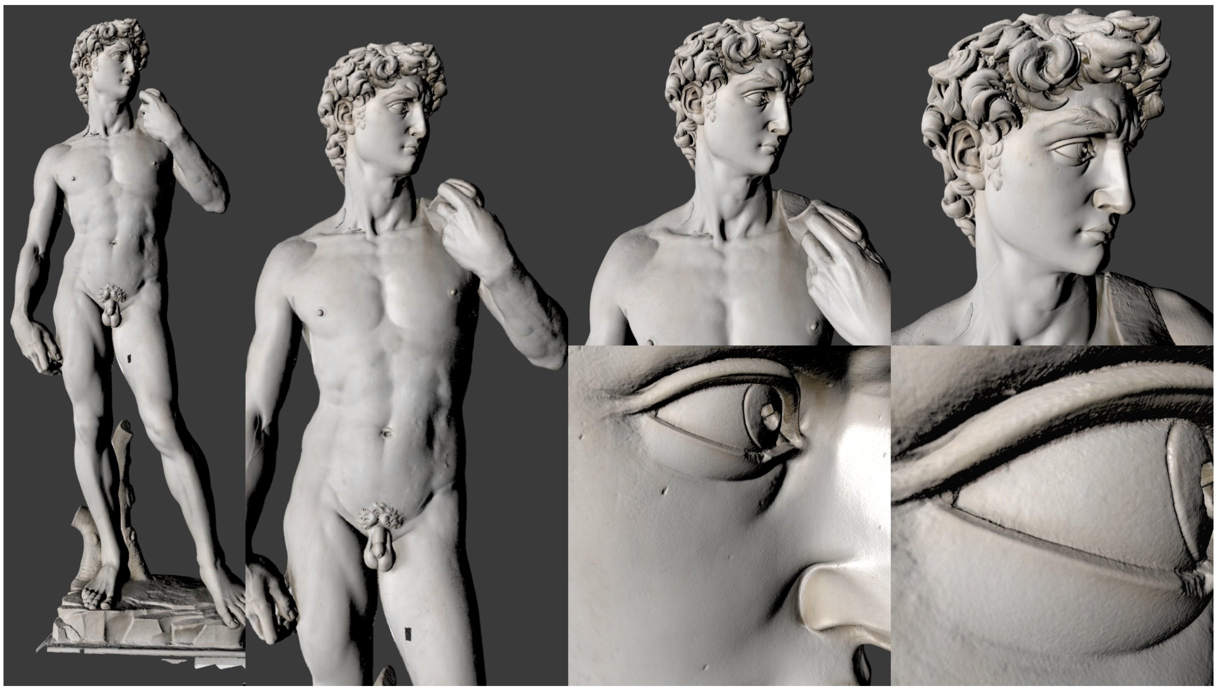
Scalable rendering for very large meshes

Matthäus G. Chajdas Matthias Reitinger Rüdiger Westermann
Technische Universität München
Informatik 15 (Computer Graphik & Visualisierung)
Boltzmannstrasse 3
85748 Garching bei München
chajdas@tum.de, reitinge@in.tum.de, westermann@tum.de



Abstract

In this paper, we present a novel approach for rendering of very large polygonal meshes consisting of several hundred million triangles. Our technique uses the rasterizer exclusively to allow for high-quality, anti-aliased rendering and takes advantage of a compact, voxel-based level-of-detail simplification. We show how our approach unifies streaming, occlusion culling, and level-of-detail into a single rasterization based pipeline. We also demonstrate how our level-of-detail simplification can be quickly computed, even for the most complex polygonal meshes.

Keywords
Computer Graphics, Large model rendering, Level-of-detail

1 INTRODUCTION

Rendering of very large meshes, consisting of several hundred millions of triangles, still represents a significant challenge. Recently, voxel based rendering techniques have been proposed to solve this problem [RCBW12, CNLE09, LK11]. Voxels are an interesting alternative to standard, polygon based level-of-detail approaches as they drastically simplify the simplification process. However, all of the current voxel-based techniques rely on the GPU's compute units to perform per-pixel ray-casting. This, however, comes with two important drawbacks: First, and most importantly, anti-aliasing in a ray-tracer is expensive. A ray-tracer scales linearly with the number of rays, and additional rays for anti-aliasing drastically increase the cost. Adaptive anti-aliasing can alleviate this to a degree, but on the very high resolution meshes we focus on in this work, geometric aliasing is present practically everywhere (see Figure 6.) Even with geometry pre-filtering through level-of-detail simplification, the surfaces are still prone to aliasing, which is amplified when moving around the object. Second, efficient ray-tracing on GPUs requires some kind of acceleration structure. These structures typically take a long time to build and require large amounts of memory.

The GPU rasterizer on the other hand does not scale well with increasing triangle counts, but allows for very efficient anti-aliasing. In this paper we demonstrate that pure GPU rasterization is a viable alternative to hybrid GPU raytracing/rasterization techniques for very large models, once it has been augmented by hierarchical streaming, level-of-detail and occlusion culling. We also exploit the characteristics of our alternative pipeline to create an extremely fast level-of-detail simplification. To achieve this, we make the following specific contributions:

- A memory-efficient, "linear" voxel model representation, which can be build from either a triangle mesh or a volume.
- A hybrid rendering approach, which combines level-of-detail, streaming, and occlusion-culling.
- A fast level-of-detail simplification working directly on the compact representation.

