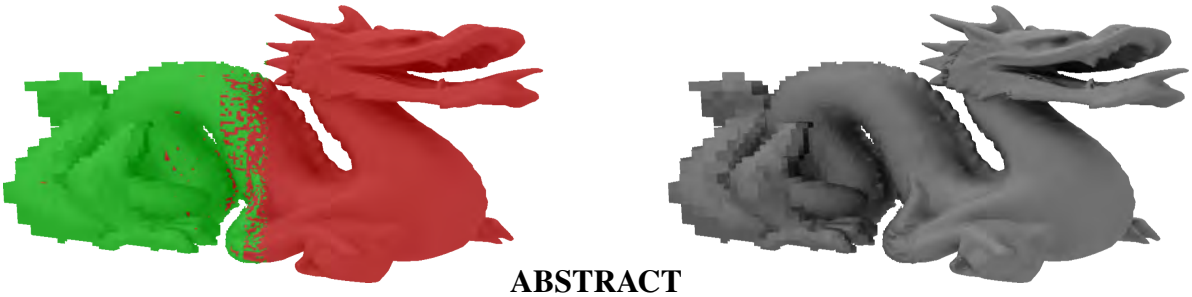
A Unified Triangle/Voxel Structure for GPUs and its Applications

Martin Weier
Institut of Visual Computing
Sankt Augustin
Grantham-Allee 20
53757, Sankt Augustin, Germany
Martin.Weier@h-brs.de

André Hinkenjann
Institut of Visual Computing
Sankt Augustin
Grantham-Allee 20
53757 Sankt Augustin, Germany
Andre.Hinkenjann@h-brs.de

Philipp Slusallek
Saarland University
Computer Graphics Lab &
Intel Visual Computing
Institute Campus E 1 1
66123 Saarbrücken, Germany
slusallek@cs.uni-saarland.de



ABSTRACT

We present a system that combines voxel and polygonal representations into a single octree acceleration structure that can be used for ray tracing. Voxels are well-suited to create good level-of-detail for high-frequency models where polygonal simplifications usually fail due to the complex structure of the model. However, polygonal descriptions provide the higher visual fidelity. In addition, voxel representations often oversample the geometric domain especially for large triangles, whereas a few polygons can be tested for intersection more quickly.
We show how to combine the advantages of both into a unified acceleration structure allowing for blending between the different representations. A combination of both representations results in an acceleration structure that compares well in performance in construction and traversal to current state-of-the art acceleration structures. The voxelization and octree construction are performed entirely on the GPU. Since a single or two non-isolated triangles do not generate severe aliasing in the geometric domain when they are projected to a single pixel, we can stop constructing the octree early for nodes that contain a maximum of two triangles, further saving construction time and storage. In addition, intersecting two triangles is cheaper than traversing the octree deeper. We present three different use-cases for our acceleration structure, from LoD for complex models to a view-direction based approach in front of a large display wall.

Keywords

Visualization, Computer Graphics, Ray Tracing, Level-of-Detail, Voxelization, Octree, SVO

1 INTRODUCTION

In contrast to polygonal model descriptions, volumetric descriptions are less sensitive to the scene's complexity and enable a progressive refinement – using e.g. octrees, necessary for out-of-core rendering and Level-of-Detail (LoD). However, if these Sparse Voxel Octrees (SVOs) [LK11] are to have a visual quality that compares to a polygonal description, they need a high resolution and require much memory space. When arbitrary scenes are voxelized, many voxels need to be created for single triangles, possibly oversampling the geometric domain even though the polygonal representation is more compact and provides the higher visual fidelity. In addition, it is often cheaper to intersect a couple of triangles compared to traversing an octree deeper. In this paper a hybrid approach is introduced where a SVO is extended with triangle references in the leaf nodes. The voxelization and construction of the structure is entirely performed on the GPU.

