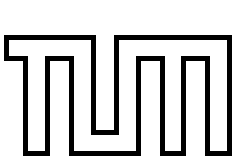


Technische Universität München
Fakultät für Informatik
Lehrstuhl für Computer Graphik und Visualisierung

A voxel-based visualization pipeline for high-resolution geometry

Matthäus G. Chajdas

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Th. Neumann
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. R. Westermann
	2.	Univ.-Prof. Dr. M. Stamminger Friedrich-Alexander-Universität Erlangen-Nürnberg

Die Dissertation wurde am 04.11.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.03.2015 angenommen.

To my family and friends

ABSTRACT

Rendering of extremely high-resolution meshes and iso-surfaces from complex simulations remains one of the major challenges in 3D visualization. In recent years, new data acquisition techniques like 3D scanning have drastically increased the amount of available high-resolution data. This data requires quick and efficient pipelines for visualization. Unfortunately, current approaches for rendering of very large models and meshes require expensive, time-consuming preprocessing to generate level-of-detail simplifications of the input data. New approaches, which combine fast pre-processing and efficient visualization are hence necessary.

In my work, I present a novel processing and rendering pipeline built on a resampling strategy. The input data – either triangle-based meshes or volumes – is first converted into a voxel representation. The resampling is very fast, even for large data sets, and is a key enabler for efficient simplification. Instead of using the source data, my simplification is performed on the much more compact voxel representation. Even though the data has to be converted first, the complete process is orders of magnitude faster than existing approaches. It allows even the most complex models, comprised of nearly one billion triangles, to be prepared for visualization within minutes on commodity hardware.

The runtime consists of a novel, GPU based renderer which combines the voxel based simplification and the original source data. Unlike other approaches, which rely on raytracing using the GPU compute units, I makes extensive use of the rasterization pipeline in my system. This allows me to obtain efficient, high-quality rendering with very low memory requirements. In particular, by using the rasterizer, is possible to take advantage of hardware-accelerated anti-aliasing, which is crucial to obtain a smooth and stable rendering of high-resolution geometry. The renderer also integrates occlusion culling, level-of-detail and streaming into a single framework. Besides the advantages for interactive rendering, the underlying data structure in conjunction with my rendering approach also enables efficient run-time manipulation. For instance, it is possible to remove or add new geometry without having to perform costly data structure rebuilds.

As mentioned above, voxel representations can also be rendered using GPU raytracing. This typically requires the creation of an acceleration structure. Previously, octrees have been the only data structure used for voxel raytracing due to the simple integration of level-of-detail. In my work, I have adopted several well-known data structures

which have been used for triangle raytracing to voxel raytracing and analyzed them in detail. I found that bounding volume hierarchies, combined with an optimized, voxel-specific traversal routine, can outperform the currently used octree based rendering techniques. In my investigation, I have identified key metrics for this behavior through an extensive analysis of the execution characteristics on modern GPU hardware.

I have also developed a highly scalable, memory and time-efficient voxelization for triangle meshes. It builds an adaptive data structure while resampling, which reduces memory usage and also enables parallelization. The re-sampler can process extremely detailed meshes very quickly. Beside speed, it also provides guarantees on the generated topology. Depending on the user needs, it can generate conservative, 6- or 26-separating voxelizations.

ZUSAMMENFASSUNG

Die Visualisierung von sehr hoch aufgelösten Modellen und Iso-Flächen von komplexen Simulationen ist immer noch eine der großen Herausforderungen in der 3D Visualisierung. In den letzten Jahren hat sich die Anzahl an verfügbaren, hoch-aufgelösten 3D Modellen durch neue Akquise-Techniken wie 3D Scanner stark erhöht. Zur Visualisierung dieser Daten sind schnelle und effiziente Rendering-Pipelines notwendig. Unglücklicherweise erfordern die aktuellen Ansätze zur Darstellung sehr großer Modelle teure, zeitaufwändige Vorverarbeitungs-Schritte um eine Vereinfachung der Datensätze zu generieren. Neue Ansätze, die eine schnelle Vorverarbeitung mit effizienter Visualisierung kombinieren, werden somit benötigt.

In meiner Arbeite stelle ich eine neue Verarbeitungs- und Visualisierungs-Pipeline vor, die auf einer Resampling-Strategie basiert. Die Eingabe – entweder ein Dreiecksbasiertes Modell oder ein Volumen – wird dabei zuerst in eine Voxel-Repräsentation überführt. Diese Konvertierung ist sehr schnell, auch für große Datensätze, und ermöglicht eine effiziente Vereinfachung. Anstatt auf den Quelldaten wird die Vereinfachung nun auf der Voxel-Repräsentation durchgeführt. Obwohl die Daten erst konvertiert werden müssen, ist der gesamte Prozess um Größenordnungen schneller als existierende Ansätze. Damit wird es möglich, selbst die komplexesten Modelle, bestehend aus fast einer Milliarde Dreiecken, innerhalb von Minuten auf Standard-Desktop-System darzustellen.

Die Darstellung selbst wird von einem neuen, GPU basierten Renderer übernommen, der sowohl die vereinfachten Daten als auch die Quelldaten kombiniert. Anders als bisherige Ansätze, die sich auf GPU Raytracing stützen, das auf den Compute-Units der GPU durchgeführt wird, nutzte ich in meinem System die Rasterisierungs-Hardware. Dies ermöglicht mir, eine effiziente, hoch-qualitative Darstellung bei niedrigem Speicherverbrauch zu erreichen. Zudem kann ich die Hardware-Beschleunigung für Anti-Aliasing nutzen, das für eine saubere und stabile Darstellung der hoch-aufgelösten Geometrie unerlässlich ist. Der Renderer kombiniert zudem Verdeckung, Level-of-Detail und das Streaming in einem einheitlichen System. Neben den Vorteilen bei der interaktiven Darstellung erlaubt die darunter liegende Datenstruktur, zusammen mit meinem Rendering-Ansatz, die Geometrie effizient zur Laufzeit zu verändern. Zum Beispiel kann Geometrie hinzugefügt oder entfernt werden, ohne Beschleunigungsstrukuten aktualisieren oder neu erstellen zu müssen.

Wie bereits beschrieben, kann die Voxel-Repräsentation auch mittels GPU Raytracing dargestellt werden. Dies erfordert typischerweise die Erstellung einer Beschleunigungsstruktur. Bisher wurden dafür nur Octrees benutzt, da diese eine einfache Integration von Level-of-Detail Vereinfachungen erlauben. In meiner Arbeit habe ich mehrere bekannte Datenstrukturen aus dem Bereich des Dreiecks-Raytracing für Voxel-Raytracing adaptiert und im Detail analysiert. Dabei stellte ich fest, dass Bounding Volume Hierarchies, zusammen mit einer für Voxel optimierten Traversierung, die aktuell verwendeten Octrees bei der Ausführungsgeschwindigkeit überlegen sein können. In meiner Untersuchung habe ich die Haupt-Metriken für dieses Verhalten durch eine umfangreiche Analyse, die sich auf die tatsächliche Ausführung der verschiedenen Algorithmen auf einer modernen GPU konzentriert, identifiziert.

Ich habe ebenfalls eine hoch-skalierbare, Speicher- und Laufzeiteffiziente Voxelisierung für Dreiecks-Modelle entwickelt. Diese baut eine adaptive Datenstruktur während der Konvertierung, die sowohl den Speicherverbrauch minimiert als auch eine Parallelisierung des Algorithmus zulässt. Der Voxelisierer kann auch sehr hoch aufgelöste Modelle schnell verarbeiten. Neben der hohen Geschwindigkeit bietet er zudem Garantien zur generierten Topologie. Je nach Anforderungen des Benutzers ist es möglich, konservative, 6-separierende oder 26-separariende Voxelisierungen zu erzeugen.

ACKNOWLEDGMENTS

I gratefully acknowledge the support of all of the people who made this thesis possible. First and foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Rüdiger Westermann for offering me the great possibility to pursue research in the field of scientific visualization. I am very grateful for his guidance, for his commitment to my work, and for the numerous discussions. I also want to thank the co-authors of my papers, Florian Reichl, Andreas Weis, Matthias Reitinger, Julian Amman, Morgan McGuire and David Luebke for contributing their suggestions and ideas. I have always enjoyed working with them.

Furthermore, I would like to thank my current and former colleagues, Florian Reichl, Stefan Auer, Kai Bürger, Shunting Cao, Marc Treib, Ismail Demir, Christian Dick, Florian Ferstl, Roland Fraedrich, Raymund Fülöp, Stefan Hertel, Hans-Georg Menz, Mihaela Mihai, Tobias Pfaffelmoser, Marc Rautenhaus, Matthias Reitinger, Jan Sommer, Nils Thuerey, Mikael Vaaraniemi, and Jun Wu, who have always been available for discussions.

I am enormously thankful to my parents, my girlfriend Nicole and my friends for giving me all the support and encouragement that I needed during this time.

CONTENTS

1 INTRODUCTION

1.1 OUTLINE

The rendering of very large and complex data sets is one of the major problems in computer graphics. With recent advancements in acquisition and modeling tools, meshes consisting of tens to hundreds of million triangles can be easily obtained. This rapid growth in data set size mandates new visualization techniques to quickly inspect the scans, allowing users to check the scan quality and spot errors early.

On the visualization side, graphics hardware has and is still becoming more powerful every year. This is driven by the game industry and the ever increasing demand for higher fidelity graphics. Over the years, graphics cards have become massively parallel architectures with increasingly programmable compute units. This trend is expected to continue in the following years. Besides flexible, programmable units, graphics cards still sport a large variety of fixed function elements to handle tasks like rasterization, texturing and blending. Unfortunately, memory size on GPUs as well as triangle throughput has not scaled as rapidly as the raw compute performance. As such, it is not possible to simply load meshes consisting of hundreds of million triangles on even the most recent GPUs and visualize them interactively.

On the CPU size, Moore predicted that improvements in circuits would allow to double the number of transistors on a chip every year. Looking back, we now know that the actual rate is closer to once every two years, but the exponential improvement did indeed occur. At the beginning of Moore's law, the number of transistors equaled performance, with twice as many transistors, the serial execution performance of a given chip would also double. In recent years, due to thermal and power constraints, serial performance is no longer a target of chip manufacturers. Instead, the additional transistors are used to increase the number of execution units. While the total performance is still scaling, it requires parallel algorithms to take advantage of modern multi- and many-core architectures. GPUs are spearheading this process by relying on tens of processing cores, each using very wide SIMD units.

Unfortunately, memory and disk I/O speed did not follow Moore's law. Even though modern machines have far more total capacity, the relative performance of the memory and disk subsystem has actually decreased over the years. This trend is expected to

continue, as power usage constraints the amount of external bandwidth that can be provided to a chip.

These two trends require new approaches to process data. Algorithms which rely on random access to the data sets are no longer an adequate solution. Even for in-core algorithms, high locality and predictable access patterns are required for optimal performance.

Additionally, for very large data sets, the ability to efficiently stream data is becoming more and more important. It is no longer feasible to load a significant part of the data set before displaying it to the user, as this can easily take several minutes. Instead, low-resolution previews have to be quickly loaded and refined on-demand.

In this work, I present an optimized visualization pipeline which has been designed from the beginning for very efficient usage of parallel compute units, disk and memory bandwidth. This is possible by the combined design of both the preprocessing and the rendering, which are closely tied to each other for best performance.

1.2 CONTRIBUTIONS

Efficient, scalable voxelization with topology guarantees and fast simplification. I present a highly scalable voxelization pipeline for large triangle meshes. It can process objects with hundreds of millions of primitives on commodity hardware in minutes, even at very high output resolutions. Besides performance, the method presented in this work provides topological guarantees. In particular, it can generate minimal sets of voxels required for rendering, which is crucial to reduce memory usage and efficient display. Combined with the very fast simplification, the rasterizer forms the core foundation for the visualization pipeline.

Scalable, high-quality rendering. I present a novel renderer which combines the voxel data with source geometry and enables efficient viewing of very large meshes. The new renderer is optimized for low memory usage, high image quality and immediate exploration of the scene. Combined with a fast preprocessing step, it enables the user to inspect extremely detailed meshes within minutes of the acquisition. My rendering framework relies purely on hardware rasterization, which allows for hardware-accelerated anti-aliasing. This is a crucial element to obtain high-quality rendering.

Analysis of GPU execution characteristics for ray-tracing. Even though GPUs are comprised of large amounts of simple “shaders”, the complete architecture has become extremely complex due deep cache hierarchies, predicate-based execution and wide vector units. This makes it necessary to re-evaluate and analyze algorithms directly on the GPU to identify all elements that contribute to the final performance

of an algorithm. To this end, I have performed an extensive analysis of GPU voxel ray-tracers and measured the results on actual hardware, exposing the key algorithmic design choices that are required for high performance on such architectures.

1.3 LIST OF PUBLICATIONS

Several of the research results that are presented in this thesis have been originally published in the following peer-reviewed conferences or journal articles:

1. CHAJDAS, MATTHÄUS G. AND MCGUIRE, MORGAN AND LUEBKE, DAVID: Subpixel Reconstruction Antialiasing for Deferred Shading *Symposium on Interactive 3D Graphics and Games*, 2011, 15–22 doi.acm.org/10.1145/1944745.1944748.
2. CHAJDAS, MATTHÄUS G. AND REITINGER, MATTHIAS AND WESTERMANN, RÜDIGER: Scalable rendering for very large meshes *WSCG 2014*, 2014, 77–85.
3. CHAJDAS, MATTHÄUS G. AND WESTERMANN, RÜDIGER: A quantitive analysis of acceleration structures for voxel raytracing *Pacific Graphics* 2014.

Additionally, some of the algorithms developed in the course of this thesis have been incorporated into the VOTA 3D modeling software.

2

FUNDAMENTALS

In this section, I will cover several fundamental concepts related to my work. First of all, as I present a voxel based pipeline, I will introduce voxel representations. This also includes a brief interlude on iso-surfaces. After this, I will cover the two main rendering techniques for image generation: Rasterization, which is the primary technique used by games, and raytracing, which is the standard technique in film rendering. Finally, I will provide a brief explanation of the programming model on massively parallel architectures like GPUs and how it affects algorithm design.

2.1 VOXEL REPRESENTATIONS

Voxels are the 3D equivalent of pixels in 2D. Similar as in 2D, voxels are placed on a 3D grid with typically uniform spacing in all dimensions. Just like a pixel corresponds to a 2D square, a voxel corresponds to a 3D cube.

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 2D, the two main representations of images are vector graphics and bitmaps. In 3D, surfaces like triangles or NURBS are the equivalent of vector lines in 2D, and voxels are the equivalent of bitmaps. They share the similar advantages and drawbacks as pixels in 2D. Being a *sample* based representation, continuous, smooth surfaces can only be represented approximately using voxels. Depending on the resolution, the resampling can lead to highly visible aliasing artifacts (see also Figure 1).

Mathematically, voxels can be defined as follows [COK95, Lai13]: Let $\mathbb{Z}^3$ be the subset of $\mathbb{R}^3$ which contains only the points that can be represented using integer coordinates. This subset is also known as the *integer lattice* or *grid*. I will denote points on the grid as p_{ijk} with $i,j,k \in \mathbb{Z}$. For a point p_{ijk}, let $V(p_{ijk})$ be the set of points $p = (x,y,z)$ with $i \leq x \leq i+1$, $j \leq y \leq j+1$ and $k \leq z \leq k+1$, that is, all points in an unit cube with the origin on the grid point. The union of all $V(p)$ now tessellates $\mathbb{R}^3$, and the interiors of all $V(p)$ are disjoint. The set of points $V(p_{ijk})$ is what I have previously introduced as a *voxel*.

Given a mesh represented by a surface, it can be voxelized in two different ways: Either as a *Solid* or as a *surface* voxelization. A surface voxelization contains only the surface, that is, voxels are only created at points close to the original surface of the model. A solid voxelization consists of the surface of an object *and* the interior. It is closely related to a volume representation of the mesh. A solid voxelization can be only computed exactly for a two-manifold mesh, that is, an oriented mesh with no holes or interior faces. It is always possible to generate a surface voxelization from a solid voxelization, while the converse is generally not true.

The size of a voxel representation is also directly tied to the resolution and the volume extents. This is a major difference to surface based meshes, where doubling the size, that is, the spatial extents, requires no additional storage as it is tied to the number of vertices. For a solid voxelization, doubling the resolution will require $2^3 = \mathcal{O}(n^3)$ more memory. The factor improves to $2^2 = \mathcal{O}(n^2)$ for sufficiently smooth surface voxelizations, as those represent a 2D manifold within the 3D space.

A key property of a voxel representation is that it has uniform resolution and all voxels are independent of each other. Unlike triangle meshes, which may have dense and sparse areas, voxels are placed on a grid with equal spacing and no topological information. This regular structure simplifies various computations and manipulations. An example is coloring: For a triangle mesh, this is typically performed by applying textures with roughly uniform world-space resolution, while for a voxel mesh, the voxels can be used directly.

Manipulation is also much easier, in particular, if the topology is affected. A prime example is cutting a hole. For a triangle mesh it is necessary to produce new edges and to connect them with the surrounding mesh. For voxels, it is enough to simply remove those inside the hole. As the voxels carry no topological information, no neighborhood updates are necessary. For solid voxelizations, CSG operations can be also easily computed by bitwise computations on the voxel grid.

Another advantage of voxels is the trivial simplification. Similar to mip-maps for 2D textures, 3D volumes can be easily filtered and simplified. Typically, a simplification factor of two is used, which combines 2^3 voxels into a single voxel at each step.

Finally, it is very easy to stream voxel-based data. As there is no connectivity between voxels, an object can be easily broken apart into separate elements and loaded on demand. Combined with a level-of-detail simplification, this makes it possible to stream in very large 3D scenes with low memory usage.

Unfortunately, currently nearly all 3D content is created as triangle meshes. In order to convert those into a faithful voxel representation, a *voxelization* is necessary. In the next section, I will cover several modern approaches for the voxelization of meshes.