We start with a brief introduction to existing work in the field. In the algorithm part, we describe in detail how each part of our pipeline works. Finally, we present the performance results of various representative data sets on commodity hardware and provide details about quality and memory usage.

2 RELATED WORK

Voxel models have been first introduced 1993 by Kaufman in the seminal paper [KCY93]. Compared to polygonal representations, they provide an interesting set of advantages like easier level of detail computation and combined storage of surface and geometry information.

In the last years, there has been a lot of research into large octrees to render such voxel models [CNLE09, LK11]. [CNLE09] subdivides the model into a sparse volume, storing only small volume “bricks” along the surface. It uses a compute based octree traversal to render the contained surface, and also supports fully volumetric rendering as required for instance for clouds. However, it has a significant memory overhead for solid models as it stores parts of the volume around the object surface. It also requires additional memory for the octree data structure on the GPU. [LK11] provides an interesting optimization by focusing on octrees for surfaces: Along with the surface data like color, they also store contours which both improve the quality of the reconstructed surface as well as the performance of the rendering. In this case, the octree must be built top-down as successive levels combine the contours. Similar to GigaVoxels, the sparse voxel octrees also use the GPU’s compute units for rendering.

Our representation builds upon the very first published work on iso-surface visualization: *Cuberilles* [HL79]. The Cuberille method—or opaque cubes—works by computing the set of grid cells that contain the iso-surface and rendering those as small cubes. The original method creates a single connected mesh during traversal to minimize the memory required by duplicated vertices. In order to improve the apparent surface quality, gouraud shading is used to interpolate the per-vertex normals along the surfaces.

Recently, [RCBW12] showed how voxel raycasting can be used for rendering very large models. Their approach uses a very compact surface representation and switches between voxels and triangle rendering for close-up views. They also show that voxels can be used to provide a high quality level-of-detail simplification. However, in their work, the level-of-detail computation is done in a pre-process and is not created by using their compact representation. This makes it very time-consuming, as it has to process the complete model for every simplification step. Finally, they rely on ray-tracing for rendering, making anti-aliasing very expensive.

Point-based rendering [BK03] is also related to our technique, but there are several significant differences. Our surface voxels always produce a watertight surface and require no blending between points. We can also integrate our technique easily with other algorithms like shadow mapping, as our representation is view-independent for a given level-of-detail configuration. This guarantees that the voxel geometry matches exactly for different views.

3 SURFACE VOXEL REPRESENTATION

Our algorithm takes advantage of a voxel representation for level-of-detail simplification. This requires us to generate a voxelization of the input. To this end, we use a standard 3D triangle voxelization algorithm. One difference to existing approaches which require the full volume [SS10] or out-of-core processing [BLD13] is that we use an in-core algorithm. It builds an adaptive tree representation during the voxelization and computes per-voxel coverage, normals and colors. Compared to [BLD13], it trades memory for processing speed, but due to its adaptive nature the memory re-

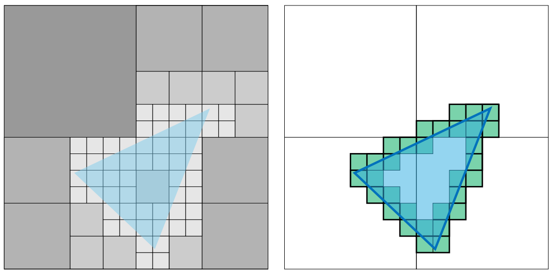


Figure 1: Adaptive tree building in 2D. The blue triangle is split into leaf nodes while the tree is being built. On the right hand side, chunk borders are visible, and the green voxels correspond to the surface voxels. The thick faces are the active surface faces.

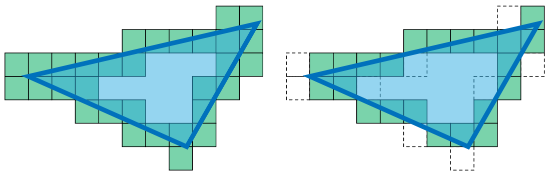


Figure 2: Comparison of conservative with 6-separating voxelization. On the left, conservative voxelization marks *any* voxel touched by the triangle. On the right, the 6-separating subset is shown.

quirements are still far below that of a full volume (see the results section for details).

Our voxelizer builds an octree step-by-step while consuming the input triangles. Starting at the root node, which represents a voxel enclosing the entire model, we build the tree top-down as follows: For each voxel, we split it into 2^3 sub-voxels and identify which of these sub-voxels intersect with the triangle. The triangle is then clipped against each sub-voxel and inserted into them (see Figure 1). At this level the triangles are then processed recursively until a leaf node is reached. The splitting process computes the exact intersection area of a voxel with a triangle, which we use at the finest level to compute an area weighted average of all triangle attributes. Once all triangles have been processed, we traverse the tree and write out the leaf voxels into a linear buffer.

As described so far, the algorithm will produce voxels which correspond to a conservative rasterization. In particular, along the surface of the object, on average twice as many voxels will be generated as necessary for rendering (see also Figure 2). To resolve this, we use an additional per-voxel test and only emit the six-separating subset of the original voxelization. This can be easily done by an additional triangle/voxel test [Lai13].