Having voxel and polygonal data in one acceleration structure is beneficial because it minimizes manage-

ment and storage cost compared to having two separate structures. In addition, having triangle information in the leaf nodes can reduce the size of the octree. The construction is stopped for those nodes that contain a maximum of two triangles. Two triangles building up a leaf node are often cheaper to intersect than traversing the structure deeper. In addition, they are common for non-isolated triangles, i.e. the ones sharing an edge. Non-isolated triangles form a solid surface and are not crucial to direct geometric aliasing problems. However, the polygonal information provides the higher visual fidelity.

Another benefit of the unified octree structure is that it allows for a convenient smooth intra-level interpolation and color blending between layers in the hierarchy and faster image generation for parts of the scene for which a coarse representation is sufficient.

We contribute by presenting a system to construct and render triangles and voxels in a hybrid acceleration structure. We show how to extend the voxelization method proposed [CG12] and how to perform an interactive construction of unified SVOs on the GPU. In addition, we present a compact data layout allowing for a fast traversal. Finally, we present three applications, where having pre-filtered voxels along with the polygonal information is beneficial and give benchmarks on the construction and traversal times and memory savings by embedding triangle data.

2 RELATED WORK

Several methods have been introduced to create a volumetric description out of a polygonal model, how to construct octrees or multi-level grids and how to traverse these structures.

One important step to generating unified triangle-voxel data is the transformation of the parametric or polygonal description of a model into a volumetric description (voxelization). Early systems such as the Cube system [KS87] try to rebuild a classical hardware-supported rasterization pipeline in software. They use a 3D Bresenham line drawing algorithm to draw the polygonal outline and perform a 3D polygonal filling step. These systems are slow and difficult to implement, as rebuilding an efficient hardware pipeline in software can be challenging.

As dedicated graphics hardware became available to the masses, systems for 3D rasterization using the GPU hardware were proposed. Systems like Voxelpipe [Pan11] and the one proposed by Schwarz and Seidel [SS10] perform voxelization using an optimized triangle/box overlap test on the GPU. The Voxelpipe system allows an A-buffer voxelization where each voxel stores a list of triangles intersecting it. However, using only a triangle/box overlap test creates a binary voxelization of the data, only specifying whether a voxel is on or off. This representation is not sufficient for a LoD representation of textured models. Another example is the system proposed by [ED06] that generates a binary voxelization. However, to use a voxel as a general rendering primitive, more information such as colors and normals are necessary.

Other approaches for performing a surface voxelization on the GPU using a GPU accelerated render pipeline are [DCB+04] and [ZCEP07]. Both approaches render the scene from three sides, combining multiple slices through the model into a final voxel representation. However, rendering a scene multiple times has a negative impact on performance. OpenGL allows to write to a 3D texture or linear video memory directly from the fragment shader. In [CG12], this feature is used to create a boundary voxelization of the model. In this approach, the model has to be rendered only once. Moreover, using the fragment shader means that colors and normals for each voxel are instantly available.

Several methods have been introduced for fast octree and multi-level grid construction. We focus on GPU in-core methods. Each voxel's position in a grid can be represented by a Morton code, that can be used for a fast bottom-up construction of the tree, e.g. in [ZGHG11] [SS10]. A way to create a two level grid is presented in [KBS11]. The algorithm starts by computing pairs of triangle-cell overlaps, sorts these pairs and then fills in the pairs in the grid cells. However, this method must sort the input data first and must be extended to more then two levels.

Another approach is presented by [CG12]. By running multiple shader threads, each voxel is written unsorted top-down to a set of leaf nodes. If a leaf node is touched by a fragment generated in the fragment shader, the node is subdivided further level-by-level. We use a similar approach, and extend it to get an A-Buffer voxelization as well as to construct our hybrid acceleration structure out of it.