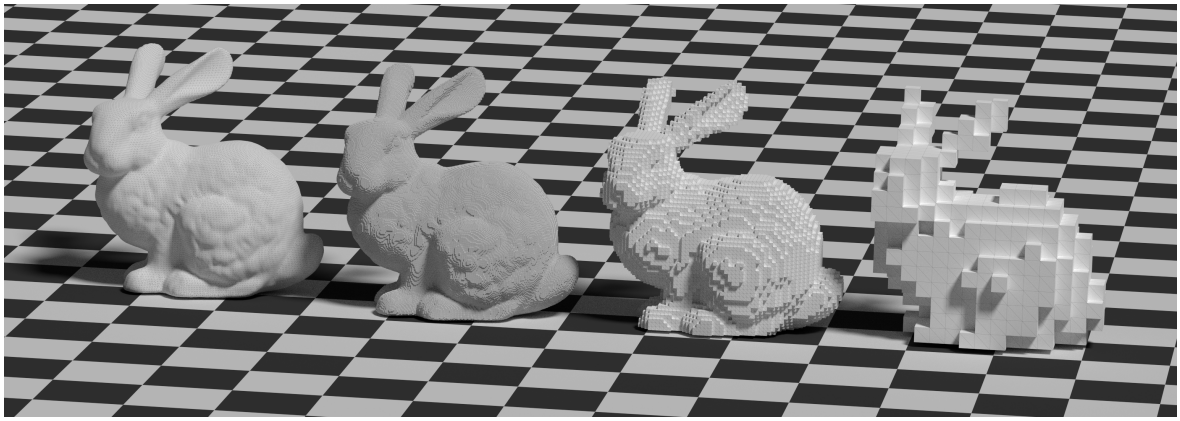
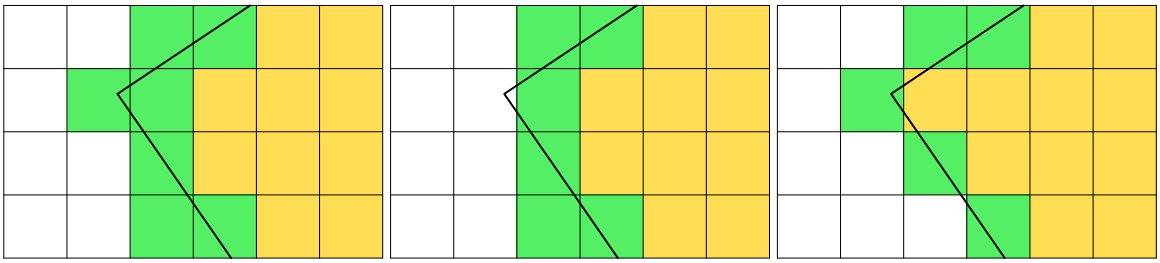


Figure 1: A voxelization of the Stanford Bunny dataset. On the left, the original input mesh, on the right, three voxelizations at different resolutions (256^3, 64^3 and 16^3).



(a) Conservative (b) 8-separating (c) 4-separating

Figure 2: Connectivity and rasterization. The thick black line is rasterized and produces the green set, which separates the white and yellow sets. The conservative rasterization, which covers all pixels that contain the line, results in a superset of both an 8-separating and 4-separating set.

2.1.1 *Topology*

A key property of a voxelization is the connectivity/separability of the resulting set. First of all, we have to introduce a few definitions [Lai13, HYFK98, COK95, COK97] before we can discuss the connectivity of a voxelization. An N_k-path – $\prod_k = (V_0, \ldots, V_n)$ – is a set of voxels such that $V_{i+1} \in N_j(V_i)$, that is, every two consecutive voxels are N_k-adjacent. If a set of voxels is N_k connected, the set is k-connected.

In 3D, a voxel has 26 neighbors. Of these neighbors, 6 share a face, 12 share and edge and 8 share one of the vertices. We can now define three adjacency configurations: If two voxels share only a face, we call them 6-connected; if either a face or edge is shared, 18-connected, and if they have a common face, edge or vertex, 26-connected:

$$N_6(V) = (x \pm 1, y, z) \cup (x, y \pm 1, z) \cup (x, y, z \pm 1) \quad (1)$$
$$N_{18}(V) = \{x + i, y + j, z + k\} - (x, y, z), i, j, k \in -1, 0, 1, i * j * k = 0 \quad (2)$$
$$N_{26}(V) = \{x + i, y + j, z + k\} - (x, y, z), i, j, k \in -1, 0, 1 \quad (3)$$

In 2D, a pixel has 8 neighbors, and the corresponding adjacency configurations are thus 4-connected if an edge is shared and 8-connected if either an edge or vertex is shared:

$$N_4(V) = (x \pm 1, y) \cup (x, y \pm 1) \quad (4)$$
$$N_8(V) = \{x + i, y + j\} - (x, y), i, j \in -1, 0, 1 \quad (5)$$

These configurations can be seen in Figure 2. The 2D N_4 connectivity corresponds to N_6 in 3D, and analogous for N_8 and N_{26}.

Now let S be a surface embedded in $\mathbb{R}^3$ such that $\mathbb{R}^3 - S$ yields exactly two separate sets I and O. Let I^d and O^d be the voxels in I and O.

By voxelizing S, we obtain a set of points S^d which is disjoint with I^d and O^d. Now, we define separability and connectivity as follows: We call S_k^d a k-separating subset of S_d if there is no k-connected path such that $V_0 \in I^d, V_n \in O^d$, and $\prod_k \cap S_k^d = \emptyset$.

The definitions so far can be easily translated into a 2D context. In this case, $S \in \mathbb{R}^2$, $S^d \in \mathbb{Z}^2$, and V, I, O and $\prod_k$ similarly defined as before.

Depending on the target application, a different separability may be required. Notice that a conservative voxelization S_c^d, which includes all voxels through which the surface S passes, will be a proper superset of both S_6^d and $S_2 6^d$. Put another way, it is always possible to obtain a k-separating voxelization from S_c^d by removing voxels which do not contribute to the separability property.

For a voxelization algorithm, we are interested in finding the minimum k-separating set of voxels. As mentioned above, a conservative voxelization will always be both 6- and 26-separable. While this allows us to use a conservative voxelization in all cases where a 6- or 26-separable is required, it will generally contain many unnecessary voxels.

Using the definitions above, we can now determine the required connectivity to obtain crack-free rendering. A view ray is a 6-connected path through the voxel grid [COK97]. It is thus sufficient to produce a 6-separating voxelization to obtain a correct rendering.

This is insofar important as the 6-separating voxelization is very sparse compared to 26-separating or conservative voxelizations, resulting in reduced memory usage.

2.1.2 *Polygon to voxel conversion*

When converting polygon meshes to voxels, we are usually interested in a surface voxelization. For closed meshes, a solid voxelization can be later derived by flood-filling the interior. While there are algorithms which can directly compute a solid voxelization, I will focus first on the more common case of surface voxelization.

The process of converting a polygon mesh to a voxel grid is not unlike the rasterization required in the normal graphics pipeline. There, the pixels covered by a triangle are evaluated in 2D, while the voxelization requires to dtermine the voxels covered by a triangle in 3D.

Very early works related to polygon voxelization adapted existing 2D scanline algorithms to the 3D context [KS87]. Similar to the 2D implementations, the edges are first rasterized in 3D and then "scanlines" between edges are filled. These algorithms did not guarantee the separability and connectivity for meshes which consist of multiple polygons. For meshes, problems arise in areas around vertices and edges, where gaps must be avoided while computing a minimal voxel set.

An early work which focused on topological correctness is [HYFK98]. In their work, they introduce a distance-based computation which can guarantee 26 and 6 separability. However, minimality is only guaranteed for individual polygons. Moreover, the algorithm is sensitive to the input tessellation. That is, if a plane is subdivided, the resulting voxelization, albeit topological correct, will be different.

Due to the similarity to the 2D rasterization problem, there has been a significant amount of work to exploit the GPU rasterization units to accelerate voxelizations. The first technique which used the triangle rasterization hardware of a GPU to directly compute a surface voxelization is [ED06]. They use the rasterizer to process every triangle and determine the xy coordinate in the output grid. For z, the discrete depth of the currently processed triangle is computed in the fragment shader and stored into a 4×8 bit color target. The output is then blended using hardware-supported binary `or` blending. The x and y resolutions can be chosen freely, while the z resolution is limited by the maximum number of bits that can be blended.

Their algorithm is not suited for the voxelization of complete scenes. In order to bypass the depth resolution limitation, the geometry must be processed multiple times. Moreover, the algorithm suffers from various artifacts which stem from the single per-

spective rendering. Polygons which are parallel to the view direction for instance are omitted completely. It is thus only suitable for approximate voxelizations.

Subsequently, the various shortcomings of the algorithm have been addressed in different works. The problem of missing voxels has been resolved in [ZCEP07]. Compared to previous work, the new implementation was the first one which could guarantee a conservative voxelization. No other guarantees on the connectivity were provided though.

An extension of [ED06] has been presented in [ED08], which adds solid voxelization. By storing the depth into a bit-volume, they can quickly compute the occupancy using an `XOR` operation. Moreover, they provide a "density" voxelization which results in a smooth density map instead of a binary solid voxelization. In their implementation, they compute a solid voxelization at higher resolution and downsample it to obtain a filtered density.

On modern GPUs, an alternative is to use the compute units and perform the triangle/box overlap test there [SS10]. In their work, they present an algorithm for surface and solid voxelization. The focus is on the surface voxelization itself, while the solid voxelization is built on top as an extension.

The general idea is to start one thread per triangle, iterate over all potentially covered voxels and evaluate the coverage test for each of them. As multiple triangles are processed concurrently, this requires the use of atomic operations – specifically, atomic `or` to update the bit masks – per voxel, which is a costly operation. This has been optimized by delaying the write until multiple voxels have been processed.

Solid voxelization requires only small modifications. The key insight is to treat each surface voxel as a "in/out" flip along a voxel column. To this end, they identify all voxels affected by the current triangle and flip their "in/out" bit. This requires a clean, closed mesh. One major problem with running one thread per triangle in this case is load-balancing. Large triangles may take much longer to finish, leading to underutilization of the compute units. [SS10] optimized this by using tile-based voxelization. During a preprocess, triangles are sorted into tiles, which are then processed by a set of threads. In this case, the threads are started per tile and loop over the triangles, which avoids the problem of a single triangle stalling a thread for a long time.

[SS10] also show how a sparse, solid voxelization can be generated directly, instead of starting from a full solid voxelization and identifying sparse regions. The key insight is to build the octree bottom-up by identifying the leaf nodes first, and then propagating the "in/out" bits through the hierarchy. In their approach, they use two voxelization passes. In the first pass, the model is voxelized into a grid with half the target resolution. Nodes in this grid correspond to the last level of interior nodes in the octree. From

these nodes, the octree is built, and in a second pass the generated voxels are placed directly in the tree.

Finally, both conservative and 6 separating tests have been presented. As the implementation does not rely on fixed-function hardware, this can be easily accomplished by a small modification to the triangle/box test.

[Pan11] improves on this work by addressing two shortcomings of the original approach. Compared to [SS10], [Pan11] adds:

- Blending: Values can be computed for each triangle and blended together. This allows for example to obtain smooth normals.
- Load balancing: The original algorithm is optimized for equally sized, ideally small triangles.

The load balancing is improved by utilizing a coarse and fine raster phase, in addition to a separate path for large triangles. For small triangles, [Pan11] performs a coarse rasterization to identify the target tiles, and then a fine rasterization for each tile to generate the voxels. During the fine raster stage, the threads process one tile and loop over the triangles, similar to the solid voxelization in [SS10].

For large triangles, a separate kernel is used which starts multiple threads per triangle. In this case, the threads loop over the bounding box of the triangle. By combining both the small and the large triangle kernel, [Pan11] is able to handle scenes with highly irregular triangle sizes efficiently.

Another addition is the support of attributes. They key insight is that the fragment emission can be separated from the blending. First, all fragments for all triangles are generated, and stored unordered into a global output buffer. In a second step, the fragments are sorted using an efficient radix sort by their voxel id and eventually blended together.

Recently, the idea of [Pan11] has been applied to rasterization-based voxelization, yielding the hybrid rendering pipeline of [RB13]. In their work, instead of using the compute units, the hardware rasterizer is used again for the actual voxelization. The key idea is to perform the classification at the same time as the processing, and only defer large triangles to a second step.

In their work, everything is implemented using the standard rendering pipeline. The classification is performed using the geometry shader. If a triangle is found to be sufficiently small, it is processed directly in the geometry shader stage. Otherwise, it is forwarded to the second pass, which again uses the geometry shader to create a conservative bound and then uses the fragment shader to perform the actual voxelization.

There are also techniques which are not based on the rasterization of primitives into a volume grid. Wavelet rasterization [MS11] rasterizes the primitives directly into a wavelet. At the end, a volume is reconstructed from the wavelet, from which a surface voxelization can be obtained by iso-surface extraction. Compared to the previously described algorithms, the wavelet rasterization is more robust and can various errors in the input geometry.

As mentioned, solid voxelization requires a closed object. Most real-world meshes however contain additional geometry, small holes, duplicated faces and other problematic areas, making it nearly impossible to obtain a consistent voxelization. This has been recently resolved by the introduction of generalized winding numbers [JKSH13]. This technique is aimed at the generation of solid voxelizations and resolves many previously ambiguous cases.

As the focus of my work is on a rendering pipeline, I will focus on surface voxelizations, which can be directly computed from a mesh. For solid voxelizations, a separate iso-surface extraction step is necessary, as the output is a density volume. This makes them less suited if preprocessing performance is paramount.

2.2 ISO-SURFACES

Iso-surfaces can also be easily extracted and converted into a voxel-based representation. In general, a volume is a 3D scalar field, defined as a function $f : \mathcal{D} \to \mathbb{R}$ [Mie09]. For every point $p = (x, y, z)$ inside the domain, there is a real number $f(p)$ associated with it. Similar to the definition of the voxel grid, we can now define $f_d(p_d), p_d \in \mathbb{Z}^3$. This a *discretization* of f which is only defined on the integer grid.

An iso-surface is a surface comprised of all points p with a constant value v, that is $S = p \in \mathcal{D} | f(p) = v$. As the scalar field f is continuous, this surface is well defined. Unfortunately, it is not possible to define an iso-surface directly on f_d, which is no longer continuous and only covers an enumerable subset of $\mathbb{R}^3$.

A very simple approach to still obtain an iso-surface from the discrete grid f_d is the cuberille algorithm [Liu77, HL79]. Instead of identifying points on the surface, it finds the cells through with the iso-surfaces passes and generates cubes for each such cell. If the cell value is above the iso-value and a neighbor is below, the cell is determined to be part of the iso-surface. This results in an approximate iso-surface.

The correct solution to resolve the non-continuity is to introduce a new scalar field $f'_d(p)$ as by *interpolating* between the discrete grid points $f_d(p_d)$. On the interpolated grid, it is now possible to define an iso-surface again which will closely match the original iso-surface.

There are two major ways to approach iso-surface visualization: Direct rendering of the iso-surface from the volume or indirect rendering, which extracts the iso-surfaces first and renders the extracted surface.

Direct rendering has the advantage that a high-quality intersection test can be implemented, which computes the exact hit point of a ray with the iso-surface. The disadvantage is that direct rendering requires the volume to be available in memory to allow for interpolation. Additionally, the rendering itself requires an expensive ray-traversal which has to interpolate the volume at each single step. Depending on the required quality, the interpolation can be very costly [MKWF04]. The traversal can be optimized by providing hierarchical acceleration structures, which bound the iso-surface and allow for fast empty space skipping.

An alternative approach is to extract the surface in a pre-process. The simplest solution is to use linear interpolation along the grid to determine the interception points with the iso-surface and then connect those using straight lines. In 2D, this will yield the *marching squares* algorithm (see Figure 3). Marching squares has 16 possible configurations, as there are four vertices which can be either above or below the iso-value. Five of them are unique, while the rest are rotations.

The equivalent to marching squares in 3D is called *marching cubes* [LC87]. As there are eight vertices, the total number of configurations is 256. Similar to marching squares, there are only 15 unique configurations; the remaining ones can be obtained using rotations. Implementations of marching cubes typically compute the vertices along each edge and use an index table to create the final topology. This makes marching cubes very efficient to implement. Moreover, by splitting large volumes into separate parts, it can be also easily parallelized.

Both marching squares and marching cubes suffer from ambiguous configurations, which can lead to incorrect surfaces. For example, in Figure 3, the first configuration in the second row is assumed to be "outside" in the center of the cell. Alternatively, the two corner vertices could be also connected, yielding a different configuration. Similar cases appear for marching cubes. Consider the first case in the third row in Figure 4: Here, the two "inside" vertices are assumed to be separated. An alternative configuration would connect those two vertices. These ambiguities can be resolved using an asymptotic decider [NH91, LB03]. It takes the behavior of the iso-surface inside the cell into account and allows to resolve such cases.

Even without ambiguities, marching cubes is still prone to cracks in the geometry. These can be only resolved if a cube and its neighborhood is considered to ensure a consistent topology [LLVT03].

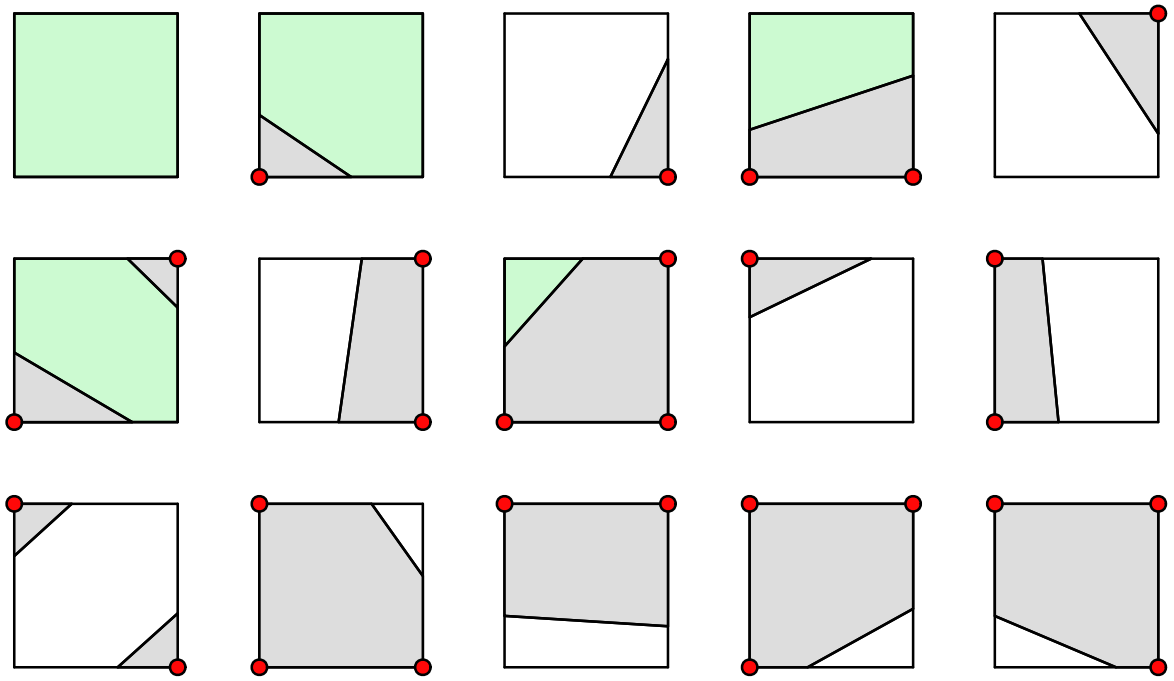


Figure 3: 15 configurations of the marching squares algorithm. Red dots mark vertices above the iso-value. The last configuration, where all vertices are inside, has been omitted. Several of the configurations can be obtained through rotation; the unique configurations are marked in green.