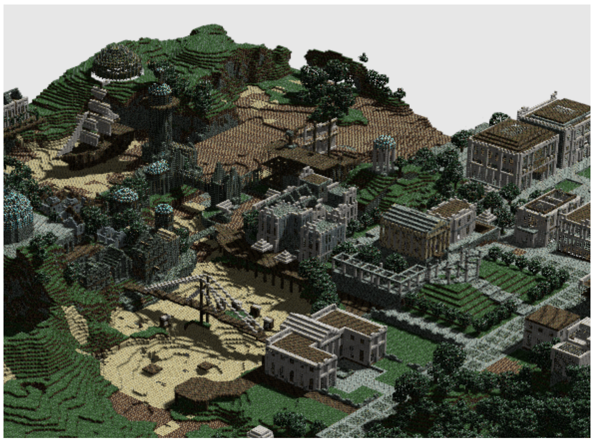


Figure 3: Our approach handles arbitrary per-face data such as material IDs. In this example, a Minecraft level is rendered with per-face textures.

The result is finally emitted as chunks of 256^3 voxels or less. This allows us to store the voxel-position using 8 bits per component. We use the normal at each voxel to determine the outward facing voxel faces (see Figure 1) and store them using a 6 bit mask. Together with the active face mask and padding, this results in a total of 4 bytes per voxel. We also store a quantized normal for each voxel with 10 bits per channel in the voxel buffer, resulting in 4 additional bytes of data. If color data is available, it is also stored using 4 bytes due to alignment.

Optionally, we can also store per-face data. For example, different materials can be assigned to each face (see Figure 3). In this case, we create an additional buffer with per-face attributes and store an index into this buffer along with each voxel. The indirection is necessary, as the amount of per-face data can vary between voxels.

Level of detail

Level-of-detail is an integral part of voxel rendering and is also a core component of our approach. As discussed, every voxel will be eventually rendered as a small cube. This has three important implications for level-of-detail: Firstly, we can use the voxels to solve problems with transitions between different levels. By ensuring that all border voxels of a chunk have "caps" at the exterior, we can guarantee that no holes will ever be present at level-of-detail transitions.

The second implication is that we can run our simplification directly on the voxels. This might sound trivial at first, but it has an extremely high impact on performance and memory usage, as our voxel representation is very compact. Instead of having to process the complete polygonal input model, we can work with the much smaller voxelization.

Finally, for the rendering, we will also require conservative bounds for any given chunk. We can achieve this if we can guarantee that every single voxel at a higher level-of-detail encompasses all child voxel. This can be incorporated into the simplification by computing which faces are present for any child and `or`-ing them together.

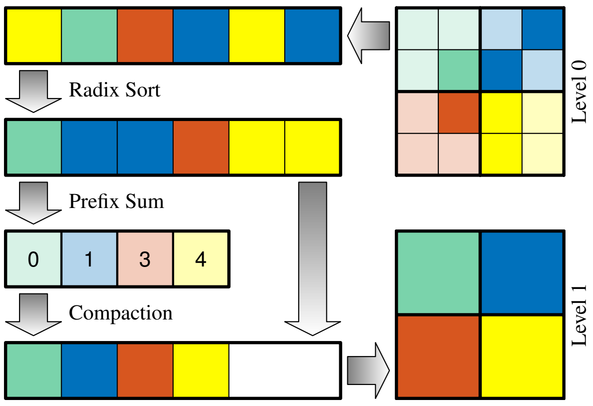


Figure 4: The simplification process: Starting from a high-resolution block at the top right, the surface voxels are extracted, sorted, and compacted to create the level-of-detail simplification at the bottom right.

Our simplification is designed to be fully parallel, suitable for efficient execution on both CPUs and GPUs. The main problem when trying to run the simplification in parallel is that we cannot easily identify all neighboring voxels directly in the voxel buffer. To solve this, we use a multi-pass algorithm which first sorts the voxels and then compacts them. In the first step, we produce "runs" of voxels which all correspond to a single voxel at the next lower level. This can be done by performing a radix sort [Mer12] which ignores the last bit of each component of the position. Having the runs generated, we now have to find out where each run starts. This can be accomplished with a parallel prefix-scan [Mer12], which computes the start offset of each voxel run (see Figure 4).

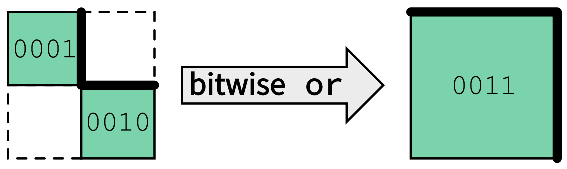


Figure 5: Each voxel contains a bit-mask which specifies the visible faces. During simplification, the masks are merged by using a bitwise `or`.

Finally, we compact each run into a single output voxel. Using GPU terminology, we start one thread per run, which first computes the new boundary face using an `or` operation of all children (see also Figure 5.) Attributes are combined using averaging: Either per-voxel or per corresponding face. Combining the face bits using `or` creates a convex hull, which also implies that small features increase in size instead of disappearing. As mentioned above, the choice of the combination algorithm is not arbitrary, but necessary for efficient culling as described in the rendering section.

