadows to global illumination. It can be seen in Table 5 that even the most complex models from the test suite can be rendered at interactive speeds with our approach. Thanks to the LOD mechanism, the required system memory is also kept within acceptable bounds.

Compression not only reduces hard disk space requirements, but also improves I/O performance. For the Asian Dragon model, which has a compression ratio of 70%, compression increases the loading speed by 18%.

7.1. Comparisons

R-LODs: The most similar method to ours is that of Yoon *et al.* [YLM06], which is also ray tracing based and uses

(a) PoE = 0 (b) PoE = 3 (c) PoE = 32

Figure 9: *Magnified renderings of the Lucy model at different LOD error thresholds.*

the same LOD error metric for primary and shadow rays as our algorithm. While in their approach the LOD primitives are simple planes with one color value each, which are called R-LODs, we have instead opted for voxels with varying amount of shading attribute sets. This way, we can better approximate complex volumetric details and avoid holes in the simplified versions of the model.

Another key difference is that their system does not load the data asynchronously, thus, it provides less smooth visualization than our method. Also, it was not designed to use compression. However, they employ a more efficient *cache-oblivious* data layout [YLPM05, YM06], which can lead to faster rendering and loading. The two approaches could be combined to obtain an even more powerful method.

Far Voxels: The approach introduced in [GM05] uses GPU rasterization instead of ray tracing, but similarly to our method, implements LOD with view-dependent voxels and does asynchronous I/O. Rendering massive models with rasterization has two significant advantages compared to ray tracing. First, for primary visibility, GPU rasterization with LOD can be much faster (nevertheless, without LOD, ray tracing is the clear winner for very large data sets). Secondly, it does not require deep hierarchical acceleration structures, which usually occupy huge amounts of space for massive models. However, it is more limited in features than ray tracing.

Although the Far Voxels method produces higher frame rates than our approach, it uses very simple shading (direct lighting without shadows). Another drawback is that it renders voxels with splatting instead of displaying them as 3D primitives, which can be distracting if the voxels occupy more than a few pixels. In exchange for higher computational and memory requirements, our ray tracing method offers better voxel rendering quality and advanced shading like true (not screen space) ambient occlusion and global illumination.

7.2. Limitations

Our method is able to handle a wide variety of model types and to compute ray-traced effects, but it has certain limitations. Most importantly, voxel-based LOD approximation can cause visual artifacts. Voxels with only six shading attribute samples may not provide high enough shading quality for some very complex parts of the model. A possible remedy for this problem is to adaptively increase the number of samples per voxel. Moreover, thin or small objects are difficult to represent with voxel hierarchies because the size of the voxels can be much larger than the original geometry. The resulting aliasing artifacts are especially apparent at lower levels of detail. These artifacts could be alleviated by using semi-transparent voxels [WRG07].

Another limitation is that our LOD error metric does not support non-planar reflection and refraction rays. Also, our approach is practical only for models larger than the system memory. For smaller models, the computation and storage overhead induced by the out-of-core representation may outweigh the benefits.

8. Conclusions and Future Work

We have presented a new efficient technique for interactive out-of-core rendering of massive polygonal models. The main contributions of this paper are:

- a compressed out-of-core model data structure for ray tracing, which stores triangles and voxel-based hierarchical LODs;
- an efficient memory management method, which supports prioritized asynchronous loading of details from disk;
- a LOD-based ray traversal algorithm suitable for primary, shadow, diffuse, ambient occlusion, and planar reflection rays.

Our approach can render hundreds of millions of triangles at interactive speeds on a desktop PC, even with shadows, ambient occlusion, and indirect illumination. The hierarchical LOD mechanism improves ray tracing speed and reduces the amount of required memory. Data reading and decompression are asynchronous to make it possible to inspect the model even if not all of the necessary details are loaded yet.

In the future we would like to improve the voxel approximation quality and to develop a more general and fast LOD error metric. We also plan to investigate possibilities for improving the performance of incoherent ray shooting. Another interesting research avenue is the adaptation of the rendering algorithm for GPUs.

Acknowledgements

This work was possible with the financial support of the Sectoral Operational Programme for Human Resources Development 2007-2013, co-financed by the European Social Fund, under the project number POSDRU/107/1.5/S/76841 with the title *Modern Doctoral Studies: Internationalization and Interdisciplinarity*. This work has also been supported by OTKA K-719922 (Hungary).

The Boeing 777 data set was provided by and used with permission of The Boeing Company. The MPI Informatics building model is courtesy of the Max Planck Institute for Informatics [HZDS09]. The Lucy, Asian Dragon, and Bunny models are courtesy of the 3D Scanning Repository at Stanford University.