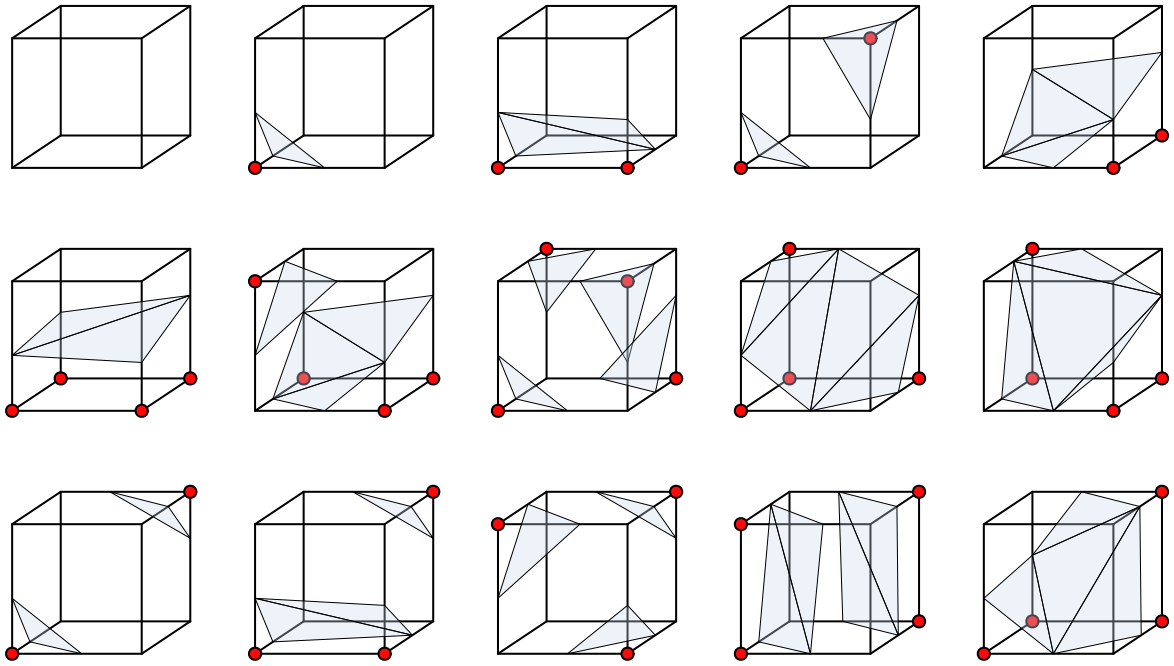
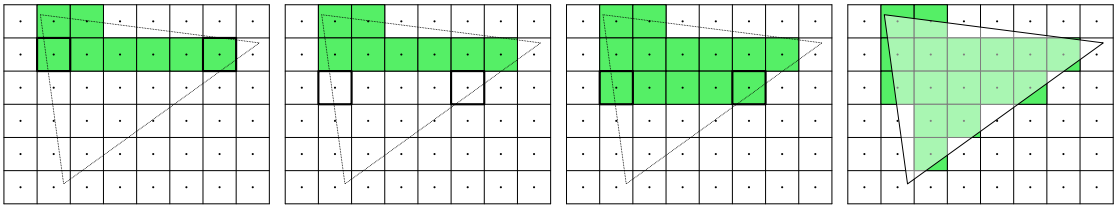


Figure 4: The 15 unique configurations of the marching cubes algorithm. Red dots mark vertices above the iso-value [LC87].



(a) Scanline 2 (b) Span update (c) Fill span (d) Final result

Figure 5: Rasterizing using span rendering. In the second image, the scanline is advanced, updating the span begin/end pixels. In the third image, the scanline is filled. The right image shows the final result. Notice that only pixels inside the triangle are accessed by the algorithm.

Compared to direct rendering, the advantage of marching cubes and similar algorithms is that the surface is extracted only once and reused for rendering. Additionally, as the marching cubes are only stored along the surface, they can be relatively compact if the volume contains only a simple iso-surface. There are however multiple disadvantages as well. The linear interpolation along the grid cell edges produces piecewise linear surfaces, while a trilinear interpolation, which can be easily performed during ray-tracing, will yield smooth surfaces.

Another disadvantage is the very large size of a marching cubes mesh. For high-resolution iso-surfaces, the size of the mesh can rival or even exceed the size of the original volume representation. The Richtmeyr-Meshkov instability shown in Figure 19 requires 463 million triangles, which result in a mesh size of 5.3 GiB while the original volume requires 7.5 GiB. Finally, changing the iso-value requires re-extraction of the surface, which in turn requires the original volume to be present.

2.3 RASTERIZATION

The first approach to triangle rasterization is *span*-based scan-conversion [ST86, FvDFH90]. The basic idea is to decompose a triangle into spans, that is, horizontal segments. The implementation is straightforward: For a given triangle, the edges are sorted in y order and then processed top-to-bottom. At each scanline, the start and end points of the two current active edges are computed. This can be efficiently performed by a simple multiply-add per scanline if the DDA constants have been precomputed for each edge. The start and end points define a span, which is then filled from left-to-right (see also Figure 5). Along the way, the attributes are interpolated and written to memory. Processing along a span has the advantage that memory access is completely

linear. Moreover, span conversion only accesses pixels which are actually covered by the triangle, not wasting any computation on uncovered pixels.

Span conversion has been very popular in CPU based rasterization, in particular for games. Using low-level optimizations, it was possible to optimize the inner loop – which interpolates across a span – to a few CPU cycles [Abr97].

Unfortunately, span-based rasterization has various problems:

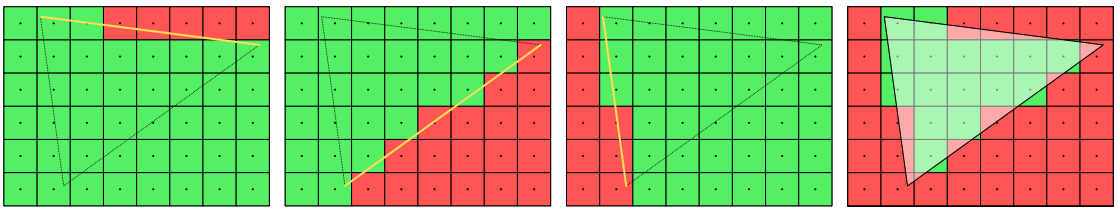
- It is difficult to add support for anti-aliasing if the sample locations are not on the pixel grid.
- The algorithm cannot be easily parallelized, as it relies on the DDA evaluation per-scanline and variable-length loops for each span.
- Fill-rules are difficult to implement.
- There are various corner-cases which have to be handled manually, for example, perfectly horizontal edges.

For a parallel evaluation, it is necessary to be able to evaluate the triangle coverage evaluation as well as the interpolation for multiple points independently. A possible implementation is to pre-compute edge and plane equations [FGH+85, FPE+89, FvDFH90] and evaluate them for a complete *tile* of pixels. For a triangle, we can define linear edge equations $F_0(x,y), F_1(x,y), F_2(x,y)$ of the form $F_i(x,y) = A_i x + B_i y + C$. Each edge equation separates the plane into half-space where $F_i(x,y) < 0$ or $F_i(x,y) > 0$. The equations can be easily determined if we transform the vertices t_0, t_1, t_2 of the triangle into homogeneous coordinates. The line connecting $t_0 = (x_0, y_0), t_1 = (x_1, y_1)$ can be computed as:

$$\vec{E_0} = \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} y_0 - y_1 \\ x_1 - x_0 \\ x_0 y_1 - y_0 x_1 \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \tag{6}$$

For any point $p = (x,y)$, we can now evaluate $\langle [x\, y\, 1]^T, [A\, B\, C]^T \rangle$; the sign will indicate in which half-space the point lies relative to the edge. A point is inside the triangle if $F_i(p) > 0\ \forall i$ (see also Figure 6).

A hardware implementation can easily evaluate multiple points in parallel. Per pixel, only two multiply-add instructions are necessary to evaluate the equation, followed by a sign comparison.



(a) Edge F_0 (b) Edge F_1 (c) Edge F_2 (d) Combined result

Figure 6: Rasterizing using edge equations. On the left three images, the edge equations have been evaluated for each edge individually. In the right image, the three coverage masks have been combined using and, resulting in the final coverage mask.

This can be further optimized if we take advantage of the linear form of the edge equations [Gie11]. Notice that for two points, $p_j = (x, y), p_{j+1} = (x + 1, y)$, two evaluations of the edge equations along a scanline will only differ by $F_i(p_0) - F_i(p_1) = A$. Similarly, the equations will differ by B in vertical direction. This makes it possible to pre-compute the constants $c_{ij} = iA + Bj$ for a grid of pixels. Evaluation now requires only to compute F_i once for the top-left corner of a tile, followed by additions. This makes it very cheap to implement the test in hardware [Gie11].

It is also not necessary to perform all the tests in floating-point precision. Instead, the rasterization is executed on a sub-pixel grid using integer precision. All the computations above are performed in screen-space coordinates, which makes it possible to transform the vertex positions into the pixel-grid first and evaluate all computations above using fixed-point integer arithmetic.

Unlike for scanline rasterization, it is easy to integrate fill-rules into edge-equation based rasterization. Fill rules determine whether a pixel is covered by a triangle if the edge passes exactly through the center of the triangle. As is easy to see above, all that is needed to make an edge inclusive is to change the equation $F_i(p) > 0$ to $F_i(p) \geq 0$, or, alternatively, adjust the constant C. As mentioned above, the computation is typically performed in integer coordinates, which makes the fill-rule adjustment a trivial addition of 1 to C. Using edge equations, it is also easily possible to change the test to a *conservative* coverage test [AMA05]. Even despite its simplicity, this functionality is not available on current hardware.

The whole pipeline using tile-based rasterization is as follows. Once a triangle is ready, it is transformed into screen space and the bounds are computed. Then, for each tile inside the bounding box, the edge equations are evaluated in parallel and the barycentric coordinates are computed.

Another optimization which can be performed is the to use a hierarchical rasterizer [Gre96, MM00, SCS+08, LK11b]. As explained above, the algorithm will become

very inefficient for large triangles, where many tiles will be completely empty. However, it is very easy to re-use the approach outlined above to compute a *coarse* rasterization first, where the edge equations are evaluated for a complete tile, followed by a *fine* rasterization phase where each tile is processed. Evaluating complete tiles during the coarse phase has the additional advantage that the initial values of F_i can be reused for the fine rasterization phase. Tiles can be also classified as covered, partially covered and empty; for fully covered tiles, the rasterization can be skipped completely.

GPUs provide dedicated hardware units to perform triangle rasterization. On modern GPUs, multiple rasterizers are present which can process one triangle each per clock cycle. Each rasterizer is built to evaluate multiple pixels or samples in parallel [Pin88, FBH+10]. Triangles are processed using large stamps, for example, 8×8 pixels at once [AMHH08].

As mentioned above, scanline rasterization has the advantage that memory is written sequentially. For tile based rasterization using stamps, it is beneficial to pre-swizzle the framebuffer into 2D tiles. Tile-based storage also makes it possible to easily integrate hierarchical rasterization schemes.

On modern GPUs, the rasterizer is only used to compute coverage, the z value and the barycentric coordinates. Vertex attributes are interpolated in a separate step using the shader ALUs. This avoids the need to set up interpolation equations for all attributes in the setup stage. For example, the following HLSL code

```
float4 VS_main (float4 v : V) : SV_Target
{
    return v;
}
```

Listing 1: HLSL code interpolating a `float4` vertex attribute.

is compiled to the following assembly

```
s_mov_b32     m0, s2

; Interpolation performed in two steps
; (1-t)*a + t*b
v_interp_p1_f32  v2, v0, attr0.x
v_interp_p2_f32  v2, v1, attr0.x

v_interp_p1_f32  v3, v0, attr0.y
v_interp_p2_f32  v3, v1, attr0.y

v_interp_p1_f32  v4, v0, attr0.z
v_interp_p2_f32  v4, v1, attr0.z
```

```
13
14 v_interp_p1_f32  v0, v0, attr0.w
15 v_interp_p2_f32  v0, v1, attr0.w
16
17 ; export to framebuffer
18 v_cvt_pkrtz_f16_f32  v1, v2, v3
19 v_cvt_pkrtz_f16_f32  v0, v4, v0
20 exp mrt0, v1, v1, v0, v0 done compr vm
```

Listing 2: GCN assembly generated for Listing 1.

By deferring the attribute interpolation to the ALUs, it is also possible to bypass the interpolation completely. In this case, per-triangle constant data is only fetched once. In HLSL, the `nointerpolation` option can be used, as in the following example:

```
1 float4 VS_main (nointerpolation float4 v : V) : SV_Target
2 {
3     return v;
4 }
```

Listing 3: HLSL code using a non-interpolated, `float4` vertex attribute.

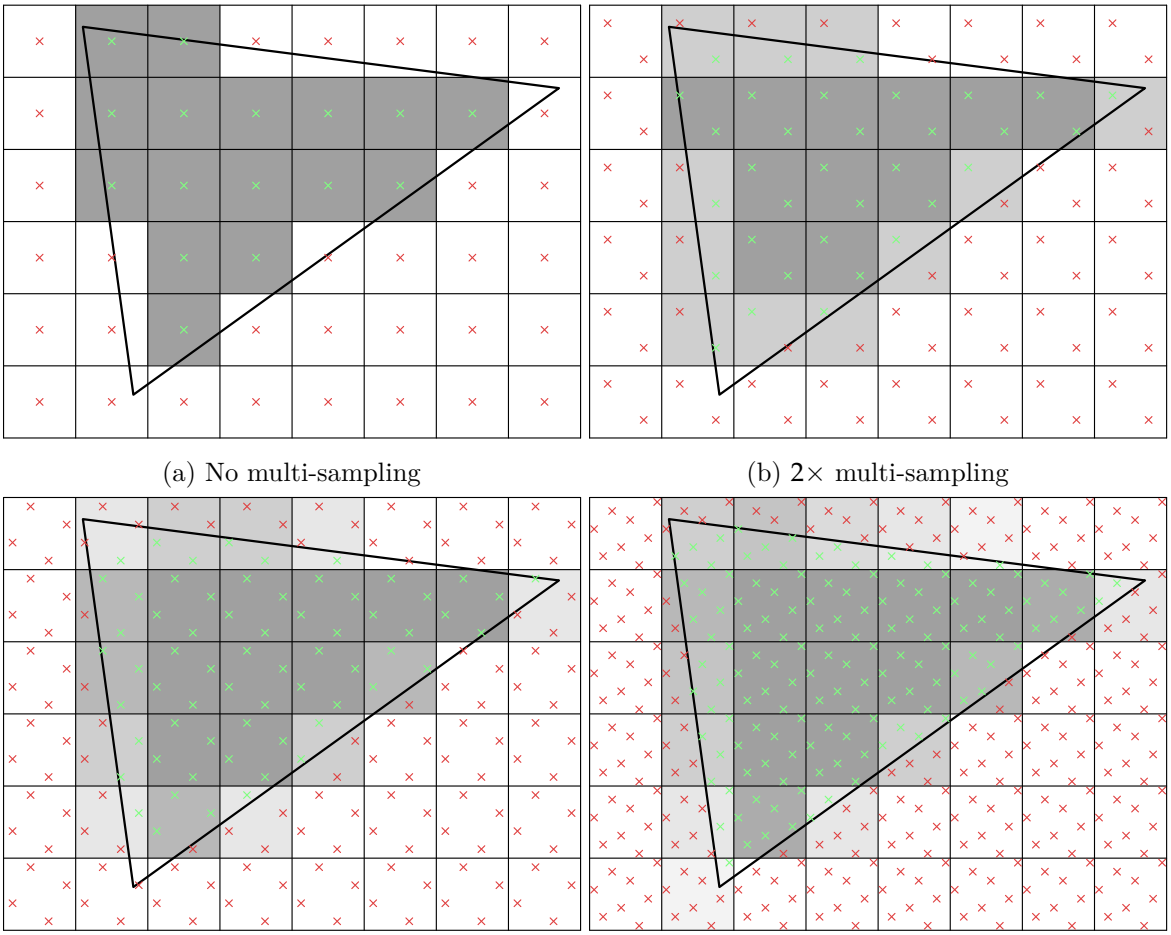
After compilation, the hardware will only load the attributes from the first vertex, avoiding any interpolation:

```
1 s_mov_b32     m0, s2
2
3 ; No interpolation, simple move
4 v_interp_mov_f32 v0, p0, attr0.x
5 v_interp_mov_f32 v1, p0, attr0.y
6 v_interp_mov_f32 v2, p0, attr0.z
7 v_interp_mov_f32 v3, p0, attr0.w
8
9 ; Export to framebuffer, omitted
```

Listing 4: GCN assembly generated for Listing 3.

Especially for very small triangles and of course for per-triangle constant data, disabling the interpolation can be beneficial. Besides the reduced computation count, on GCN hardware, it also reduces the amount of required registers.

Even with attribute interpolation turned off, small triangles with a size of a few pixels at most are still inefficient. One reason is the relatively complex edge equation setup stage, which is only beneficial if many pixels are going to be covered by a triangle. Another problem are derivatives: In order to provide those, modern GPUs render and shade pixel quads [FBH+10]. For pixel-sized triangles, this results in severe "overdraw", further reducing performance.



(a) No multi-sampling

(b) 2× multi-sampling

(c) 4× multi-sampling

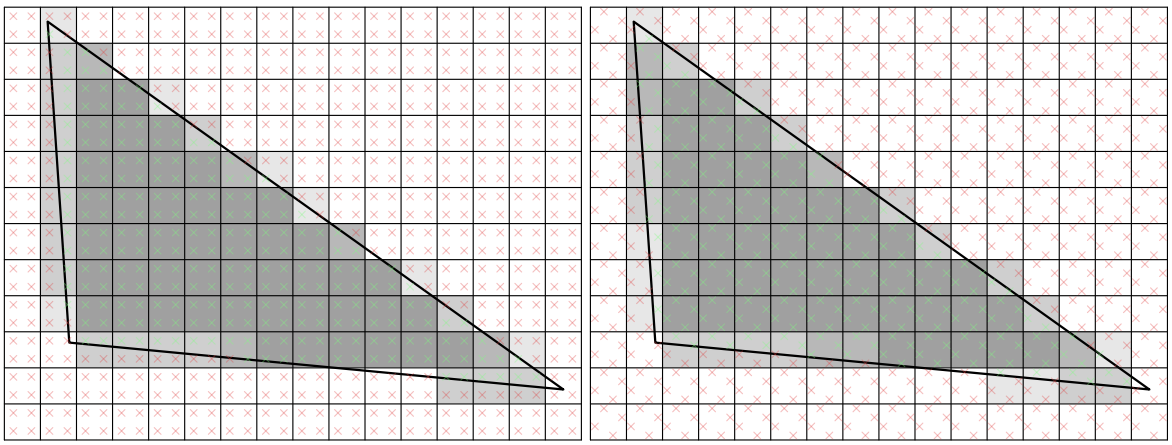
(d) 8× multi-sampling

Figure 7: Quality comparison between different multi-sampling levels for a single triangle.