A few approaches combine voxel and point based models with polygonal data – one is FarVoxel [GM05]. There, a voxel-based approximation of the scene is generated using a visibility-aware ray-based sampling of the scene represented by a BSP tree. FarVoxels can be used for out-of-core rendering of very large but static models only – the construction of the tree is an offline process. Another approach that combines rasterization and sample-based ray casting is [RCBW12]. In this approach, all the polygonal data is subdivided into cubical bricks, essentially performing a voxelization. However, it is mainly used to speed up rasterization using ray casting methods and not as a general rendering structure.

Sparse Voxel DAGs [KSA13] are an effective way to compact voxel data since they encode identical structures of the SVO in a DAG. However, this method lacks

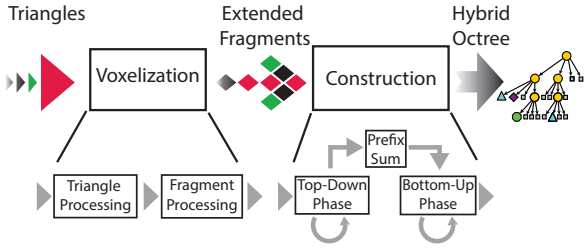


Figure 1: Overview of the GPU-based construction pipeline for the unified structure.

colors and normals for each voxel. If they were to be included, most of the compactness would be gone since colors and normals are unique for most parts of the scene and cannot be easily compacted in a DAG.

3 TRIANGLE/VOXEL STRUCTURE CONSTRUCTION

Our voxelization and octree construction process uses an approach similar to [CG12] using programmable shaders with GLSL. This approach is extended to generate the information on which primitives are touching each non-empty voxel. We show how to use this information to construct the unified acceleration structure on the GPU. Fig. 1 shows the GPU construction pipeline.

MORTON CODE 8B
RGBA 4B
NORMAL 12B
PRIMITIVE ID 4B

Table 1: Structure of an extended fragment entry generated during voxelization, including each element's memory size in byte

Voxelization: The voxelization is performed using OpenGL. The view port's resolution is set to match the voxelized model's target voxel resolution. The view frustum is set up to match the greatest extent of the scene's bounding box. After disabling depth writes and backface culling, each triangle within the view frustum creates a set of fragments accessible in the fragment shader. To extend the projected area of the triangle with respect to the view plane, the triangle is projected to the view plane as if it had been rendered from another side of the bounding box.

Since OpenGL samples each rectangular pixel during the rasterization within the pixel's center, the triangles need to be extended slightly in the geometry shader to ensure that each triangle intersecting a rectangular pixel area covers the pixel's center. This is performed by applying conservative rasterization [HAMO05]. Using the OpenGL Shading Language GLSL and atomic counters, each fragment is written from the fragment shader to a chunk of linear video memory.

Each of these extended fragments stores a position encoded in a Morton code. This enables us to perform a fast per-fragment traversal using bit shifts and a fast comparison of fragments generated at the same spatial position. In addition, the extended fragments store a color, a normal and a triangle index, i.e. the fragment index it originates from. To determine this index, we use the built-in variable `gl_PrimitiveID`. Tab. 1 gives an overview on the memory layout of each extended fragments.

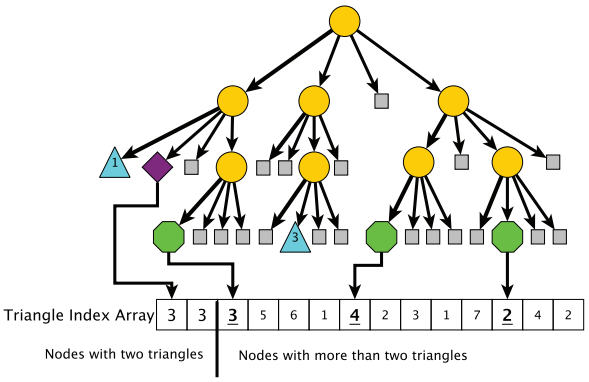


Figure 2: Overview of the unified data structure storing triangles and voxels. Inner nodes (orange), empty nodes (grey), leaf nodes containing a single triangle (light blue), leaf nodes containing two triangles (purple), and leaf nodes containing more than two triangles (green).