References

[Áfr10] ÁFRA A. T.: Interactive out-of-core ray casting of massive triangular models with voxel-based LODs. In *Proceedings of the Fifth Hungarian Conference on Computer Graphics and Geometry* (Budapest, Hungary, 2010). 1

[Ben06] BENTHIN C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University, 2006. 9

[CKL*10] CHOI B., KOMURAVELLI R., LU V., SUNG H., BOCCHINO R. L., ADVE S. V., HART J. C.: Parallel SAH k-D tree construction. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, 2010), HPG '10, Eurographics Association, pp. 77–86. 11

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2009), I3D '09, ACM, pp. 15–22. 3

[Fra04] FRASER K.: *Practical Lock-Freedom*. Tech. Rep. UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004. 8

[GM04] GOBBETTI E., MARTON F.: Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics 28*, 6 (2004), 815 – 826. 1

[GM05] GOBBETTI E., MARTON F.: Far Voxels – a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Transactions on Graphics 24*, 3 (August 2005), 878–885. Proceedings of SIGGRAPH 2005. 1, 2, 3, 12

[GR08] GRIBBLE C. P., RAMANI K.: Coherent ray tracing via stream filtering. In *Proceedings of 2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (August 2008), pp. 59–66. 3

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 3, 4, 5

[HZDS09] HAVRAN V., ZAJAC J., DRAHOKOUPIL J., SEIDEL H.-P.: *MPI Informatics Building Model as Data for Your Research*. Research Report MPI-I-2009-4-004, MPI Informatik, Saarbruecken, Germany, December 2009. 11, 13

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, ACM, pp. 55–63. 3, 5

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: ReduceM: Interactive and memory efficient ray tracing of large models. *Computer Graphics Forum 27*, 4 (2008), 1313–1321. 2

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time Whitted ray tracing. In *Proceedings of IEEE/Eurographics Symposium on Interactive Ray Tracing 2008* (2008), pp. 41–48. 3

[PFHA10] PANTALEONI J., FASCIONE L., HILL M., AILA T.: PantaRay: Fast ray-traced occlusion caching of massive scenes. In *ACM SIGGRAPH 2010 Papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 37:1–37:10. 10

[RL00] RUSINKIEWICZ S., LEVOY M.: QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000* (July 2000), pp. 343–352. 1

[RL01] RUSINKIEWICZ S., LEVOY M.: Streaming QSplat: A viewer for networked visualization of large, dense models. In *I3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics* (New York, NY, USA, 2001), ACM, pp. 63–68. 1, 8

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM, pp. 1176–1185. 3, 5

[SE10] SEGOVIA B., ERNST M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Proceedings of Graphics Interface 2010* (2010), pp. 153–160. 2

[SKHBS02] SZIRMAY-KALOS L., HAVRAN V., BENEDEK B., SZÉCSI L.: On the efficiency of ray-shooting acceleration

schemes. In *Proceedings of Spring Conference on Computer Graphics (SCCG)* (2002), pp. 97–106. 3

[TM98] TOMASI C., MANDUCHI R.: Bilateral filtering for gray and color images. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 839–846. 10

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 3, 6, 9

[WDS04] WALD I., DIETRICH A., SLUSALLEK P.: An interactive out-of-core rendering framework for visualizing massively complex models. In *EGSR04: Proceedings of the 15th Eurographics Symposium on Rendering* (Aire-la-Ville, Switzerland, 2004), Eurographics Association, pp. 81–92. 2, 8

[WRG07] WAGNER G. N., RAPOSO A., GATTASS M.: An anti-aliasing technique for voxel-based massive model visualization strategies. In *Proceedings of the 3rd International Conference on Advances in Visual Computing - Volume Part I* (Berlin, Heidelberg, 2007), ISVC'07, Springer-Verlag, pp. 288–297. 12

[YGKM08] YOON S.-E., GOBBETTI E., KASIK D., MANOCHA D.: *Real-Time Massive Model Rendering*. Synthesis Lectures on Computer Graphics and Animation. Morgan & Claypool Publishers, San Rafael, CA, USA, 2008. 1, 3

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-LODs: Fast LOD-based ray tracing of massive models. *The Visual Computer: International Journal of Computer Graphics 22*, 9 (2006), 772–784. 1, 2, 10, 11

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-oblivious mesh layouts. *ACM Transactions on Graphics 24* (July 2005), 886–893. 12

[YM06] YOON S.-E., MANOCHA D.: Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum 25*, 3 (2006), 507–516. 12

[YSGM04] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick-VDR: Interactive view-dependent rendering of massive models. In *Proceedings of IEEE Visualization 2004* (2004), pp. 131–138. 2, 10

[ZL77] ZIV J., LEMPEL A.: A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23* (1977), 337–343. 7