2.4 ANTI-ALIASING

The GPU rasterizer can optionally perform anti-aliasing. As mentioned above, this is a simple addition to the edge equation based rasterization pipeline. For maximum efficiency, and to enable the continued use of integer units, the sample locations are restricted onto a sub-pixel grid with a fixed resolution. The multi-sampling patterns used by Direct3D 11 can be seen in figure 7. They have been specified on a 4-bit sub-pixel grid.

A major difference to common anti-aliasing algorithms is how the shading is evaluated on GPUs. Typically, anti-aliasing processes everything in higher resolution and down-samples the results using a filter. On GPUs, evaluating the fragment shaders 8× per pixel for anti-aliasing was considered too expensive, and thus a more efficient solu-



(a) 4× super sampling (b) 4× multi-sampling

Figure 8: Comparison between super-sampling on a regular grid and the corresponding multi-sampling patter. At the same sample count, multi-sampling is able to capture more gradient levels. This is particularly visible on the left and the lower edge.

tion is provided. It is based on the observation that shader aliasing is typically much less severe than geometry aliasing. In practice, all textures are already pre-filtered through the use of mip-maps to avoid aliasing. This can be exploited by decoupling the fragment shader execution frequency from the coverage evaluation.

This kind of anti-aliasing is known as *multi-sample* anti-aliasing [AMHH08]. Once enabled, coverage is computed using the full sub-pixel grid and for every sample. The actual shading is only executed once per primitive per pixel. That is, if two primitives overlap a single pixel, and each covers two samples, the fragment shader will be invoked only twice – once per primitive, and not four times. If the whole pixel is covered by a single primitive, the fragment shader will be invoked only once. This reduces the shading overhead per pixel drastically, making multi-sample anti-aliasing comparatively cheap.

The multi-sampling patterns allow for higher quality compared to super-sample anti-aliasing on a regular grid. All multi-sampling patterns are placed on a rotated respective irregular grid. This drastically improves edge quality compared to a high-resolution regular grid, as can be seen in figure 8.

Current GPUs support up to 8× multi-sampling. One difficulty which arises from MSAA based anti-aliasing is that it is not compatible with deferred rendering. In a deferred renderer, the actual shading is performed on a per-sample buffer, and the original information which samples have originated from the same primitive is lost. This information has to be reconstructed during the deferred shading pass in order to minimize the number of actually shaded samples.

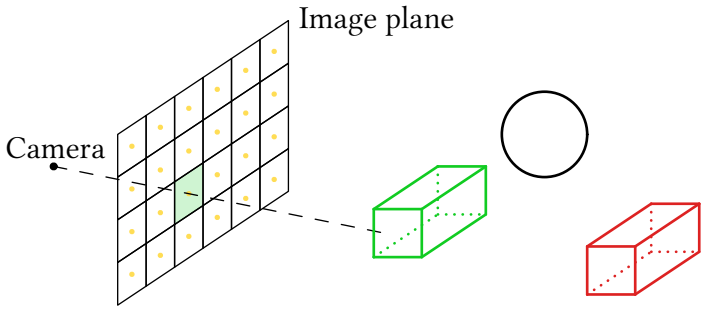


Figure 9: The basic principle of raytracing. Rays are traced from the camera through the pixel grid into the scene, and the closest intersection is determined.

2.5 RAYTRACING

Raytracing is an image generation technique which relies on tracing rays through a 3D scene and computing their intersections with scene objects [App68]. It can be used to simulate a wide range of optical effects.

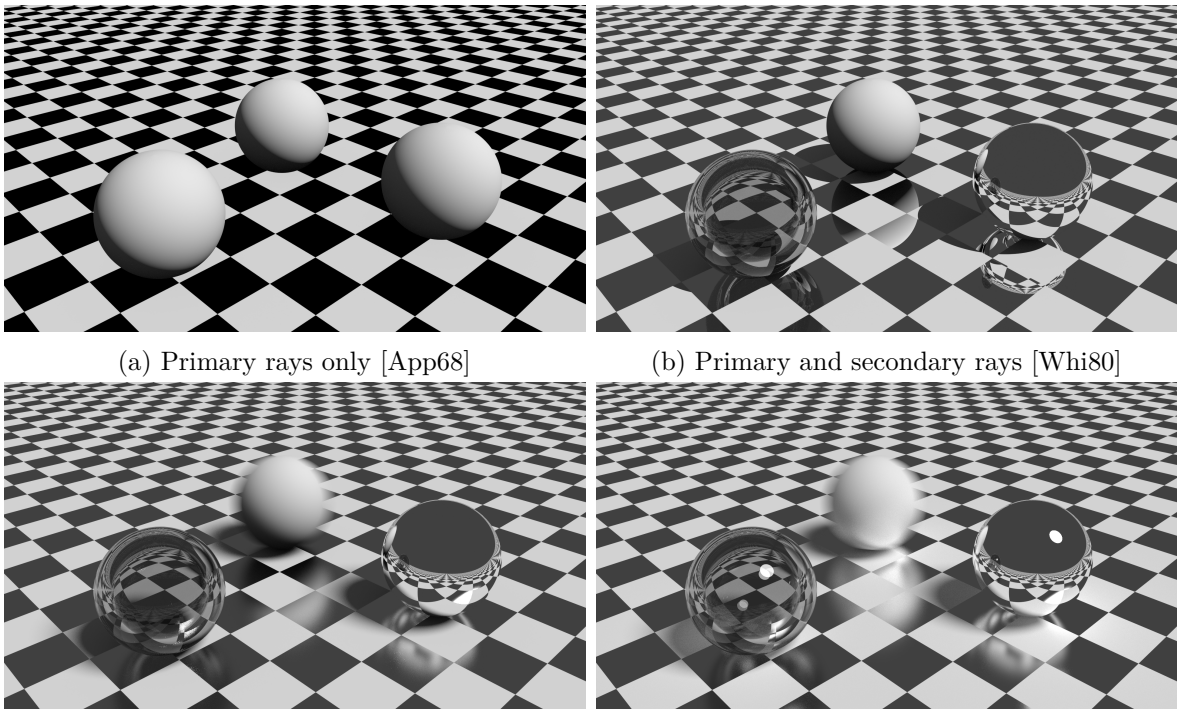
A basic raytracer can be seen in Figure 9. It starts by generating one ray per pixel of the image plane. The ray is then cast into the scene, all intersections with the scene objects are determined and the closest intersection is returned. The initial view rays are also known as *primary* rays. Such a basic raytracer is very limited and can only be used to visualize direct lighting, without any shadows or reflections (see Figure 10a.)

For secondary effects like shadows, reflections and refractions the raytracer must be extended to allow recursive ray generation, that is, rays can be spawned not only for the camera, but at arbitrary points of the scene [Whi80]. These rays are known as *secondary* rays. For instance, to simulate shadows, a shadow ray is spawned at the hit point towards the light source. If it is occluded, the point is in shadow and can be shaded accordingly. With this extension, it is possible to simulate perfect mirrors, refractive objects as well as shadows (see Figure 10b). The main limitation of this method is that it cannot capture effects which require integration over multiple rays, for example, glossy reflections or motion blur.

This is solved by the introduction of *distributed* raytracing, which traces multiple rays to integrate results. With distributed raytracing, it is now possible to capture motion blur, soft shadows from area lights and glossy reflections (see Figure 10c). The major disadvantage of using raytracing for such effects is that many more rays are required.

The final evolution of raytracing is *path tracing*. A path tracer solves the rendering equation directly by sampling individual light paths [Kaj86]. It can capture diffuse

interreflections, caustics, and various other effects (see also Figure 10d). Compared to distributed raytracing, path tracing requires a dramatic increase in ray counts to achieve noise-free results.



(a) Primary rays only [App68]

(b) Primary and secondary rays [Whi80]

(c) Distributed ray tracing [CPC84]

(d) Path tracing [Kaj86]

Figure 10: Raytracing can cover a wide variety of effects. Secondary rays allow for perfectly specular reflections, refractions and shadows. Distributed raytracing enables effects such as glossy reflections, motion blur and depth of field. Path tracing provides a way to directly solve the rendering equation, covering diffuse and specular inter-reflections.

The key to fast, high-quality raytracing thus requires fast solutions for two separate problems: Ray/scene intersections and sampling. Ray/scene intersections is a key problem, as billions of rays have to be traced and intersected with potentially hundreds of millions of primitives. Sampling is crucial to avoid wasting time on rays with low contribution to the final scene. For example, ray density should be guided by the BSDF to avoid tracing rays where the BSDF is zero.

These are also the two areas where the majority of the research has been focused. On the sampling side, techniques like bidirectional path-tracing [LW93], multiple importance sampling[VG95], Metropolis light transport [VG97, KSKAC02] have been introduced to improve the sampling quality. There has been also considerable research in “biased” techniques, that is, rendering techniques which do not converge exactly

against the correct solution but approximate the result of the rendering equation instead. Popular examples are irradiance caching [WRC88] and photon mapping [Jen96].

The raytracing itself, that is, the act of sending rays into the scene and intersecting them with the primitives, can be also sped up in a variety of ways. There are three main areas which can be optimized: Ray traversal, ray/primitive intersections and ray/scene intersections.

Ray traversal can be optimized for instances where multiple rays have to be traced from a spatially close location or in a similar direction. In this case, it may be more efficient to group the rays into ray packets and use a packet traversal algorithm which shares the computations between all rays in a packet [BEL+07].

Ray/primitive intersections can be similarly optimized. For example, it is possible to pre-compute various terms for a ray/triangle intersection in order to reduce the actual intersection cost [Woo04].

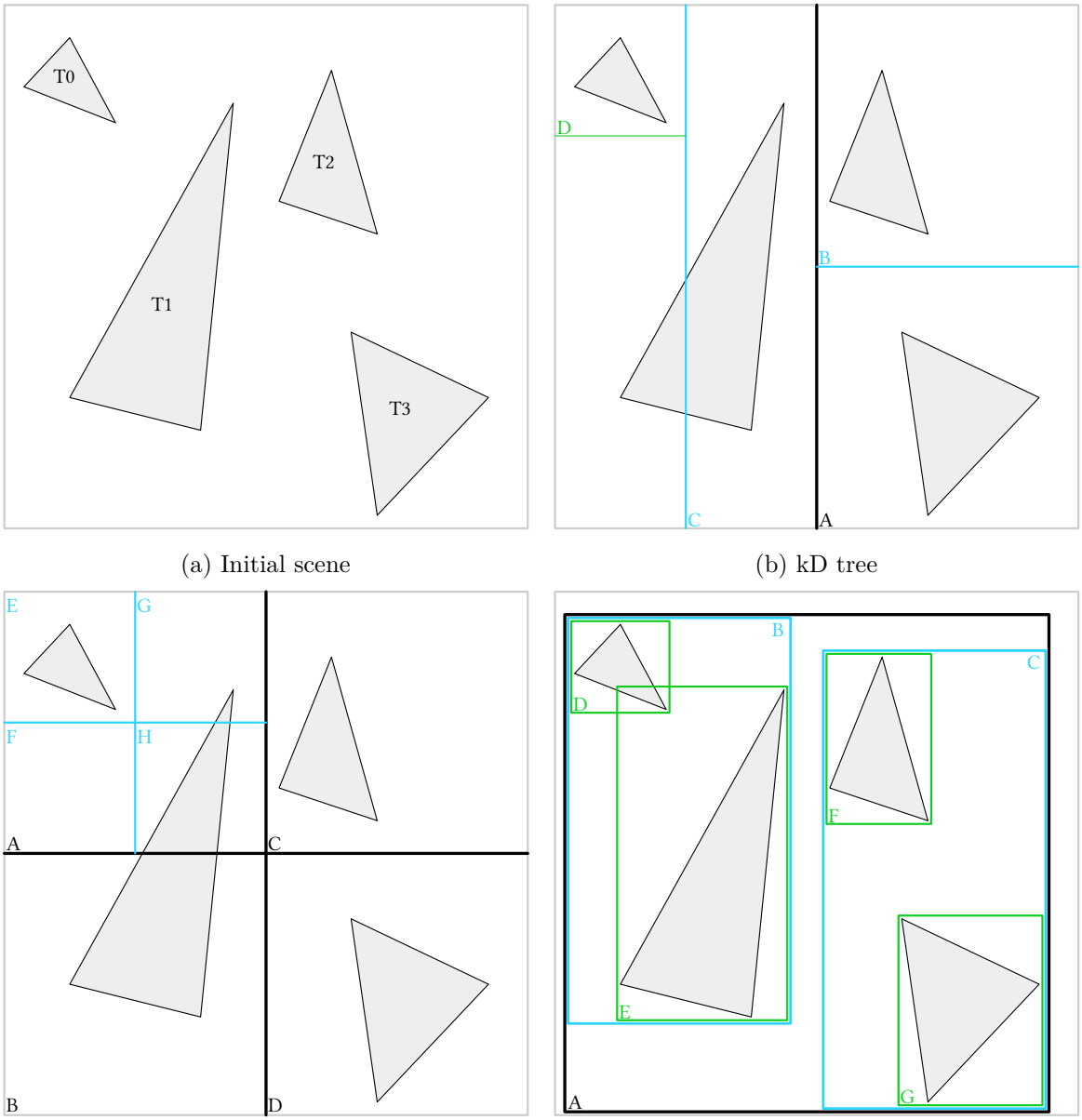
The last area are ray/scene intersections. To this end, a large variety of acceleration structures have been introduced, which minimize the number of ray/primitive tests that have to be performed. In the following sections, I will cover three popular structures which are widely used for raytracing.

2.5.1 *kD tree*

A kD tree is an extension of a binary tree for multiple dimensions [Ben75]. The kD tree can be used to accelerate range queries in multiple dimensions and has been originally designed for use in databases. Every interior node of the kD tree defines a split position and axis. Each split divides the domain into two sub-spaces at a hyperplane perpendicular to the split axis. In 1D, a kD tree is equivalent to a binary tree.

In graphics, kD trees have been successfully used to accelerate ray-tracing. As mentioned above, the domain is split during the build. This also includes primitives; if for instance a triangle straddles the split plane, it is split into multiple triangles (see also Figure 12). This guarantees that leaf nodes never overlap. Depending on the depth of the tree and the split locations, this can result in significant amounts of duplicated references, when primitives are split multiple times.

The kD traversal algorithm (see also Algorithm 2.1), used to trace a ray through a kD tree, is conceptually very simple [FS05]. For each interior node, the ray is intersected with the split plane. If a hit occurs, the far child is pushed onto the stack and traversal continues into the near child. Otherwise, traversal continues solely in the near child.



(a) Initial scene

(b) kD tree

(c) Quadtree

(d) BVH

Figure 11: Acceleration structures subdivide the scene and create a hierarchical structure to accelerate ray queries. In this example, the scene in 11a is subdivided with three different algorithms until a leaf size of 1. Black indicates the first subdivision level, blue the second and green the third level.

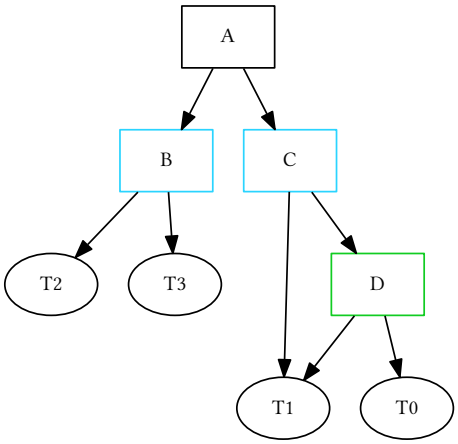


Figure 12: The kD tree corresponding to Figure 11b. Notice that the triangle `T1` is referenced multiple times. There is also an empty node in the tree, as `C` contains only one child.

```
1  function Traverse(r, t)
2      result = miss
3      Push(stack, r_min, root, r_max)
4      loop
5          r_min, r_max, next = Pop(stack)
6          if IsInterior(next) then
7              node = t[next]
8              close, far = SortChildren(r, node)
9              if Intersect(r, node, t) then
10                 Push(stack, t, r_max, far)
11                 next = close
12                 r_max = t
13             else
14                 next = close
15         else                                  ▷ Else, leaf or empty node
16             if IsLeaf(next) then
17                 node = t[next]
18                 if IntersectLeaf(node, hit, t) then
19                     if t ≤ r_max then
20                         return result
21             if Empty(stack) then
22                 break
23     return result
```

Algorithm 2.1: Standard kD tree traversal

Once a leaf is hit, the ray is intersected with a all primitives. The traversal guarantees that children will be visited in strict front-to-back order.

Restart traversal

Maintaining the stack can require a significant amount of memory, especially if many rays have to be traced in parallel. The kD tree traversal can be reformulated to avoid the need for a stack completely. Instead of a stack, the ray is shortened at each step to the current ray/plane intersection depth. Once a leaf node is reached, the ray is restarted from the root. This will still result in a correct, front-to-back traversal at the expense of additional interior intersections. Notice that the restarting is only possible as there is no overlap of objects between leaf nodes.

Short stack

It is also possible to combine restart and stack-based traversal for kD trees. In this case, a *short stack* is used which only stores the last few nodes [HSHH07]. Once the short stack is exhausted, the algorithm falls back to a standard restart traversal. This reduces the amount of restarts significantly compared to a completely stack-less traversal, while maintaining good performance. The key insight is that restarts due to leaf misses are the most expensive. With a short stack, the traversal can visit multiple leaves before it has to start again at the root.

2.5.2 *Octree*

An octree recursively subdivides the domain into N equally sized sub-domains along the three major axes (see also Figure 11c). Typically, the subdivision count is 2, splitting the domain into 2^3 sub-regions (hence the name *Oct*ree.) The equivalent in 2D is called a *Quadtree*. Originally, it has been used to allow efficient geometry modeling [Mea82]. Subsequently, it has been also adopted as a spatial acceleration structure [Sam89, Sam90]. Traversal is similar to a kD tree, with the main difference that three separate planes have to be processed at each interior node.

In Figure 13, the tree for the scene in Figure 11c can be seen. Similar to the kD tree, an octree may contain empty nodes and duplicated references. It is also limited in its adaptivity to the scene. Unlike the kD tree or the bounding volume hierarchy, which allow the split planes to be placed freely inside a node, the octree always has to split at the center. This can lead to imbalanced trees, especially when geometry is misaligned relative to the octree grid.

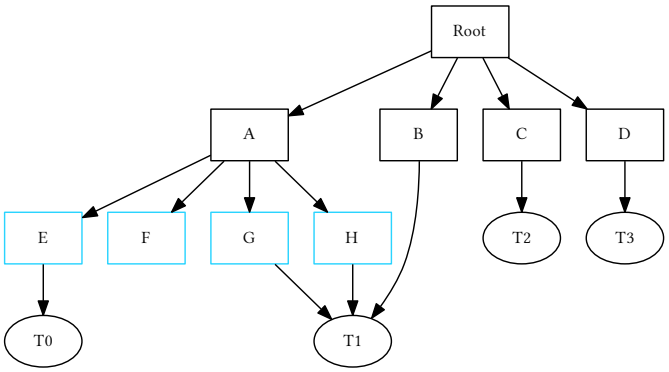


Figure 13: The octree tree corresponding to 11c. Notice that the triangle `T1` is referenced three times, at multiple tree levels. There is also an empty node in the tree, `F`.