Data Structure: Fig. 2 shows the data structure. If a leaf node contains only a single triangle or two triangles, the tree does not need to be constructed for deeper levels for these nodes. If it contains more than two triangles, the node needs to be split. A single node can store the reference to a single triangle alongside with the voxel information. However, if it needs to encode two or more triangles, they are stored in a triangle index array.

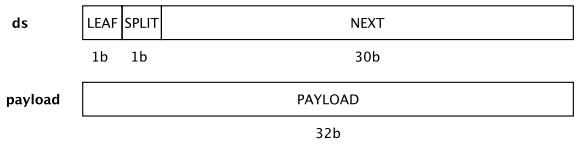


Figure 3: Structure of a single node in the octree.

Each node of the data structure is encoded in two 32 bit fields (see fig. 3). A single bit is used to encode whether the node is a leaf or not, another bit is used to mark a node during construction if it needs to be split further. The next 30 bits either encode the index of the first child node, the id of the triangle if it is the only one represented in the voxel or the index into the triangle index array. The other 32 bits `payload` hold a reference to a voxel array storing the voxel's color, its normal and possibly user-defined fields e.g. material parameters.

Construction: The main idea during construction to decide whether a node contains a single, two or more than two triangles is to cache and compare triangle indices in the 64 bit nodes.

The construction is a splatting process in which several vertex shaders are executed repetitively spanning an arbitrary number of threads using indirect draw calls. First the tree is traversed per-fragment in parallel and construction is done level-by-level. Afterwards the values from the inner nodes of the voxel structure and the color information are written back to the tree nodes bottom-up.

In the first top-down construction phase of the structure, we store the individual triangle IDs from each fragment in the node's two 32 bit fields using atomic comp-and-swap operations. If more than two triangles have to be stored in a node, this node needs to be marked for further splitting. In the next shader step new nodes and voxel payloads for deeper levels are created and the triangle IDs of those nodes that contain only two triangles are written to the triangle index array. Now the first stage is executed again.

Eventually, when the tree is created for the highest resolution, the number of triangles that fell into the leaf nodes are counted using an atomic add operation in the `payload` field. In this stage, each leaf node that has not been already finalized in a earlier shader stage, since it contained only up to two triangles, contains more than two. Afterwards, the triangle counts stored in each leaf node are written to a temporary triangle index count array.

In the next step the prefix sum of the triangle index count array is computed. Finally, the tree is traversed once more and the primitive IDs in the fragment are written to the final array locations in the triangle index array using the triangle index count array and the nodes are relinked accordingly. In this phase we can keep track of the individual primitive id locations in the triangle index array by decrementing the values in the triangle index count array using atomic add operations.

To decide whether a leaf node contains a single, two or more triangles offsets are added to the indices, we store in each leaf node's `next` field. If a node stores an index to a single triangle it encodes the triangle id directly. If it holds an index to a node containing more than two triangles it stores the maximal triangle id plus the index in the triangle index array storing two triangle indices consecutively. If it contains more than two triangles we add the maximal triangle id, the length of the triangle index array storing two triangles and the index. (See fig. 2)

The bottom-up phase continues by filling in the voxel colors, normals and primitive IDs for each node of the tree. Therefore, the tree is traversed per fragment in parallel. Once a shader thread reaches leaf node, the fragment's color and normal must be averaged. This is performed in a similar fashion as in [CG12]. Using an atomic compare-and-swap operation in a loop, each thread checks whether it can write its new summed and averaged value into the voxel's color field. For the normals a simple atomic add on the float components is used. If normals sum up to a zero length normal, e.g. for two opposing faces, the last valid normal is stored.

Finally the tree is processed bottom-up and level by level. Inner nodes are filled by averaging colors and normals and by normalizing the normals of all the child nodes, since the latter resulted only in adding up the normals in the step before.