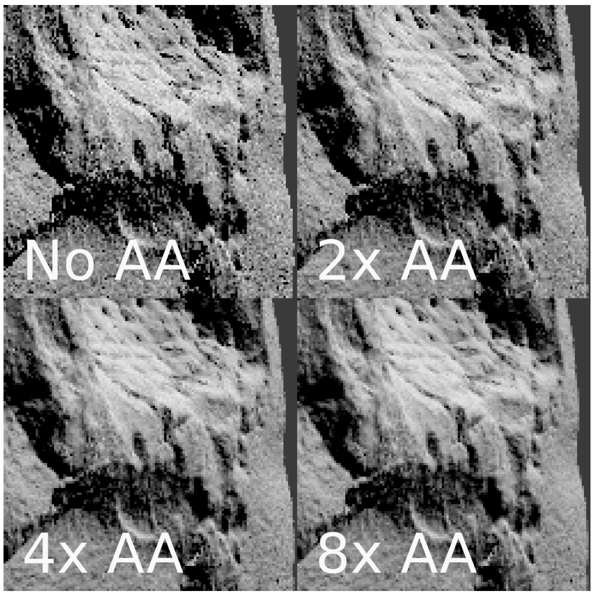


Figure 6: Magnified view of a part of the Atlas mesh. Even for pre-filtered geometry, anti-aliasing is still crucial to obtain a high-quality image.

4 RENDERING

The final stage of our algorithm consists of a fast voxel renderer. We use the hardware rasterizer to support efficient rendering at high resolutions as well as multi-sample anti-aliasing, as this is still necessary with voxel models for high quality rendering (see Figure 6). This requires us to transform the input voxels into triangle based geometry. To keep the memory usage low, we never store the geometric representation but rather generate it on-the-fly from the voxel buffers using a geometry shader.

For each voxel, we generate a partial cube with all boundary faces. In order to be fast, we must ensure that the geometry shader produces as few triangles as possible. Otherwise, the amount of transient on-chip memory reduces the possible parallelism and decreases performance. The key observation is that if the surface is seen from the "outside", at most three faces of each voxel can be visible. This optimization assumes that the viewer can never enter the object or see "into" it—very similar to the requirements for meshes that are rendered with back-face culling. If the viewer can enter the model, we have two possibilities: Interior faces can be marked as visible during the creation, resulting in a one voxel thick "shell" without reducing performance. Alternatively, the interior can be rendered by generating up to five faces per voxel at reduced perfor-

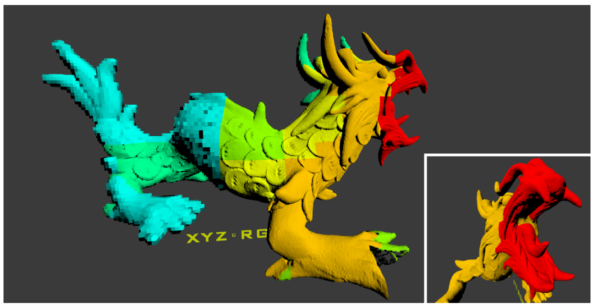


Figure 7: The octree traversal automatically reduces the level-of-detail for occluded areas. The inset shows the actual camera view, with colors indicating the level-of detail. In the side view, we can see that the occluded parts are rendered using drastically reduced level-of-detail.

mance. Throughout our testing, we always used the first approach.

Inside the vertex shader, we perform backface culling to determine which of the active boundary faces are actually visible. This information, together with the position of the voxel, is then forwarded to the geometry shader. In the geometry shader, we generate up to three faces consisting of two triangles each. As we could have per-face data, we cannot share vertices between faces; the maximum number of vertices generated by the geometry shader is thus 12. The geometry shader also extracts and decompresses the normals, and optionally generates additional per-face attributes like texture coordinates as can be seen in Figure 3.

Level-of-detail and occlusion culling

We have integrated the level-of-detail solution into our rendering with a combination of a CPU octree and GPU occlusion determination. On the CPU side, we compute a complete octree covering the whole scene and traverse it per frame to identify the potentially visible leaf nodes. The traversal is guided by the level-of-detail computation, that is, it stops as soon as a node has the correct resolution for the current view. While traversing, we also perform hierarchical frustum culling. During rendering, we force early depth/stencil testing and write to a buffer with one entry per brick for every pixel that passes the depth test. Using a single buffer instead of occlusion queries is important for performance, as we typically have to issue a few thousand draw calls per frame. It also reduces the CPU time for readback, as only a single API call is necessary to obtain the results. We improve the efficiency of the depth test and culling further by sorting the visible leafs by depth before issuing the draw calls.

Once the results are back, we update the octree and determine the visible nodes for the next frame. The key insight here is that any node can be used as a conservative

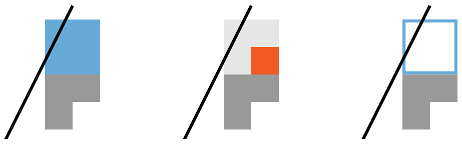


Figure 8: On the left, a view ray hits the blue octree node which should be refined. In the middle, the children of the blue node are highlighted in red. As all children are invisible, we mark the blue node for occlusion rendering only in the next frame.