Octrees have become the data structure of choice for voxel rendering [CNLE09]. For voxel data, the regular structure of the octree allows the octree grid and the voxel grid to be coincident. Together with the fact that voxels can be trivially simplified by merging, this allows an octree to be used for both traversal acceleration, storage and the level-of-detail selection.

At preprocessing time, this requires to generate voxels for all levels of the tree. During the traversal, a voxel octree raytracer keeps track of the distance to camera. Once it reaches a node, it computes the correct level-of-detail and only descends the tree down to this level. With this approach, it is possible to render highly complex scene with low memory usage.

A possible traversal routine can be found in Algorithm 2.2. It uses a stack to store the current node if multiple children have to be visited. Storing the current node instead of all intersected children on the stack reduces the storage requirements. The traversal can be further optimized by exploiting the power-of-two sizes and the regularity of an octree [RUL00, LK11a].

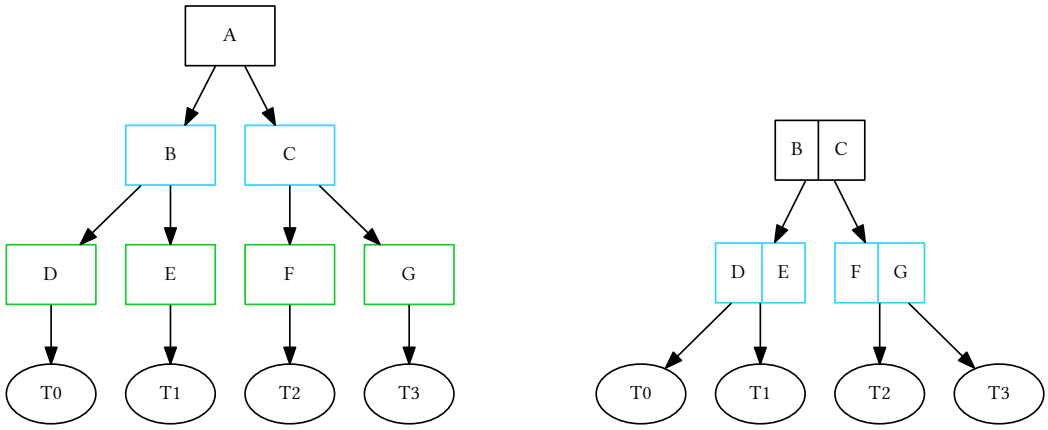
Similar to the kD tree, an octree can be also traversed using a stack-less traversal kernel or using a short-stack. This is possible due to the fact that there are no overlaps in an octree.

2.5.3 *BVH*

A bounding volume hierarchy is a tree which stores a bounding volume at each node (see also Figure 11d) [RW80]. Child nodes are completely contained within their par-

```
1  function TRAVERSE(r, t)                    ▷ r has been already intersected with scene bounds
2      next = root
3      o = (0,0,0)                                                  ▷ Offset into the current node
4      l = t_depth                                                  ▷ Current traversal level
5      PUSH(stack, root, r_min, r_max)
6      loop
7          if ISINTERIOR(next) then
8              node = t[next]
9              s = 2^(l−1)
10             c_max = r_max
11             result = INTERSECT(r, node, t)
12             if MULTIPLEHITS(result) then
13                 PUSH(stack, node, c_max, l)
14             octant = CLOSESTCHILD(r, node)
15             next = GETCHILD(node, octant)
16             l = l − 1
17             UPDATEOFFSET(o, octant, l)
18         else                                                     ▷ Else, leaf or empty node
19             if ISLEAF(next) then
20                 node = t[next]
21                 if INTERSECTLEAF(node, hit) then
22                     return hit
23             if EMPTY(stack) then
24                 return miss
25             else
26                 r_min = r_max
27                 next, r_max, l = POP(stack)
28                 CLEARBITS(o, l)
```

Algorithm 2.2: Octree traversal



(a) Standard BVH

(b) Two wide BVH

Figure 14: Bounding volume hierarchies corresponding to Figure 11d. On the right side, two levels of the tree have been folded to generate a BVH2.

ent. The scene is thus not subdivided by split planes like for the octree or kD tree, but successively refined hierarchy of bounding volumes. Similar to the kD tree, the bounding volume hierarchy depends on a good build heuristic to minimize costs.

Unlike for a kD tree or octree, a restart traversal without external storage is not possible for a bounding volume hierarchy. Consider the case depicted in Figure 11d. If a ray is traced from the top which intersects both nodes `D` and `E`, it is not possible to shorten the ray such that a restart will correctly visit only `E` without potentially missing intersections. It is thus necessary to use additional external storage during traversal.

By using a stack, we can formulate the bounding volume hierarchy traversal algorithm (see also Algorithm 2.3). Starting from the root, the first node is intersected. If the node is not hit, the next node is popped from the stack; if the stack is empty, traversal terminates. Otherwise, if the node is an interior node, the closer node is marked as the next node for traversal and the node further away is pushed onto the stack, and traversal continues immediately at the next node. If the node is a leaf node, the ray is intersected with all primitives. On hit, the ray is shortened and the stack is popped if not empty, otherwise, traversal terminates.

It is also possible to implement the algorithm by always popping from the stack and pushing two nodes during the interior handling; using a *next* node pointer is a trivial optimization which saves unnecessary stack traffic.

An optimization which can be easily performed with bounding volume hierarchies is merging levels to obtain wider, but more shallow trees [DHK08]. This can be easily performed as a post-process by merging nodes from adjacent levels. In Figure 14b, the original tree (see Figure 14a) has been flattened by one level. As multiple bounding boxes are stored in a single node, it is now possible to process a node using vector instructions and achieve higher performance, at the expense of additional bounding box tests compared to a non-flattened tree.

One additional benefit is that the bounding boxes of the leaf nodes move up to the interior nodes. Previously, all nodes contained a bounding box and some auxiliary data. After the flattening, only interior nodes contain bounds, while leaf nodes merely consist of a pointer to the primitives. This separation makes it possible to pack the data more tightly and to transform general gather instructions into efficient vector load operations.

```
1  function Traverse(r,t)
2      next = root
3      result = miss
4      loop
5          if Intersect(r, node, t) then
6              if IsInterior(next) then
7                  node = t[next]
8                  close, far = SortChildren(r, node)
9                  Push(stack, far)
10                 next = close
11
12                 continue
13             else if IsLeaf(next) then
14                 node = t[next]
15                 if IntersectLeaf(node, hit, t) then
16                     if t ≤ r_max then r_max = t; result = hit
17         if Empty(stack) then
18             break
19         else
20             next = Pop(stack)
21     return result
```

Algorithm 2.3: Standard BVH traversal

2.5.4 *Tree building heuristics*

For kD trees and bounding volume hierarchies, the decision where to place the split planes respectively which primitives should be placed into a node has a significant impact on performance. A standard approach to minimize the expected traversal cost is the *surface area heuristic* [MB90]. It can be used for both kD trees and bounding volume hierarchies without modification. It estimates the cost of a tree as:

$$\frac{C_i * \sum_{i=1}^{N_i} A(i) + C_l * \sum_{l=1}^{N_l} A(l) + C_o * \sum_{i=1}^{N_l} A(l) * N(l)}{A(\text{root})} \tag{7}$$

In the equation above, N_i, N_l denote the number of interior and leaf nodes, $N(l)$ the number of objects in leaf l, and $A(i), A(l)$ the surface area of the interior node i or the leaf l. C_i, C_l and C_o are constants which describe the costs of the various operations. C_i is the cost to traverse an interior node, C_l the traversal cost for a leaf and C_o the cost to intersect an object.

The cost function tries to balance the number of nodes against the subdivision depth. Very deep trees will minimize the last term at the expense of high node counts, while a shallow tree will minimize the first two terms at the expense of a high amount of ray/object intersections.

The surface area heuristic is evaluated while building the tree. At every subdivision step, two cost terms are computed:

$$C_s(p) = C_i + C_o * \left(\frac{A(\text{left})}{A(\text{node})} N(\text{left}) + \frac{A(\text{right})}{A(\text{node})} N(\text{right}) \right) \tag{8}$$

$$C_t = C_o * N(L) \tag{9}$$

C_s describes the costs which would happen if the current set of objects is split into two groups, left and right at a split position p; while C_t is the cost of creating a new leaf node. Computing C_s is very costly, as the best split has to be found. For scenes with hundreds of millions of objects, this can quickly become prohibitive, as each object boundary is a potential split location. A common optimization is to use a binned estimator, which only considers a fixed set of split candidates [WH06]. While this may not find the best possible split location, in general, binned builders reach similar quality as precise builders while making the build much faster.

2.6 PARALLEL PROGRAMMING ON GPUS

Graphics has been accelerated very early using dedicated hardware [FPE+89]. This is easy to explain by looking at the typical graphics workloads: Vertex transformation, projection, rasterization and shading. All of these tasks share the property that a tiny *kernel* is executed for many independent elements. Accordingly, GPUs have been optimized for throughput processing of simple code [FH08]. This led to highly parallel architectures, with a very high number of execution units compared to CPUs. As the hardware was and is widely available, there has been a lot of interest to run non-graphics algorithms on the GPU, which is called GPGPU programming [OLG+07] – general purpose GPU programming. At the beginning, this required that the algorithms are expressed in terms of graphics API calls. For instance, to perform a dot product, textures containing the input had to be created, and a fragment shader would be executed to produce an output image containing the result.

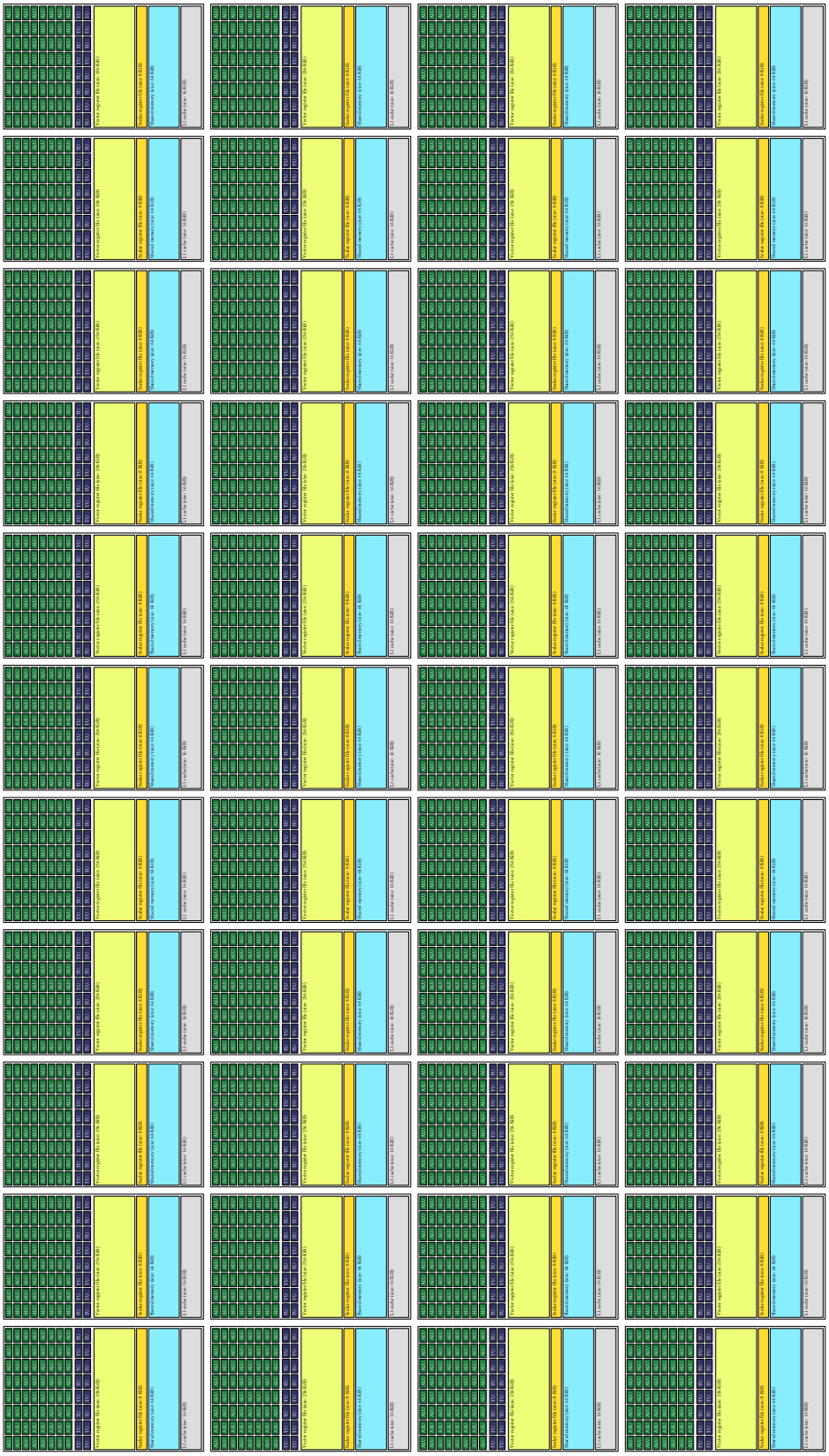
Very quickly, low-level APIs like Brook [BFH+04], Lib SH [MQP02] and CTM/-CAL [Hen07] appeared which allowed to bypass the graphics APIs in order to facilitate compute-only code. The low-level APIs provided higher-level abstractions and compiled down to shader code and draw calls. As such, they shared the limitations with the graphics APIs. A notable exception in this context is CTM, which was set below the graphics API layer and allowed to program the shader ALUs directly using the hardware ISA. Unfortunately, even CTM did not expand the programmability significantly as the hardware was not designed for compute tasks.

Over time, both the hardware and the associated APIs evolved into full-fledges compute programming frameworks, for example, OpenCL [Khr12], DirectCompute, OpenGL compute shaders [SSKLK13] and NVIDIA CUDA [NBGS08]. Hardware changes included unified shaders, random memory access and communication between threads. Until late 2005, GPUs had dedicated shader ALUs for vertex and fragment processing with different capabilities. For instance, the vertex ALUs were not connected to the texture units and thus it was not possible to sample textures. This changed with the introduction of unified shaders in the XBox 360 Xenos GPU, which, for the first time, had a unified shader ALU array capable of doing both vertex and fragment processing [AMHH08]. This was a crucial step for GPU programming, as now all ALUs could be used uniformly for processing.

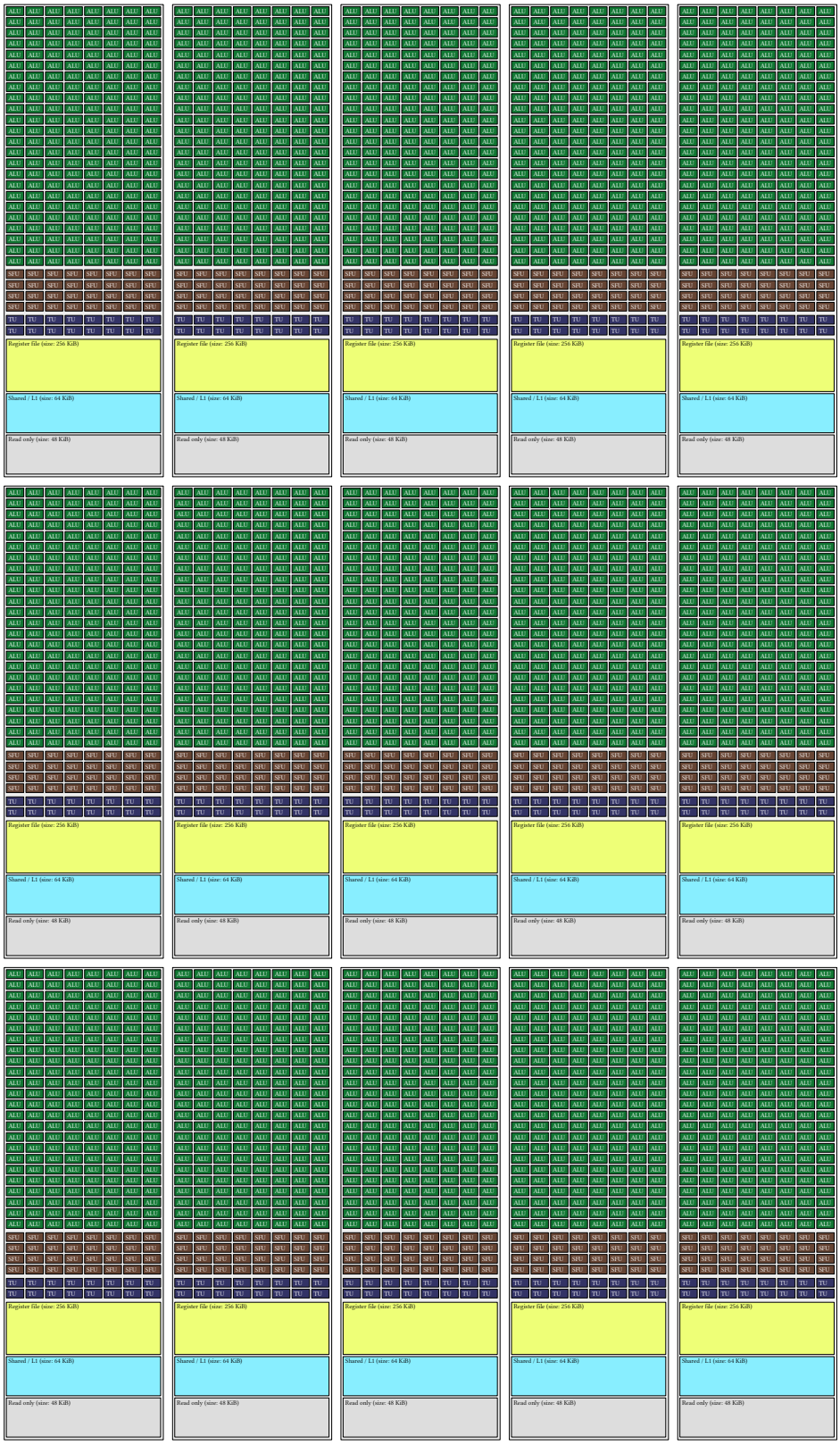
On the software side, the programming abstractions became increasingly high-level. NVIDIA CUDA is NVIDIA's in-house GPU computing language which uses an extended C++ dialect and targets only NVIDIA GPUs. The extensions to C++ include graphics specific built-in instructions to sample textures as well as inter-thread com-

munication. OpenCL is a more general programming framework; targeting parallel architectures from CPUs over FPGAs and DSPs to GPUs, using a C dialect called OpenCL C. It also provides graphics-related functions to sample images, but has less low-level, device-specific tuning options than CUDA. DirectCompute and OpenGL compute shaders are high-level, graphics oriented APIs which aim for direct integration into rendering pipelines.