4 TRIANGLE/VOXEL STRUCTURE TRAVERSAL & INTERSECTION

Rendering of the data structure is performed using a prototypical ray tracer using OpenCL. After the construction, each OpenGL buffer is mapped to OpenCL. These are the buffers containing the nodes, the voxels and the triangle index array and all triangle data, as well as the material information of the model.

Traversal: We decided to implement a traversal using a small stack on the GPU. We set the active parametric t-span of each ray that hits the scene's bounding box to the extent of this bounding box. The algorithm has three phases:

1. If the current first hit voxel within the active t-span is not empty, we traverse the tree deeper and push the parent node with the current t_{max} onto a stack. We set t_{max} to point to the end of the active voxel.
2. If the voxel is empty, we either need to process the next sibling node of the active parent by setting t_{min} to the beginning of the next node within the t-span or,
3. if the node is not a sibling node of the active parent, we need to pop nodes from our stack, reset t_{max} to the position stored on the stack until we can hit the first possible neighboring voxel, and traverse the tree deeper again.

If the traversal reaches a leaf, its triangles can be intersected - either one, two or more. Therefore, the algorithm looks at the index stored in the leaf's `next` field. Since the index is encoded using offsets, it can be decided directly if the node references a single, two or more triangles. The traversal code now determines the closest hit point of the ray and all triangles lying within that leaf node. If the closest triangle is hit and the intersection is within the boundaries described by the leaf node, the traversal returns a structure representing the hit point. Otherwise the traversal is continued with the next sibling node.

Full Octree Resolution						
Scene	Nodes	Triangles	Triangle Index Array	Voxel	**Overall**	
Sponza	42.29	27.66	14.14	46.06	130.15	
Urban Sprawl	18.32	75.19	19.31	20.38	133.21	
Happy Buddha	11.42	103.07	21.94	11.95	148.38	
Forest Scene	30.41	156.25	34.58	33.29	254.53	
Our Method						
Scene	Nodes	Triangles	Triangle Index Array	Voxel	**Overall**	**Saved**
Sponza	10.89	27.66	12.51	13.97	65.03	**50.03%**
Urban Sprawl	12.37	75.19	18.47	14.77	120.81	**9.31%**
Happy Buddha	10.97	103.07	21.92	11.81	147.77	**0.41%**
Forest Scene	21.27	156.25	34.00	27.2	238.72	**6.21%**

Table 2: Size of the acceleration structure (MB). The upper part of the table shows the acceleration structure size of the test scenes for a tree build for all octree levels. The lower part of the table shows our method, where the tree is built only for nodes containing more than two triangles.

Inter-level blending: For the LoD selection and to enable a smoother blending between different levels of the hierarchy we use Ray Differentials [Ige99]. Each ray is represented by its origin and a unit vector describing its direction. In addition, we store its' differentials describing the pixel offset on the image plane in x and y direction.

By using ray differentials, we can compute an estimated pixel's footprint in world space on the voxels. This footprint can be compared with the size of an individual voxel at level l. If the pixel's footprint is roughly equal or smaller than the voxel, we can stop traversing deeper.

In addition, we compute a value describing the underestimation $i(l, f)$ of the size of the pixel's footprint and the actual size of nodes at level l and $l-1$ by computing

$$i(l, f) = \frac{2 \cdot v_w(l) - f}{v_w(l)}$$

with $v_w(l)$ being the length of a side of a voxel in world space and f being the estimated length of the pixel's footprint at the ray's hit point. This value can be used as interpolation factor between the two subsequent levels in the SVO. Since we traverse the tree using a small stack, we can keep track of the voxel at level $l-1$ directly and use the interpolation factor during shading and lighting computations.

5 BENCHMARKS

The benchmarks of our system were performed using a Nvidia GeForce GTX Titan with 6GB video memory on an Intel Core i7 system with 16GB RAM. Fig. 4 shows the construction times of four different test scenes. The forest test scene shows 13 highly detailed plant models on a small plane. As expected, increasing triangle counts increase the run time of the construction. However, the pure triangle count is not the only parameter when it comes to measuring construction times as highly detailed textures and shaders extend the time it takes to voxelize the model.