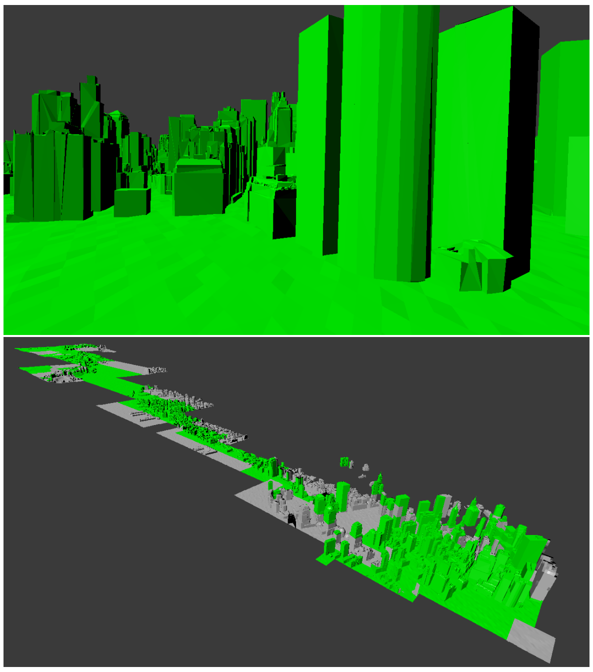


Figure 9: Top: Camera view into a city model. Bottom: Parts visible to the camera are highlighted in green, nodes rendered and determined invisible are gray. Notice that our occlusion culling algorithm pruned most of the scene and only few nodes adjacent to visible areas are rendered.

visibility bound of its children. This is a major difference to algorithms like [BWPP04], which render object bounding boxes. In our algorithm, we can take easily advantage of the level-of-detail simplification, which is also the best possible bound for the underlying geometry at any given level. As a result, we obtain a tight bound on the visible data set, as can be seen in Figure 9.

If we determine that all children of a node are invisible due to occlusion, we simply stop the traversal in the next frame at the node itself and render it. This scheme quickly propagates information about visibility up through the tree. As a result, occluded branches of the tree get rendered at very low resolution. This is a safe operation in our scheme, as our level-of-detail is conservative, which guarantees that we can never miss a visible block. The effect of this optimization can be seen in Figure 7.

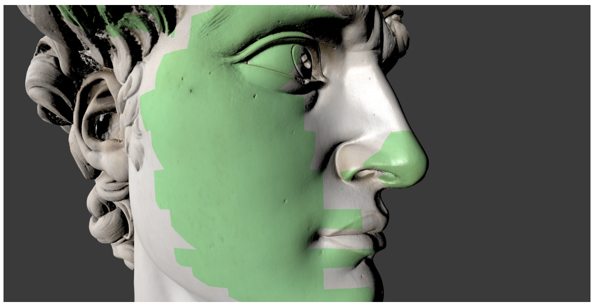


Figure 10: Once the camera is close enough to the model, we render the source geometry (tinted green) instead of voxel data.

One issue with this method is flickering if a node is visible at a level-of-detail which requires refinement, but its children are not. In this case, we would render the children in one frame, determine that they are invisible, render the parent node in the next frame, which would create a few visible pixels, and then render the children again. We avoid this by disabling writes to the color buffer if we render a node only to determine visibility (see Figure 8.)

Streaming

We take advantage of our octree representation to integrate streaming. Instead of loading all the geometry and voxels up-front, we fetch the data on demand. The streaming process always starts at the root of the octree. If a node hasn't been loaded yet but is part of the view frustum, a load request is issued and traversal stops. Otherwise, we determine if the node is visible at the correct level-of-detail. If not, traversal continues into the children.

At this point, we have two options how to handle a node which needs to be refined and contains children which haven't been loaded yet. The first option is to mark all children as invisible, issue load requests and render the parent node instead. This will guarantee that the model is always closed by reducing the level-of-detail temporarily until all visible children of a node have been loaded. The second option is to allow partial updates. In this case, load requests will be issued for all children and those already loaded will be rendered. This may lead to holes while child nodes are being streamed in, but avoids reducing the level-of-detail.

The first method is preferable when zooming in on the model, as it guarantees that visibility is never overestimated. The level-of-detail reduction is also almost never triggered when zooming in. On the other hand, while panning across a model or zooming out, new parts of the model come into the view frustum frequently, requiring the first method to reduce level-of-detail. The second approach copes much better with such situations, as the parts loaded at the correct level-of-detail remain fixed at that level.

In both cases, we have optimized the load order by sorting the requests such that we process one complete detail level before continuing, and by prioritizing parts closer to the camera. This ensures that the model refines quickly using the first method, and that only small parts become invisible at a time using the second method.

On the GPU, we try to cache as many chunks as possible to allow for quick movement in the scene without having to fetch data from the host every time. Once a cache becomes 95% full, all chunks which are currently not visible are evicted and the cache is defragmented.

We can also artificially limit the amount of data uploaded each frame to reduce frame-rate spikes, for instance, if large parts of the model become visible at the same time. This is only an issue if the data has been preloaded to the host memory already, as otherwise, disk I/O speed will be a far more limiting factor.

Due to our streaming system, we can render a model with very low memory requirements. As can be seen in Figure 2, the total amount of GPU memory for a single frame is typically around 1% of the total model size. Any additional memory is used to cache chunks on the GPU to avoid re-uploading them.