In the following, I will introduce the OpenCL terminology to describe the hardware units. In OpenCL, a GPU or a CPU is called a *compute device*. Each compute device consists of one or more *compute units* and *global memory*. For a CPU, a compute unit is a CPU core; on GPUs, these corresponds to AMD's CUs or NVIDIA's SM. Each compute unit is comprised of *processing elements*, which are the ALUs that execute the computation. Additionally, a compute unit also has *local memory*, which can be only accessed from the processing elements inside the compute unit.



(a) AMD R9 290X



(b) NVIDIA GTX TITAN

Figure 15: GPU architecture of two modern GPUs. The AMD GPU has 44 compute units with 2816 ALUs, the NVIDIA GPU has 15 compute units with 2880 ALUs.

Using the terminology above, the AMD GCN architecture found on the R9 290X is a GPU containing 44 compute units with 64 processing elements each (see Figure 15). Each compute unit has 64 KiB of local memory attached, called local data store (LDS). Additionally, a 16 KiB sized L1 cache is attached to each compute unit, which, just like the shared 1 MiB sized L2 cache is not directly programmable from OpenCL.

Similarly, an NVIDIA GeForce GTX TITAN using the Kepler architecture contains 15 compute units with 192 processing elements each. Each compute unit has a 64 KiB sized, combined local memory/L1 cache. It can be reconfigured to assign 16, 32 or 48 KiB to L1 or local memory. Additionally, a 48 KiB sized read-only cache is attached to each compute unit. Finally, the chip also contains a 1.5 MiB large L2 cache. Similar to the AMD GCN architecture, the L1, the read-only cache and the L2 cache is not exposed to OpenCL and hence cannot be programmed directly.

Current GPUs also include additional, fixed-function hardware units. For example, for texture access, dedicated texture units are used instead of issuing read instructions and performing the filtering in the ALUs.

Due to their unique characteristics, GPUs require a highly parallel programming model. In the following sections, I will explain the OpenCL execution and memory model, which is an example of such a parallel programming model.

2.6.1 *Execution model*

GPUs are wide vector machines which require a different programming model for efficient usage. On CPUs, a scalar execution model is prevalent. In the scalar execution model, each instruction is processed in-order by a single execution thread. To execute an operation on multiple items, a loop must be used. Parallelism is thus not explicitly modeled, but has to be recovered from loops.

In contrast, in the parallel execution model, parallelism is explicitly modeled. Instead of a single thread, groups of threads which execute in parallel are the basic building block. In OpenCL, this is modeled by splitting the *domain* into *work group*s. A work group is a set of *work items*, each work item representing one data element (see also Figure 16). The execution model is also relaxed. The order in which the work items are processed is undefined; and synchronization is only allowed within a single work group, but not across the whole domain. The parallelism comes from the execution order, which allows an implementation to execute all work items in parallel or sequentially. Instead of a loop which processes multiple elements as in the sequential model, the individual loop iterations would be modeled as separate work items to enable parallel execution. The relaxed computation model is in fact very similar to the model used

Work domain Work group

Figure 16: Work definition in OpenCL. On the left, the complete work domain (`NDRange`) is visible. It consists of multiple, independent work groups, as seen on the right. Each work group is comprised of multiple work items.

for graphics. There a single, small kernel is executed independently in non-observable order for all vertices of a mesh or for all fragments generated by a draw call.

However, unlike in the graphics model, multiple work items inside a single work group can be synchronized in OpenCL using barriers. These force all work items to advance until a certain point in the program before execution continues. By using a barrier, and local memory visible to the complete work group, it is possible to exchange information between work items. This is a major difference to the computation model in graphics, where all work items execute independently.

By only allowing communication inside a single work group and removing ordering guarantees, it is possible to map the work load efficiently onto a wide range of parallel architectures. In general, one work group is assigned to a single compute unit, and the work items assigned to the processing elements. On an AMD GCN GPU for instance, 64 work items are queued onto the 64-wide vector unit. One difficulty that arises from this implementation is the handling of conditional instructions, or in general, *divergence* across a vector unit.

On a scalar architecture, control flow is implemented using conditional branches as can be seen in Listing 5. If a jump is taken, instructions are skipped and execution continues elsewhere.

```
1 void conditional (float* a, float* b, int c)
2 {
3     if (c == 0) {
4         *a = 1337;
5     } else {
6         *b = 42;
```

```
    }
}
```

Listing 5: C code containing a simple conditional instruction.

When compiled for a scalar x86 CPU, it is easy to see that the condition is implemented using a conditional jump instruction (see Listing 6). This will skip the memory write completely and requires no transformations on the input code.

```
    ; if (c == 0)
    test    edx, edx
    ; if true, execute jump
    je      .LBB0_2
    mov     dword ptr [rdi], 1109917696 ; = 42.0f
    ret
.LBB0_2:
    mov     dword ptr [rsi], 1151803392 ; = 1337.0f
    ret
```

Listing 6: Generated code for the C conditional example, compiled using Clang 3.4.

On a vector architecture, this approach does not work. The problem is that the instructions operate on multiple elements at the same time. For each lane, the conditional may have a different result. The equivalent code for the scalar example, rewritten for using OpenCL C, can be found in Listing 7. The key difference is the `get_global_id (0)` function call, which returns a *different* value for each SIMD lane. It is no longer possible to execute a conditional jump, as this would skip the first branch for all lanes. What we need now is a way to execute the first and second branch for a subset of the SIMD lanes.

```
__kernel void conditional (
    __global float* a,
    __global float* b,
    __global int* c)
{
    const int idx = get_global_id (0);

    if (c [idx] == 0) {
        a [idx] = 1337;
    } else {
        b [idx] = 42;
    }
}
```

Listing 7: OpenCL code containing a simple conditional instruction.

This is typically solved by *masking*, or function predication. Using an execution mask, some of the vector units can be "turned off" and their results ignored. Comparisons no longer return a single scalar, but instead a mask for all vector lanes. The code is then transformed as following: After the mask is computed, the code for the first branch is executed. Next, the mask is inverted and the second branch is executed, and finally, the mask is cleared again.

On the AMD GCN architecture, this separation is directly visible through the separate *scalar* and *vector* instructions. The vector instructions are executed on the 64-wide vector unit. Each vector instruction inspects the current *execution mask* which contains one bit for every lane. If not set, the vector lane is effectively "off" during a vector instruction.

Scalar instructions are executed on the scalar ALU. It is responsible to set up conditional execution by keeping track of the active vector lanes. In the following code, instructions prefixed with `v` are executed on the vector ALUs, instructions prefixed with `s` are executed on the scalar ALU:

```
1     ; load parameters
2     s_buffer_load_dword  s0, s[4:7], 0x10
3     s_buffer_load_dword  s1, s[4:7], 0x60
4     s_buffer_load_dword  s4, s[8:11], 0x20
5     s_waitcnt     lgkmcnt(0)
6     ; compute get_global_id
7     s_min_u32     s0, s0, 0x0000ffff
8     s_mul_i32     s0, s12, s0
9     s_add_u32     s0, s0, s1
10    v_add_u32     v0, vcc, s0, v0
11    v_lshlrev_b32  v0, 2, v0
12    s_load_dwordx4  s[12:15], s[2:3], 0x1c0
13    v_add_u32     v1, vcc, s4, v0
14    s_waitcnt     lgkmcnt(0)
15    ; load c into v1
16    tbuffer_load_format_x  v1, v1, s[12:15],
17        0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
18    s_buffer_load_dword  s0, s[8:11], 0x00
19    s_buffer_load_dword  s1, s[8:11], 0x10
20    s_waitcnt     vmcnt(0)
21    ; if (c [idx] == 0), per lane
22    v_cmp_eq_i32  vcc, 0, v1
23    ; save execution mask
24    s_and_saveexec_b64  s[4:5], vcc
25    s_waitcnt     lgkmcnt(0)
```

```
26      v_add_u32      v0, vcc, s0, v0
27      ; if mask is 0, jump to else branch
28      ; otherwise, continue
29      s_cbranch_execz  label_0024
30      s_load_dwordx4  s[8:11], s[2:3], 0x180
31      v_mov_b32      v1, 0x44a72000 ; = 1337.0f
32      s_waitcnt      lgkmcnt(0)
33      ; a [idx] = 1337
34      tbuffer_store_format_x  v1, v0, s[8:11],
35          0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
36  label_0024:
37      ; invert the current execution mask
38      s_andn2_b64    exec, s[4:5], exec
39      v_add_u32      v0, vcc, s1, v0
40      ; if execution mask is 0, branch, otherwise continue
41      s_cbranch_execz  label_002E
42      s_load_dwordx4  s[0:3], s[2:3], 0x1a0
43      v_mov_b32      v1, 0x42280000 ; = 42.0f
44      s_waitcnt      lgkmcnt(0)
45      ; b [idx] = 42
46      tbuffer_store_format_x  v1, v0, s[0:3],
47          0 offen format:[BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]
48  label_002E:
49      s_endpgm
50  end;
```

Listing 8: Generated assembly for the OpenCL condition code.

In the generated assembly (see Listing 8), in line 22, the conditional is set up by computing the execution mask. The program then continues to compute the first branch in line 30, and, for the remaining units, the second branch in line 42.

With masking, it is possible to transform any structured control flow into vector instructions. Unfortunately, the execution efficiency is now strongly tied to the number of active vector units, called the execution *coherency*. For highly incoherent code, a GPU may have most of its vector units masked off all the time and only execute at a fraction of the possible theoretical throughput.

2.6.2 *Memory model*

For highly parallel architectures as GPUs, it is not possible to provide fast memory access to each processing unit. An AMD R9 290X can perform one FMA operation per cycle per ALU. Assuming a clock frequency of 1 GHZ and that all three input

Memory space	Visibility	Location	Access	Size	Speed
Private	Work item	On-chip	R/W	Small	Very fast
Local	Work item & group	On-chip	R/W	32-64 KiB	fast
Constant	Work item	On/Off-chip	R	64 KiB	slow-very fast
Global	Work item & group	Off-chip	R/W	Large	slow

Table 1: Memory spaces available in OpenCL. Private memory size is in the order of few KiB; global memory is typically several GiB.

operands have to be fetched from off-chip memory, over 30 TiB/s of read bandwidth would be necessary to saturate the ALUs. This is roughly 100× more than is available, even though the GPU already uses extremely fast memory on a wide bus. In order to reach the peak throughput, the data must be stored in on-chip memory, in this case, the register file, which is fast enough to saturate the ALUs. Only those provide enough bandwidth. The next closest memory, which is the local memory local to each compute unit, only provides an aggregated bandwidth of nearly 5.5 TiB/s, followed by the level 1 cache with 2.8 TiB/s and level 2 cache with 1 TiB/s.

This makes it necessary to give the developers explicit control over where data is stored. In general, off-chip communication must be limited as much as possible. As mentioned in the execution model, the programming framework exposes *work groups* which can be synchronized together. Additionally, all items in a work-group can also access local memory. By using the local memory as a manually managed cache, it is often possible to achieve substantial performance improvements, due to the extremely high bandwidth of local memory relative to global memory.

In OpenCL, the memory hierarchy is exposed as `private`, `local`, `constant` and `global` memory (see also Table 1). Private memory are the registers allocated for one work item. These are typically stored in the register file. Local memory is per compute unit; on GCN, it is placed in the local data store. Constant memory is a very small memory space designed for read-only access. As it is constant, it can be easily cached on-chip. Finally, global memory is the off-chip memory, generally multiple orders of magnitude larger than the on-chip memory.

Another difference between the memory on GPUs and traditional architectures is they way latency is handled. As mentioned above, the vector unit is rather simple and features no out-of-order processing. Specifically, it cannot speculatively load data in order to reduce stalls. On GPUs, this problem is magnified as the memory controllers are optimized for throughput. The best access patterns on GPUs are large, continuous reads which often occur when geometry or texture data is accessed and hence the

hardware has been optimized for this scenario. Accordingly, small, random reads and writes are not optimized and take comparatively long. This makes it necessary to hide the latency somehow as otherwise the GPU compute units will starve while accessing memory. The solution used on GPUs are very large register files and context switching.

They key idea is to keep multiple execution contexts on-chip and switch between them to continue execution while memory transactions are processed. That is, instead of running just enough work items to fill the vector unit, the scheduler will assign as much work as can possibly fit onto a compute unit. Once a load or store instruction is encountered, the execution switches to other work items and continues until another load or store is encountered, and so forth. The number of work items that can be scheduled onto a compute unit is limited by the size of the register file and the used local memory, as the state of all concurrently executing work items must be stored.

For instance, on GCN, the vector register file has 256 KiB of memory; for 1024 work items, this allows for 64 registers to be used per work item. The ratio of work items that are scheduled onto a compute unit relative to the maximum number of work items is called *occupancy*. For GCN, 100% occupancy can only be reached if each work item uses 25 or less registers, exactly 40 work groups are scheduled onto a compute unit and each work group consists of 64 work items which use at most 1.6 KiB of local memory.

The ability to hide latency does not only depend on the number of work-items in flight. Even with 100% occupancy, the GPU will stall if every instruction is a memory access, as there is not enough available parallelism. Thus, besides the number of work items, the ratio of ALU to memory instructions is also crucial. For example, assuming that a load has a latency of 400 cycles, at an ALU:Memory ratio of 20, only 5 work groups are needed to perfectly hide latency [AMD13]. 5 work groups correspond to an occupancy of 12.5%.

In order to obtain the best possible performance, it is thus necessary to optimize algorithms in three areas. First, coherence during the execution should be maximized to ensure that all ALUs are used and the maximum number of computations is performed per cycle. Second, the occupancy should be maximized by using low amounts of registers per work item. In general, this requires simple compute kernels and careful ordering of instructions to ensure short lifetimes for variables. Third, memory access must be optimized. This includes placing data in the right memory space as well as the access pattern, as the off-chip memory is optimized for very large reads.

3

PREPROCESSING

In this section, I will cover the pre-process. This is the stage where the input – either meshes or volumes – is converted to voxels and the level-of-detail simplification is generated. I will first cover the voxelization itself, which is dependent on the input type. After that, I will explain the simplification, which works directly on the voxels and is thus input-agnostic. Finally, I will present performance and memory usage results for a variety of scenarios.

3.1 VOXELIZATION

The voxelization converts iso-surfaces and triangle meshes into a *surface voxel* representation. For meshes, an adaptive rasterizer is used, which can also ensure different topological constraints on the output. Iso-surfaces are converted directly by identifying surface voxels in the volume.

3.1.1 *Adaptive rasterization*

The adaptive rasterizer expects that the input consists of triangles placed inside the unit cube. For arbitrary geometry, the first step is thus to compute the scene bounding box and determine the transformation to the unit cube. All triangles are then transformed before being passed on to the rasterizer. In the following, all coordinates are assumed to be in the $[0, 1]$ range.

The voxelization is now computed as follows: Starting at the root, the node is subdivided in turn for each major axis. On each subdivision, the input triangle is split into up to three triangles at the split plane. After three successive splits, the parts of the triangle are rescaled back to the unit cube, assigned to each child and moved into the child's local coordinate frame. The algorithm now continues recursively at the child nodes. In effect, the rasterizer adapts to the underlying geometry by creating an octree during the voxelization. Each step of three successive subdivisions creates one new octree level.

Rescaling to $[0, 1]$ during the splitting has the advantage that the code and numeric constants remain fixed, without accumulating floating-point errors. At each split, all

tests are done against the same floating point constants, which is more precise than successive subdivision of the domain and additions to get the individual split planes.

The algorithm is tuned for uniformly tessellated and very dense input meshes. In particular, low resolution meshes with very large triangles or highly oversampled voxelizations will spend a lot of time in the expansion phase. In general, the best performance is achieved when the voxel to triangle ratio is below or close to 1. For the cases where low-resolution geometry is resampled, it may be beneficial to pre-tessellate the geometry to improve efficiency during voxelization. Alternatively, a separate, large triangle rasterizer could be included, similar to [Pan11] or [RB13].

Once a leaf has been reached, the exact intersection area of the triangle with the voxel can be immediately computed, as the triangle has been already clipped to the voxel. At leaf nodes, I accumulate per-vertex attributes like normal direction and color, weighted by the area of the triangle. For color, I also store the total triangle area per voxel to be able to compute the correct average color. For normals, a simple re-normalization is enough, which also reduces the amount of data per leaf voxel.

For each leaf, I also determine the active faces of the voxel. This is a pure rendering optimization to reduce the amount of generated faces later on. I consider a face with normal n_f to be active if the geometry has a normal n_g which points into the same direction. The actual test is implemented as $\langle n_g, n_f \rangle > \epsilon$. An ϵ is needed to robustly handled cases where a triangle is coincident or parallel with one of the voxel faces.

After all triangles have been processed, the tree is traversed and the leaf voxels are gathered into larger chunks, typically containing 256^3 voxels. Each voxel is also stored relative to the chunk origin. Using chunks of ≤ 256 voxels allows me to store the voxel position using 8 bits per component. Alternatively, the voxelizer can also output world-space voxels with 16 bit per component, which is used by the voxel raytracer described in Chapter 5.

The processing has been optimized in three areas: Tree memory storage, multithreading and optimized geometry input.

The first optimization helps to reduce the amount of memory required by the tree. As it is an octree, it contains a high number of leaf nodes, so a compact storage of leafs is paramount to reduce the memory usage. To this end, I have separated the interior nodes from the leaf nodes during the tree building. Instead of using the union of a leaf and interior node during the build phase, I use two separate data structures. The interior nodes in my framework contain only leaf pointers, and the leaf nodes solely consist of the payload. This enables a very tight, memory efficient packing of nodes, as the interior nodes are perfectly aligned with the cache line size of current x86 CPUs – 64 bytes for 8 pointers of 8 bytes each. This makes the access of interior nodes very

Resolution	Packed	Allocated	Unpacked
1024^3	53	64	131
2048^3	212	232	530
4096^3	855	936	2130
8192^3	3434	3648	8546
16384^3	13761	14617	34224

Table 2: Memory usage for the St. Matthew scene at different resolutions. `Packed` is the minimal memory required by separating interior from leaf nodes and packing without any alignment; `Allocated` is the actually used memory including the block allocator overhead and alignment, and `Unpacked` is the memory required by unified leaf and interior nodes. All numbers are in MiB.

fast, which is crucial during tree building, as each triangle hast to traverse the tree several times until it reaches its leaf node.

Separating is possible as the leaf nodes all exist at the same level. During the traversal, I keep a context variable which allows me to determine the current level, and from that, I can deduce whether a pointer points a leaf or interior node. This removes the need to store a bit per node.