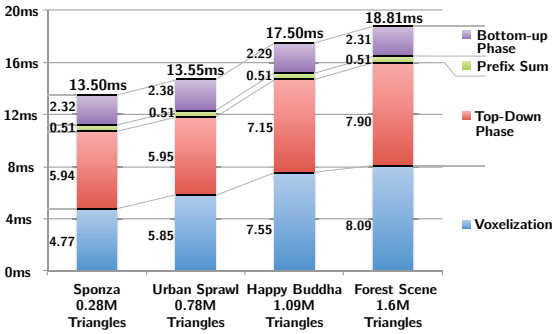


Figure 4: Run times for each phase of the construction as well as the overall construction time. Each scene was voxelized with a resolution of 512^3.

Scene	Voxel only	Triangle only	**Hybrid Structure**
Sponza	57.3 fps	18.2 fps	20.6 fps
Urban Sprawl	40.3 fps	13.3 fps	23.7 fps
Happy Buddha	63.1 fps	10.1 fps	16.7 fps
Forest Scene	64.2 fps	2.4 fps	12.9 fps

Table 3: Avg. fps of four different scenes rendered with a resolution of 1024×1024 using only primary rays and phong lighting with simple shadows and a single point light source. Each scene was voxelized with a resolution of 512^3.

Table 2 shows the advantage of our method in comparison to a full build of the octree without stopping the construction early in terms of size of the acceleration structure. Both versions store the triangles in their leaf nodes as a reference to the `triangle index array`. We have included the size needed to store the triangles themselves, which largely depends on the scene. The triangle count in the Sponza scene is very low. If one only considers the size of the nodes and the voxel data, the overall saved space amounts to a larger percentage for most scenes. The Happy Buddha scene has many, but very small triangles. For this scene construction can't be stopped for most inner nodes resulting in only a small memory saving.

We have rendered all scenes with a resolution of 1024×1024 using a typical fly through for about 700 frames and averaged the run times. The results in tab. 3 show the rendering times from the OpenCL renderer shooting primary rays with phong lighting, a single point light source and no texture filtering. Rendering only voxels is fast but lacking visual quality. Traversing our structure displaying triangles only provides the highest visual quality but is slow an offers no LoD - aliasing can occur. he hybrid structure provides a good trade-off in speed and offers LoD.

However, measuring the frame rates for the hybrid approach is non trivial since they increase drastically if parts of the scene show the voxel data only. For scenes like Sponza showing an atrium where a camera is mostly "inside" the model, only a few camera positions can make use of the voxel data, resulting in only a small speed up. In the Forest- or the Urban Sprawl scene parts of the model are in the distance more often. Thus the voxel data is used more frequently resulting in larger speed ups.

6 APPLICATIONS

Our hybrid structure is well-suited for applications that need a general LoD scheme, since the regular voxel description allows to create a representation for arbitrary input meshes. In principle the hybrid structure can be seen as a multi level grid, omitting the fact that this structure contains a color and a normal for each grid cell. However, this additional information, is well suited to some scenarios to reduce aliasing and speed up rendering. We present three different applications: a visualization of large outdoor scenes, urban environments and a view-direction based rendering approach in front of a large tiled display wall.

The first application (cf. fig. 6a) uses the hybrid acceleration structure to render highly complex vegetated areas with LoD. Here far distant models project to only a few pixels on screen creating aliasing artifacts. We use an approach similar to [DMS06] [WHDS13]. On the highest level a nested hierarchy of kd-trees over wang tiles with Poisson Disc Distributions is used to represent plant locations resulting in instanced, but aperiodic repetitions. Each scene contains millions of highly complex plant models reused throughout the scene.