Geometry

For the high quality we aim to achieve, the voxels are not sufficient for close-ups. If the camera moves close enough to an octree leaf that the voxel resolution is no longer sufficient to maintain the pixel error guarantee, the leaf node is replaced by the actual input geometry (see also Figure 10.) This requires two additional caches for vertex and index data, which are handled using the same policy as the voxel data. At runtime, we render the source geometry first to prime the z-Buffer, as it is by definition closest to the camera. Due to our rasterization-centric approach, rendering of polygonal geometry automatically integrates with our occlusion-based visibility scheme and also benefits from anti-aliasing.

5 RESULTS

Before we dive into the results of the rendering, we have to actually create the surface voxel data first. We have measured the performance of the voxelization for three scenes: David [PGC11] (see Teaser), Atlas and St. Matthew. The David mesh consists of 940195349 (940 million) triangles with per-vertex normals and colors and requires 31 GiB of memory total. Atlas contains 507512682 (508 million) triangles without any additional per-vertex data, and St. Matthew contains 372767445 (373 million) triangles, again without additional per-vertex attributes.

On a dual Intel Xeon X5560 at 2.8 GHz ($2\times$ 4C/8T), using 16 threads, the voxelization of the David mesh at

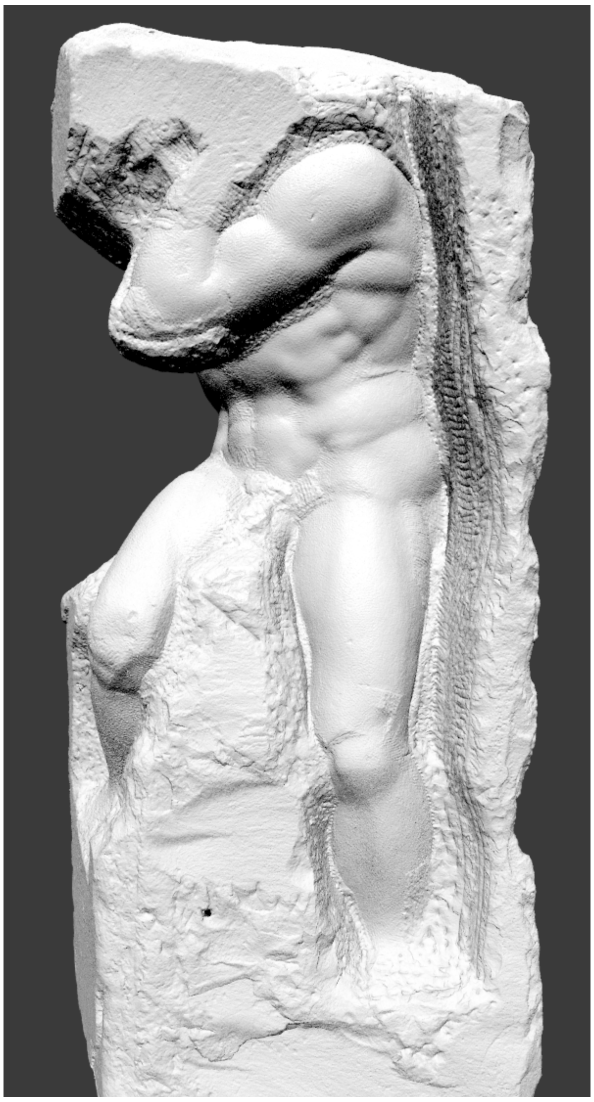


Figure 11: The Atlas dataset.

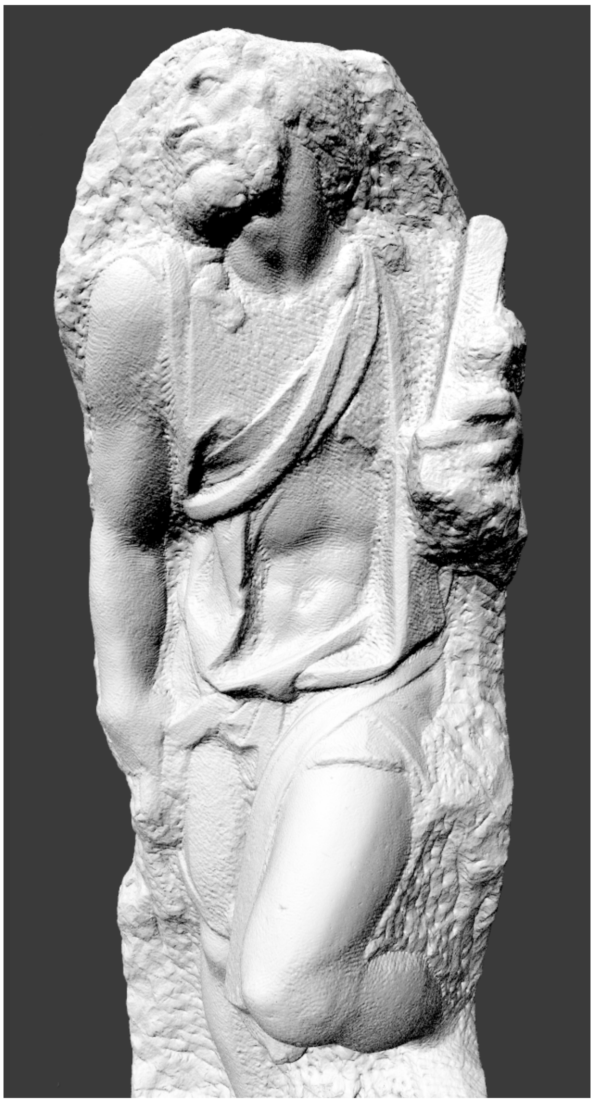
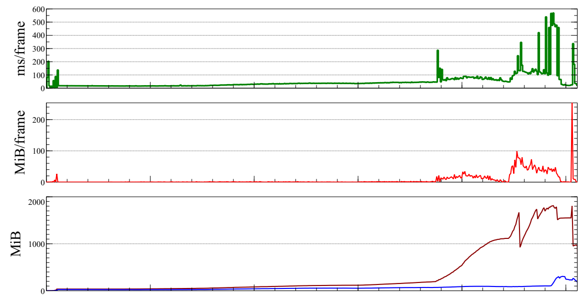