The leaf nodes are specialized for each vertex format using C++ template programming. This avoids the allocation of a "worst-case" leaf with all attributes required by any possible voxelization input. The template based system is also flexible enough that it is easily possible to add for example support for textures as the color or normal source without having to change other parts of the voxelization pipeline. In my design, vertex attribute fetching, interpolation and evaluation is fully orthogonal to the rasterization algorithm itself.

Memory fragmentation is also minimized by using a custom memory allocator. General purpose memory allocators are optimized to work with objects of any size. During tree building, I allocate only leaf or interior nodes, which are all very small and have fixed size. This is a use-case which is not optimal for the system provided memory management routines. Instead of allocating the nodes one-by-one, I use a block allocator which reserves memory in large chunks. Individual node allocations are served directly from a chunk, which is a very cheap operation as it only requires to increment a pointer. This optimization drastically reduces memory fragmentation and also improves locality, as leaf nodes which are created at the same time are also likely to be stored nearby in memory.

One problem that arises from the use of a block allocator is that individual nodes cannot be released. Fortunately, during the build, there are only allocations but never deallocations of nodes. The only situation where nodes are deleted is during the tree merging at the end. I have solved this by using a separate allocator for each tree. After a tree has been merged, the corresponding allocator is destroyed and all memory required by the tree is released.

In total, the separation of leaf and interior nodes plus the tight packing reduces memory usage by a factor of approximately 2.2 for the St. Matthew scene (see also Table 2). Compared to the minimal required memory, the block allocator introduces between 6%-10% overhead. For the very small scenes, the rounding to 8 MiB blocks is responsible for the majority of the overhead. For larger sizes, the overhead compared to the "packed" layout is related to alignment. In the tested configuration, the leaf nodes are 13 bytes in size and aligned to 4 bytes, thus consuming 16 bytes.

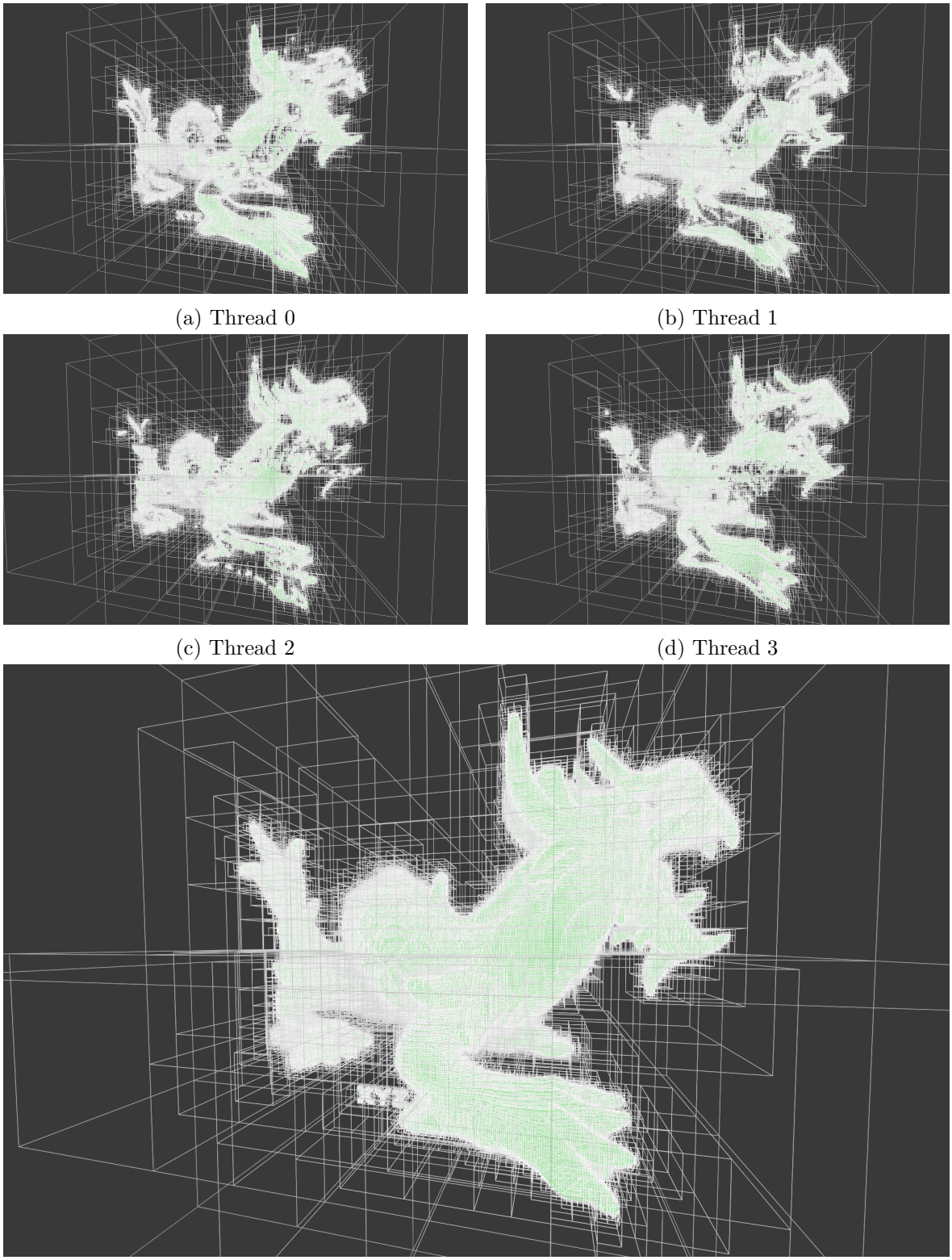
The second important optimization is to take advantage of multi-threading. In my work, I distribute the work across multiple CPU cores using a two-stage approach. In the first stage, CPUs take chunks of the input geometry and rasterize it using the technique outline above. This results in a per-CPU octree. In the second stage, the trees from all CPUs are gathered and merged into a single tree. This tree is then traversed once to emit all surface voxels.

The splitting is performed using work-stealing on the input data. Once a CPU thread is ready, it locks the input file, reads the next geometry chunk, releases the lock and continues voxelization. As the input is generally unsorted, random parts of the mesh are assigned to each CPU (see also Figure 17).

During the merge phase, the first tree is taken as the merge target and all other trees are merged into it sequentially. Merging of two trees is very cheap and requires the joint traversal of both the source and target tree. If the source tree encounters a node which is not yet present in the target, it is immediately copied the target tree and traversal continues.

At the leaf level, an attribute specific merge function is performed if the node is present in both the source and target tree. This is again implemented using template functions, which allow the optimal merge function to be used for every tree configuration.

It is possible to perform the tree merging using a parallel reduction. For instance, if 8 threads have been used during rasterization, the reduction can be performed in three steps by merging the odd and even trees in each step. A downside of such a merging scheme is that the required amount of temporary memory is linear in the number of



(a) Thread 0

(b) Thread 1

(c) Thread 2

(d) Thread 3

(e) Merged result

Figure 17: The rasterization load is distributed over multiple threads. Each thread processes a different part of the model; at the end, the results are merged into a single tree. White indicates the bounding boxes of interior nodes, green leaf nodes.

Model	# Triangles	PLY†	PLY† (LZ4)	PLY‡	OBJ‡	GS†	GS† (High)
Lucy	28055742	509	421	1086	1190	428	408
Thai	10000000	182	146	378	404	143	134
Dragon	7219045	131	103	270	286	94	91
Atlas	507512682	10183	7451	-	-	6091	4970

Table 3: On-disk data sizes for various models. GS is a plain geometry stream with "fast" LZ4 compression, GS (High) is compressed using LZ4HC. † indicates a binary format, ‡ a text based format. PLY† (LZ4) is a binary PLY file compressed using "fast" LZ4.

trees. In contrast, the sequential merge requires only temporary memory equal to the size of the currently merged source tree.

A possible optimization to reduce memory performance and slightly improve rasterization speed is to perform a rough pre-sort of the input geometry. This can be done by running the voxelization for a few levels. In this case, the work-stealing algorithm must be also adjusted so it takes the spatial location of a geometry chunk into account. The rasterization will benefit in two ways from this optimization: First of all, the trees will cover smaller volumes, which reduces the required memory. Second, as the trees become smaller, the rasterization becomes faster due to improved locality.

Eventually, this could be the basis for a complete out-of-core algorithm. In the first stage, a pre-sort with a coarse voxelization is performed as described above. However, instead of running the rasterization right away, the output for each block is flushed to disk. In the second stage one rasterizer is started per coarse block and all outputs are merged. The main difference to existing out-of-core algorithms like [BLD13] is that the voxelizer itself is used for all steps, without the need for a separate sorting step.

A key part of the fast processing pipeline is efficient geometry storage. As mentioned above, the multi-threaded rasterizer uses work-stealing to read new geometry data. This makes it necessary to provide the input data in a chunked format, that is, as blocks of independent triangles which can be quickly read from disk.

Unfortunately, common file formats like Wavefront OBJ and PLY are not suited for this usage scenario. While very popular, OBJ does not provide a binary encoding, which makes it necessary to perform string parsing while the file is read. This significantly reduces reading performance for large files, in addition to the very high disk usage requirements (see also Table 3).

PLY provides a binary encoding, which allows for efficient reading. While it is possible to implement streaming into PLY, compression can be only performed on the whole file, and not on parts of it.

To solve these problems, I introduce *geometry streams* as the on-disk storage. Geometry streams are files which are comprised of independent geometry chunks stored sequentially in the file. Each geometry chunk consists of per-chunk metadata, the vertex data and optionally index data. Inside the chunk, all data is stored as a binary blob, similar to binary encoded PLY files. Each chunk is completely self-contained and can be skipped over after the metadata has been read.

To minimize disk space, the data contents of a chunk are compressed on-disk using LZ4 compression (see also Table 3). LZ4 is a compression algorithm designed for fast compression/decompression performance, achieving decompression rates of over 1 GiB/sec on a single thread. In general, compressing chunks does not reduce the compression rate compared to merging all vertex data first and compressing.

For each chunk, the metadata describes the format of the chunk and provides information about the chunk including its bounding box. This allows the rasterizer to quickly determine the size of the work domain without having to load all triangles into memory first.

On the reader side, the file format is completely abstracted away by *mesh views*. These are interface classes which provide uniform access on any geometry format supported by my framework. Using mesh views, it is possible to choose the optimal encoding for every chunk independently without changes to the rasterizer itself. For instance, it is possible to preprocess a mesh and decide per-chunk whether an indexed mesh should be used or not, depending on the data size and vertex reuse.

3.1.2 *Topology*

By construction, the rasterizer creates all voxels which are intersected by a triangle. This results in a conservative voxelization. As described in 2.1.1, a conservative voxelization is also a superset of both 6- and 26-separating voxelizations. This makes it possible to prune the output voxel set by introducing additional tests (see also Figure 18).

To this end, I have implemented the intersection targets from [Lai13] as a post-process. If 6 or 26 separability is requested, each voxel that is touched by a triangle is additionally tested against the appropriate intersection targets. If no intersection is found, the leaf node is still created, but marked as empty. During voxel emission, empty leafs are simply skipped. For the 6-separating test, I project the triangle onto the xy, yz, xz planes and test if the mid-point of the unit square in that plane is contained within the projection. For the 26-separating test, the triangle is intersected directly in 3D with the four diagonals of the unit cube.

(a) Conservative

(b) 26-separating

(c) 6-separating

Figure 18: The adaptive rasterizer can output conservative, 26-separating or 6-separating voxelizations. 26- and 6-separating voxelizations are created by pruning of the conservative output. As can be see, all connectivity configurations allow for gap/crack free rendering.

Figure 19: A voxel iso-surface extracted from the Richtmeyr-Meshkov instability at a resolution of $2048 \times 2048 \times 1920$ voxels. The extracted surface consists of 128880391 voxels.

3.1.3 *Iso-surface extraction*

Iso-surfaces of volumes can be also visualized using my framework. The only difference between volume and triangle data is the import phase, which requires a slightly different conversion step. The simplification and rendering remains the same.

I assume that the input is provided as a block-based volume with an at least 1-wide block border. Blocks which are completely empty, that is, where all samples are below or above the iso-value, are ignored. For the remaining blocks, I compute a conservative voxelization by inspecting the N_6 neighborhood of every "inside" cell, that is, the cells which have a value above the iso-value. If at least one neighbor exists with a value below the iso-value, the voxel is emitted. [Liu77, HL79].

For high-quality rendering, it is necessary to compute a normal for each voxel. In the 3D scalar field, this can be performed directly by computing the gradient:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f(p)}{\delta x} \\ \frac{\partial f(p)}{\delta y} \\ \frac{\partial f(p)}{\delta z} \end{bmatrix} \tag{10}$$

In a discrete volume, the derivatives are typically computed using central differences:

$$\frac{\partial f(x,y,z)}{\partial x} = \frac{f(x+h,y,z) - f(x-h,y,z)}{h} \tag{11}$$

These gradients are sufficient for very smooth data sets, but exhibit problem for near-binary volumes, for example, volumes generated from a wavelet rasterizer [MS11]. During the iso-surface extraction, I use a normal estimation based on linear regression [NCKG00]. Similar to central differences, the linear regression estimation requires a one-wide border around each volume block.

3.2 SIMPLIFICATION

The surface voxel representation allows for very efficient simplification. All voxels are stored in a grid, and for each simplification, the grid resolution is halved. For each "parent" voxel, up to eight "child" voxels have to be merged. I use a merge strategy which always creates a "parent" voxel if at least one child is visible. For the parent voxel, the face mask is the union of all children. This is equivalent to a max operation on a density volume. With this strategy, I can guarantee that the boundary faces of a parent will form the convex hull of the child voxels. I will use this property during rendering to enable efficient occlusion culling and streaming (see also Section 4.5).

During the simplification, per-voxel attributes have to be combined. Again, there are multiple possible approaches here. If per-face data is present, the attributes can be averaged *per face*. In general, this provides the highest quality but also requires the most memory. Alternatively, data can be averaged *per voxel*, which can lead to problems if for example widely varying normals are combined. The decision can be also performed per-voxel by using an additional bit to mark voxels with per-face attributes, and only store the per-face attributes for those.

For my work, I have opted to use the second approach, which is to use per voxel averages, as errors due to incorrectly merged normals typically occur after many simplification steps and do not affect the rendering in normal viewing conditions. For extremely minified views, averaged normals, albeit incorrect, have the advantage that they reduce the amount of specular aliasing, especially if no re-normalization is performed.

The simplification itself allows for efficient serial and parallel implementation. The serial version has minimal memory requirements and is highly suited for streaming implementations. The parallel implementation is designed for GPU based applications.

3.2.1 *Serial*

The serial simplification consists of two steps:

1. Sort
2. Compact

In the sorting step, the voxels are sorted such that all child voxels corresponding to a single parent are placed next to each together. This is easily done by taking the position of the voxels, masking out the least-significant bit, and sorting by this new position. This can be easily accomplished by packing all but the lowest bit into a single integer and using it as the sorting key. Eventually, the buffer will contain "runs" of voxels corresponding to a single parent voxel.

In the second step, the "runs" of voxels are compacted to produce the parent voxels. Starting at the beginning of the buffer, the first voxel is examined, and all following voxels with the same position are found. Average attributes, like color and normal, are now computed for all these voxels and their active face bits are merged together using a bitwise or. Finally, a new output voxel is generated; and the algorithm continues until the complete buffer has been processed.

A minor issue of this approach is that adjacent voxels can create active interior faces which are not visible. This can be resolved with a second pass over the generated voxels; for all faces, where a neighbor is present, the active face flag should be cleared. In practice, the additional faces have minimal impact on rendering and processing performance and do not warrant an additional cleanup pass. The memory usage is independent of the active face cleanup.

3.2.2 *Parallel*

The parallel simplification is similar to the serial simplification, but it adds additional stages to the process:

1. Parallel sort
2. "Run" identification
3. Parallel prefix sum
4. Compaction

The sorting step is the same as before, but this time, a parallel radix sort on the 32-bit combined positions is used. After the parallel sort, the identification has to be adjusted. Previously, it would process the complete data set sequentially, which is not suitable for massively parallel architectures. Instead, the compaction is separated into three steps. In the first step, runs are identified by starting one thread per voxel. This requires one additional buffer, which has one entry per voxel and is initialized to 0. Each threads checks if the current voxel position is different from the following voxel. If so, the voxel is marked as a run start by storing a 1 into the separate buffer. On this separate buffer, a parallel prefix sum is computed, which now contains the start of each run. The number of entries generated by the parallel prefix sum is also equal to the output voxel count.

The compaction now continues similar to the serial version. One thread is started per run, which gathers all corresponding voxels and combines them into one output voxel.

All steps of the parallel sort can be efficiently performed even on massively parallel architectures. The additional memory requirements are two buffers; one for the transitions and one to store the parallel prefix sum result. Additionally, the output buffer must be pre-allocated to the worst-case size, that is, equal to the input size, unless a read-back is performed after the prefix-sum to allocate the output buffer.

3.3 RESULTS

All tests have been performed on a dual Intel Xeon X5650 CPU at 2.667 GHz (2× 6C/12T), using 24 threads, running Linux. The test machine has 24 GiB of memory. It is also equipped with a Samsung 840 EVO SSD disk which achieves a read transfer rate of 265 MiB/sec.

For the testing, I have used eight different datasets (see Table 4). These data sets cover a wide range of scenarios, ranging from very small, compact scenes (`Conference`) over medium sized data sets (`Lucy`, `Powerplant`) to the very large 3D scans of the Digital Michelangelo project [LPC+00] (`Atlas`, `David` [PGC11] and `St. Matthew`). The are stored as geometry streams and chunked to allow for efficient access to parts of the model. Except for `David`, which contains position and per-vertex colors, the other data sets consist of position-only data.

The voxelization is stored in a block-based file. The block size was set to 64^3. Notice that each block only stores the surface voxels and as such, the actual number of voxels stored per block will be varying. The maximum block size is only used as an upper bound. The block size has been chosen to allow for efficient rendering. For

Model	# Vertices	# Indices	# Faces	Vertex	Index	# Chunks
Atlas	265803606	1522538046	507512682	3042	5808	7748
Conference	189221	993537	331179	4	4	41
David	483943533	2820580659	940193553	18461	10760	14358
Lucy	18870809	84167226	28055742	432	321	429
Powerplant	10966358	38277738	12759246	251	146	236
San Miguel	5448810	23641536	7880512	104	90	13589
St. Matthew	195331421	1118302335	372767445	2235	4266	5695
Thai	6106658	30000000	10000000	140	114	153

Table 4: Statistics for the tested data sets. Vertex is the size of all vertices in MiB; index the size of all indices in MiB. Chunks is the number of input chunks; the very high count for San Miguel stems from many isolated objects in the original input which have been placed into separate chunks.

preprocessing performance, a larger block size may be beneficial to allow higher disk I/O throughput.