The advantage of our hybrid representation over a polygonal simplification is that, within a regular octree structure, an approximation of high-frequency input models such as trees with different LoDs can be generated. Polygonal simplification of such models usually fails due to the complex foliage and branching structure of the trees. Sample caching strategies in object space that provide LoD are limited to single instances, e.g. samples can't be cached in the accelerations structure of a single tree since it is reused. Therefore, it is beneficial to have pre-filtered voxel data at hand to limit aliasing artifacts or to reduce the oversampling needed to create smooth animations and crisp images. In addition, this speeds up rendering. We can render a scene with trillions instantiated triangles consisting of 40Mio. trees at a resolution of 720p with about 5-7fps including direct shadows using our prototypical OpenCL ray caster.

Another example where this LoD structure is beneficial are urban scenes as shown in fig. 5a and fig. 5b. Even though a polygonal simplification of such structures is not as challenging as for tree models, renderings of such scenes from far away have to cope with high-frequency aliasing. If this urban scene is viewed from a distance, the highly varying z-depth of the scene generate geometric aliasing which can be reduced by having a pre-filtered voxel structure. Moreover, voxel are independent from the scenes local complexity. In addition, possibly large triangles in such a scene further reduce the size of the octree. Furthermore, the hybrid structure allows for smoother transitions and color blending between different layers of the hierarchy (cf. fig. 5b) and faster render times for highly detailed parts in the scene that are viewed from the distance.

A further application is shown in fig. 6b. There the structure is used for a view dependent rendering on a large tiled display wall. Since coarse voxel representations can be renderer faster than highly complex polygonal models, the voxel representation is mainly used to speed up rendering.

The user's central field of view is tracked and rendered in high quality using the polygonal representation, whereas the surrounding is rendered using our LoD approach. Therefore, we compute an intersection of the tracked user's view frustum with a virtual display wall. Using the intersections an ellipsoid is generated. Points within this ellipsoid are rendered with maximal resolution using polygonal data. For points outside of the ellipsoid the distance from the ellipsoid to the current pixel is computed. This distance is use to decide whether a deeper traversal of the hierarchy is necessary or if traversal can be stopped early. The transitions between the layers of the hierarchy are blurred using a post-processing step in image space.

7 DISCUSSION

We presented an approach to building a hybrid acceleration structure storing voxels for inner nodes, stopping construction of deeper levels if the number of primitives within that node are not larger than two and storing the full triangle list for each leaf node that represents the finest voxelized level. This way, we generate a LoD description of the input geometry. The advantage of this representation over a polygonal simplification is that, within a regular octree structure, we generate a good

(a) (b)

Figure 5: Rendering of an urban environment (5a) using our unified octree structure with voxel data in the background. Fig. (5b) shows a color coding. The red areas were rendered using polygonal data and the green regions were rendered using voxels.

(a) (b)

Figure 6: Rendering of instantiated tree models (6a) and a focus and context based rendering in front of a large display wall (6b) using our unified acceleration structure.

approximation of high-frequency input models such as trees. In addition, this speeds up rendering by providing a coarse representation for areas that are of minor interest in a visualization or are not visible/noticeable to the user. Since the construction on the GPU is performed in-core, the resolution of the voxelization is limited. However, the system is fast enough to construct an octree of a scene in real time doing a complete rebuild.

One problem targeted by further research is that an octree is not truly adaptive with respect to the scene's input geometry. If one has highly complex geometry inside a single leaf voxel, traversing these parts of the scene can have a huge impact on performance. Simply building a tree deeper by a regular subdivision of these parts, is often not sufficient to divide the model's input geometry. It would be better to either identify these high resolution parts beforehand and voxelize them separately or automatically use truly adaptive acceleration structures such as BVHs or kD-Trees for these parts of the scene. However, since a coarser voxel representation is available, the renderer can decide to stop traversing these parts and display the coarse voxel representation to stay within a constant frame rate. In addition, due to the regularity of the octree's structure, more advanced optimizations such as e.g., a beam optimization [LK11] could be applied. Moreover, for improved GPU utilization, it might be beneficial to postpone the triangle intersection from inside the octree traversal to subsequent rendering passes.