Figure 12: The St. Matthew dataset.

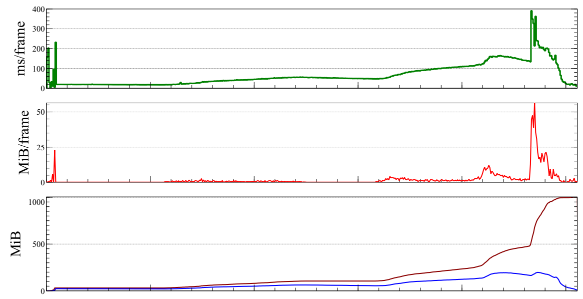
16384^3 takes 5 minutes, requires 9.27 GiB of memory and generates 166481909 (166 million) voxels. This includes 40 seconds which are spent on writing the output data to disk, which is running on a single core. The complete volume would require 32 TiB, if stored fully in memory. Atlas requires 6 minutes, 45 seconds (including 2 minutes, 45 seconds for writing data), 18.59 GiB of memory and generates 332965775 (333 million) voxels. St. Matthew requires 4 minutes, 46 seconds (including 1 minute, 43 seconds for writing data), 14.6 GiB of memory and generates 279042135 (279 million) voxels. The higher voxel counts for Atlas and St. Matthew are due to the much higher complexity of the surface. Unlike David, which is very smooth, both Atlas and St. Matthew have a very rough surface.

We used the CPU simplification algorithm for these meshes, using a single CPU core. For the David data set, the simplification requires 1 minute, 25 seconds. The corresponding times for Atlas and St. Matthew are 2 minutes, 54 seconds and 2 minutes, 9 seconds, respectively. In all cases, the CPU simplification was using less than 200 MiB of memory.

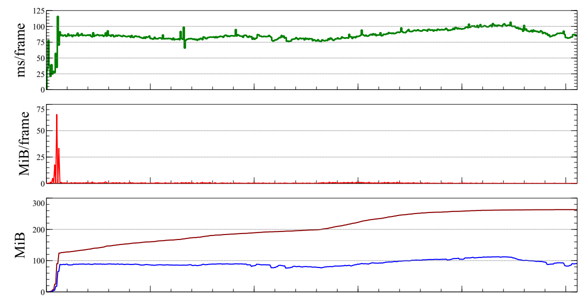
We have measured the per-frame memory usage, rendering time and upload rate for zoom-ins on the David and Atlas data sets, and a rotation of the Atlas data set (see Figure 13.) For this test, we have pre-cached the data on the CPU to limit the disk I/O impact. Still, without any restriction, GPU upload easily becomes the bottleneck as can be seen towards the end of the sequence, where large amounts of geometry have to be uploaded. In Figure 13, we can also see that for the David data set, the GPU cache had to be defragmented several times towards the end of the sequence. The cache size in this test was set to 1 GiB for voxels, 1 GiB for vertices and 512 MiB for indices. Notice that as we only remove invisible data from the cache, there is no upload spike after a defragmentation.



(a) David, zoom-in



(b) Atlas, zoom-in

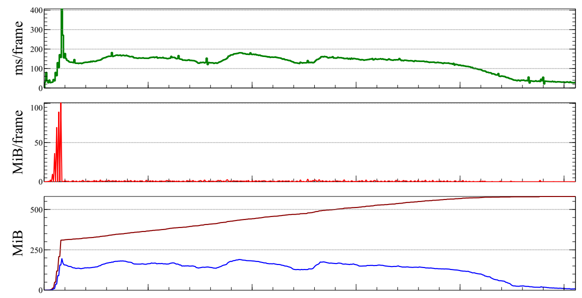


(c) Atlas, rotation

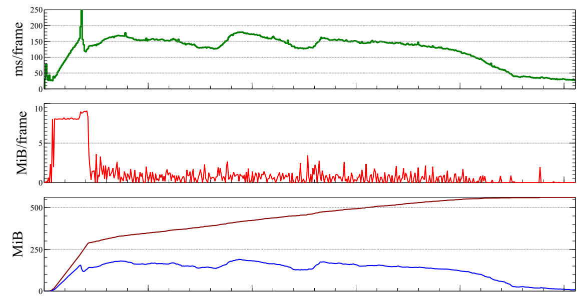
Figure 13: Timing and memory usage details at 1920 × 1080 on a AMD R9 290X. Top row: Frame time (in ms), middle row: data upload per frame (in MiB), bottom row: Required memory for the current frame (in MiB) in blue and the total cached memory in red. In a) and b), the camera zooms onto the objects, showing more and more detail. In c), the camera rotates around the Atlas data set.