Another aspect crucial to performance is memory management. Since the number of fragments generated by the voxelizer, the size of the octree and the triangle index list are not known in advance, buffers must either be preallocated with a maximal size, be used in a caching scheme (e.g. [CNLE09]), or more advanced memory management must be applied – though determining the size needed for buffers, is a problem most grid construction algorithms have in common. However, once we have generated the voxel's extended fragment list, our approach can stop the octree construction early when too much memory is needed to construct deeper levels. The system has been extended to perform an out-of-

core voxelization and construction for parts of the scene that have to be voxelized with a higher resolution.

Voxel structures have disadvantages which should be targeted by further research. It is merely possible to average different material informations inside a singe voxel cell. Furthermore, due to their grid like structure, shadows are hard to implement because neighboring voxels tend to cast shadows on themselves. These shadows create a high-frequency noise in the image which is disadvantageous if one wants to use voxels to reduce aliasing. Another issue is the size of the structure. However, we have shown that our structure is compact enough to represent several dozens of models, voxelized with a high-resolution, in GPU memory at once.

8 REFERENCES

[CG12] Cyril Crassin and Simon Green. *Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer*. OpenGL Insights. NVIDIA Research, July 2012.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. to appear.

[DCB+04] Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng. Real-time voxelization for complex polygonal models. In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, PG '04, pages 43–50, Washington, DC, USA, 2004. IEEE Computer Society.

[DMS06] Andreas Dietrich, Gerd Marmitt, and Philipp Slusallek. Terrain guided multilevel instancing of highly complex plant populations. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 169–176, September 2006.

[ED06] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78. ACM SIGGRAPH, 2006.

[GM05] Enrico Gobbetti and Fabio Marton. Far voxels: A multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 878–885, New York, NY, USA, 2005. ACM.

[HAMO05] Jon Hasselgreen, Tomas Akenine-Möller, and Lennart Ohlsson. *GPU Gems 2: Conservative Rasterization*, volume 2, chapter 42, pages 677–690. NVIDIA, Addison-Wesley, 2005.

[Ige99] Homan Igehy. Tracing ray differentials. pages 179–186, 1999.

[KBS11] Javor Kalojanov, Markus Billeter, and Philipp Slusallek. Two-level grids for ray tracing on gpus. *Comput. Graph. Forum*, 30(2):307–314, 2011.

[KS87] Arie Kaufman and Eyal Shimony. 3d scan-conversion algorithms for voxel-based graphics. In *I3D '86 Proceedings of the 1986 workshop on Interactive 3D graphics*, pages Pages 45 – 75. ACM New York, NY, US, 1987.

[KSA13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel dags. *ACM Trans. Graph.*, 32(4):101:1–101:13, July 2013.

[LK11] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17:1048–1059, 2011.

[Pan11] Jacopo Pantaleoni. Voxelpipe: A programmable pipeline for 3d voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 99–106, New York, NY, USA, 2011. ACM.

[RCBW12] Florian Reichl, Matthäus G. Chajdas, Kai Bürger, and Rüdiger Westermann. Hybrid sample-based surface rendering. In *Proceedings of VMV 2012*, pages 47–54, 2012.

[SS10] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph.*, 29(6):179:1–179:10, December 2010.

[WHDS13] Martin Weier, André Hinkenjann, Georg Demme, and Philipp Slusallek. Generating and rendering large scale tiled plant populations. *JVRB - Journal of Virtual Reality and Broadcasting*, 10(1), 2013.

[ZCEP07] Long Zhang, Wei Chen, David S. Ebert, and Qunsheng Peng. Conservative voxelization. *Vis. Comput.*, 23(9):783–792, August 2007.

[ZGHG11] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669–681, May 2011.