The rotation around the Atlas data sets exhibit very high coherency, as the model remains at a fixed distance to the camera and hence at a fixed level-of-detail. After the initial loading, on average, 280 KiB of data is uploaded per frame.

To evaluate the impact of upload rate limiting, we have used the city data set (see Figure 14.) In this scene, we start at an elevated position above the city and descent down to street level. At the beginning, nearly the complete city is visible and large amounts of voxel data have to be loaded. Without rate limiting, this results in slow rendering and a high amount of disk I/O. With an upload limit, we can see how the initial loading requests get spread out over multiple frames. Once the initial data has been loaded, the upload is no longer the bottleneck, even though the occlusion is much more complicated in this scene than for the other data sets.



(a) City



(b) City, upload limited

Figure 14: Timing and memory usage for the City data set. The camera starts above the city, where nearly no occlusion is present and ends at street level. In a), all data required for each frame is immediately uploaded, resulting in a huge spike at the beginning. In b), the upload rate has been limited to 8 MiB/sec, smoothing out the initial loading.

Model	Geometry	Voxels	Total
Atlas	9.6 GiB	5.4 GiB	15 GiB
David	31 GiB	2.7 GiB	33.7 GiB
St. Matthew	7.1 GiB	4.5 GiB	11.6 GiB

Table 1: On-disk sizes for the various models. The David geometry contains 32-bit floating point positions, normals and colors, while Atlas and St. Matthew contain per-vertex position data only.

Table 1 show the total size of the various test data sets. The voxel data always consists of color, normals and positions. As we can see, the voxel data adds roughly 50% overhead for data sets which contain only position. For David, the compressed normals and colors in the level-of-detail representation drastically cut down the required memory for the voxel-data compared to the geometry, which uses floating-point attributes.

We have also compared the performance using different MSAA levels on the Atlas dataset, which can be seen in Figure 11. In Table 2, we can see that the rendering performance is nearly independent of the anti-aliasing

GPU	No AA	2×AA	4×AA	8×AA
R9 290X	17	17	18	21
HD 7970	22	22	23	24

Table 2: Rendering times in ms for Atlas dataset onto a 1280×720 viewport with different anti-aliasing levels and graphics cards. Both cards show only minimal impact when MSAA is enabled.

level, varying between 10% and 25% overhead for 8× anti-aliasing.

6 CONCLUSION

In this paper we have proposed an efficient, scalable rendering technique for triangle models using voxels as the level-of-detail representation. Unlike other approaches, our method uses the hardware rasterization units and can thus be easily integrated into existing rendering pipelines. Using the rasterizer also enables us to use multi-sample anti-aliasing with minimal impact on the performance. We have also shown how our level-of-detail can be quickly computed even for very large data sets.

In the future, we would like to incorporate secondary effects into our framework. Our surface voxel representation can be easily used as the starting point for a voxel ray-tracing system. For example, an acceleration structure can be built externally referencing the surface voxels, which could be then used to resolve secondary effects like shadows or ambient occlusion.

We would like to investigate how per-face data can be used to improve the level-of-detail quality. As seen in Figure 3, we can easily store per-face data. For very complex models, it might be necessary to switch voxels to per-face normals and colors. For example, if two different-colored walls are merged, the resulting color should not be averaged; instead, the voxel should be adjusted to use per-face data.

ACKNOWLEDGEMENTS

We would like to thank the CNR-ISTI Visual Computing Laboratory, the Visual Computing Group in Pula and the Museum Galleria dell'Accademia, Superintendency for the Polo Museale Fiorentino (Florence, Italy) for providing the colored David data set.

7 REFERENCES

[BK03] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern gpus. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, PG '03, pages 335–, Washington, DC, USA, 2003. IEEE Computer Society.

[BLD13] Jeroen Baert, Ares Lagae, and Philip Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the Fifth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, 2013. To appear.

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. In *Computer Graphics Forum*, volume 23, pages 615–624. Wiley Online Library, 2004.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22. ACM, 2009.

[HL79] Gabor T. Herman and Hsun Kao Liu. Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Image Processing*, 9(1):1–21, 1979.

[KCY93] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *Computer*, 26(7):51–64, July 1993.

[Lai13] Samuli Laine. A topological approach to voxelization. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering 2013)*, 32(4), 2013.

[LK11] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[Mer12] Bruce Merry. CLOGS. MIT License, 2012.

[PGC11] Ruggero Pintus, Enrico Gobbetti, and Marco Callieri. Fast low-memory seamless photo blending on massive point clouds using a streaming framework. *J. Comput. Cult. Herit.*, 4(2):6:1–6:15, November 2011.

[RCBW12] Florian Reichl, Matthäus G. Chajdas, Kai Bürger, and Rüdiger Westermann. Hybrid Sample-based Surface Rendering. In *VMV 2012: Vision, Modeling & Visualization*, pages 47–54, Magdeburg, Germany, 2012. Eurographics Association.

[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.