3.3.1 *Resampling*

For all test scenes, the voxelization at the highest resolution produces output scenes with several hundred million voxels up to nearly one billion voxels for the `Conference` (see also Table 5). Outputs with very high voxel counts are `Conference` and `San Miguel`, which contain few, very large polygons. At the highest resolution, the resampling rate is close to one for all data sets except `David`, which is still slightly undersampled (see also Figure 21).

The file size for the scene is directly proportional to the voxel count (see also Table 6). For the `Atlas` data set, which has been resampled at close to one voxel per triangle at 16384^3, the voxel file size is approximately 61% of the input data size.

The preprocessing performance is highly dependent on the scene configuration (see also Figure 20). As the voxelization is parallelized over triangles, scenes with only a few triangles are limited by the voxel emission speed. This affects both `Conference` and `San Miguel`, which contain relatively few triangles covering large space extents.

For highly detailed scenes like `Atlas`, `David` and `St. Matthew`, the processing time is dominated by the per-triangle work. These data sets scale only slowly with increasing resolutions. In these scenes, each triangle is split only few times and produces eventually only very few voxels (see Figure 21 and Figure 22). If the voxelization is re-

Dataset	16384^3	8192^3	4096^3	2048^3	1024^3
Atlas	534931311	133688606	33362414	8303373	2061645
Conference	906645225	226417757	56400644	14056227	3449087
David	266667551	66585695	16620181	4149995	1036102
Lucy	395015023	98736987	24676214	6165296	1539968
Powerplant	310509751	72469820	16952067	3880932	880656
San Miguel	749592902	186504771	46403372	11527867	2870209
St. Matthew	420708194	105111535	26226157	6529552	1620059
Thai	568009917	141969441	35475799	8858855	2208164

Table 5: Number of generated voxels for the various test data sets. These are the counts from direct voxelization into the specified resolution. San Miguel and Conference produce very high amounts of voxels due to large, planar surfaces which extend through the complete input domain.

Dataset	16384^3	8192^3	4096^3	2048^3	1024^3
Atlas	5466	1347	331	82	21
Conference	13164	3320	841	206	52
David	2786	689	171	43	11
Lucy	4103	1021	256	64	16
Powerplant	4280	980	230	51	12
San Miguel	11088	2765	691	172	46
St. Matthew	4510	1115	276	68	17
Thai	6149	1546	391	101	24

Table 6: File size for the voxelized scenes in MiB.

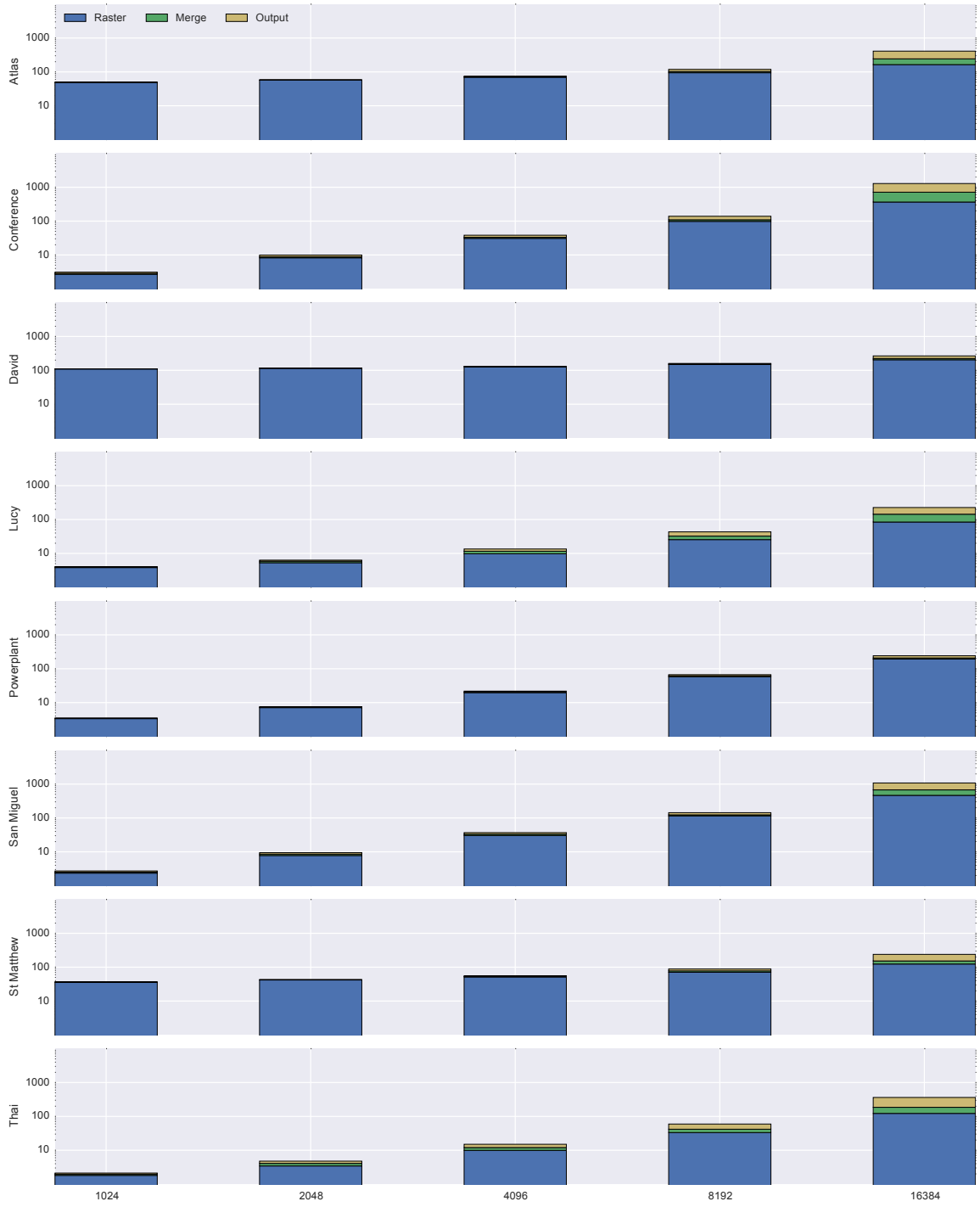


Figure 20: Performance on various test scenes and output resolutions. Complex input scenes like Atlas, David and St. Matthew are dominated by the triangle processing time and only scale slowly with increasing resolutions. Simple scenes like Conference and San Miguel are mostly limited by the voxel generation and scale directly with the output size. The output time is the time to write the voxels to disk, which is limited by disk I/O speed.

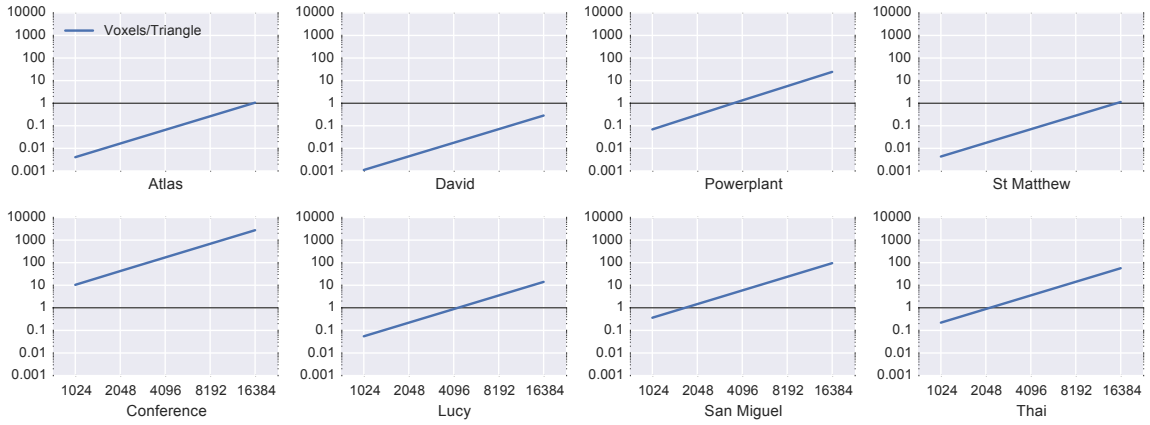


Figure 21: Effective resampling rates for the tested data sets. A sample rate of 1 indicates that the voxel data set has one voxel per triangle.

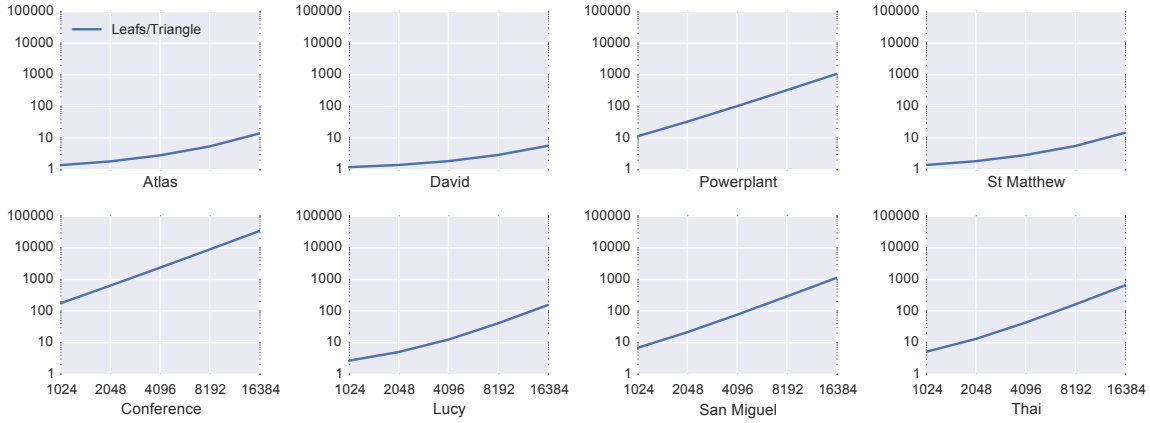
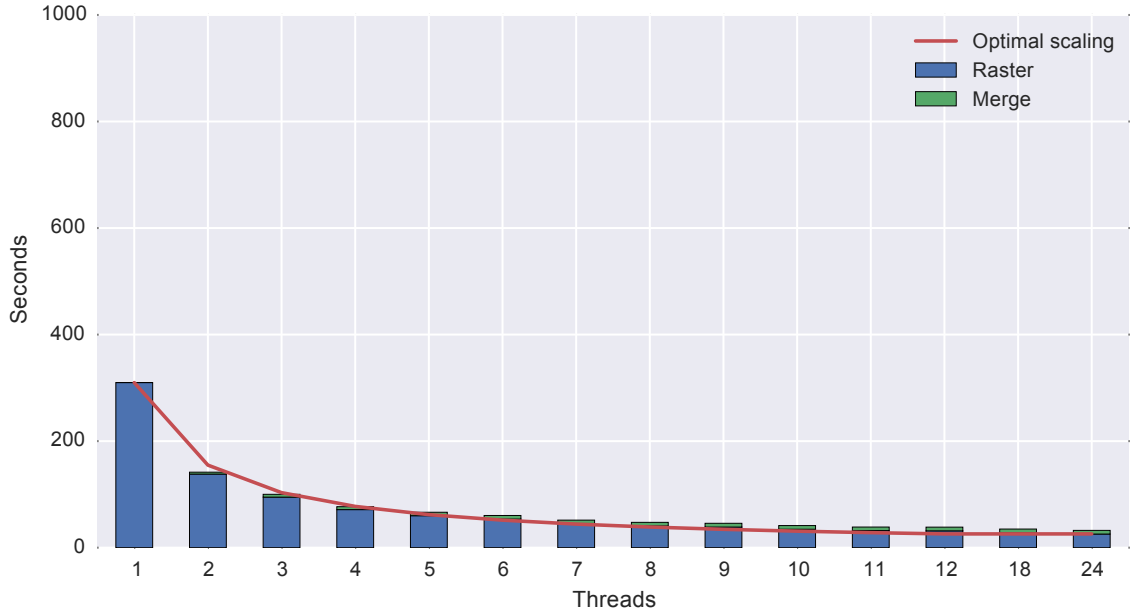


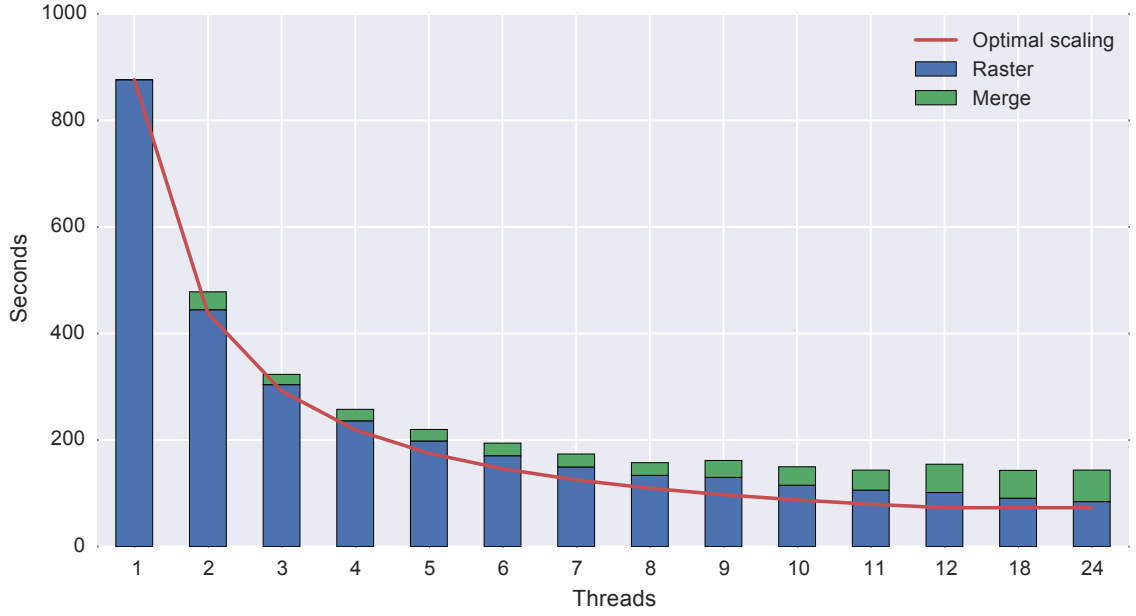
Figure 22: Per triangle expansion for the individual data sets. Leafs per triangle is the total number of generated voxels for all triangles, which contains duplicated voxels.

quired for level-of-detail purposes, a triangle/voxel ratio below 1 is sufficient, assuming uniformly sized triangles.

The adaptive rasterizer splits each triangle into individual voxels before the topological cleanup. Depending on the triangle size, this can result in very high expansion rates during the voxelization phase (see Figure 22), approaching several tens of thousands of voxels per triangle for `Conference`. The expansion rate is directly tied to the average triangle size. As long as the triangles are sub-pixel sized, the scaling is sub-quadratic with increasing resolution. Once all triangles are voxel sized or bigger, the expansion factor approaches $\mathcal{O}(n^2)$ – that is, doubling the resolution will roughly quadruple the number of voxels.



(a) 4096^3



(b) 8192^3

Figure 23: Multi-threading efficiency on the Lucy dataset for different resolutions. At 4096^3, the rasterizer reaches near perfect scaling from 1 to 12 cores. At 8192^3, the merging time starts to reduce multi-threading scalability.

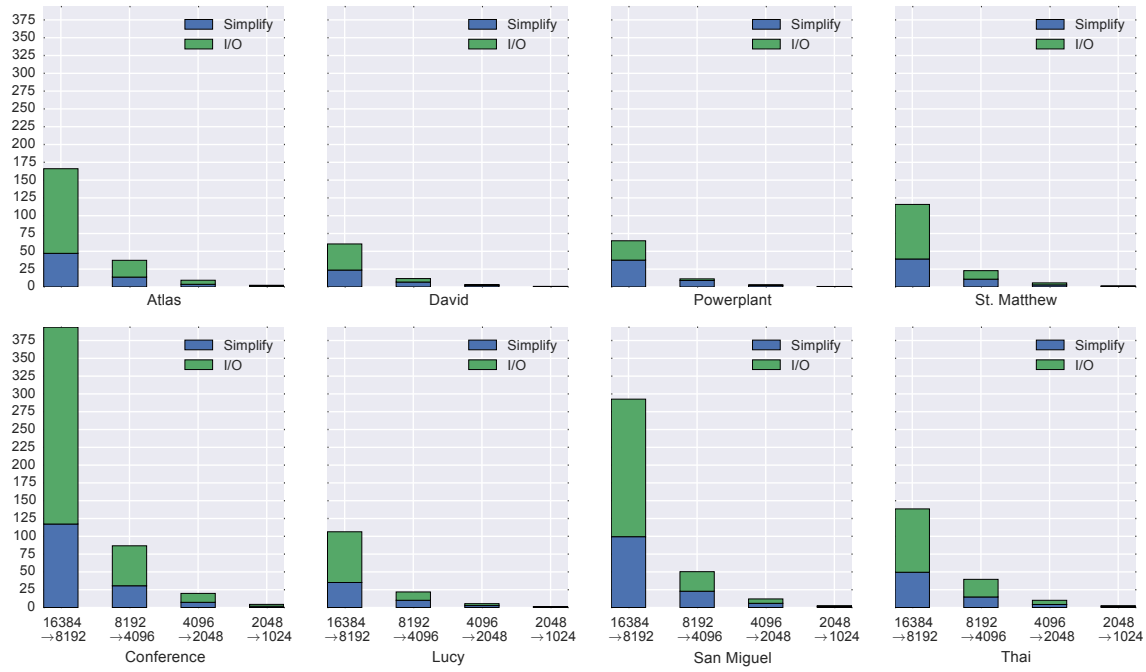


Figure 24: Simplification performance for the tested data sets. Resolutions below 1024^3 are omitted, as the simplification takes less than one second at those resolutions.

The voxelizer scales very well with increasing core counts (see also Figure 23). At low resolutions, where the time is dominated by triangle processing time, the rasterizer scales nearly linearly up to twelve cores. With hyperthreading enabled, which improves the performance in memory limited scenarios, the rasterizer is able to achieve perfect scaling. At higher resolutions, the scaling becomes worse as the merging time starts to dominate the process. As mentioned earlier, a parallel merge can be used to speed up the merging at the expense of higher memory requirements.

Iso-surface extraction is straightforward and can be quickly performed even for large data sets. I have tested the extraction performance on the Richtmeyr-Meshkov instability (see also Figure 19) which consists of a volume with $2048x2048x1920$ sample points. Extraction of 128880391 voxels requires 220 seconds and is mostly spent in I/O.

3.3.2 *Simplification*

I simplified all meshes starting from the source resolution of 16384^3 using the serial implementation. Even though it only uses a single CPU core, it is still I/O bound. Simplification performance is dependent on the sparsity of the scene. `Conference`

Dataset	16384^3	8192^3	4096^3	2048^3	1024^3	512^3	256^3
Atlas	126071	31261	7702	1883	441	110	20
Conference	196250	48047	11190	2602	540	96	32
David	63714	15783	3886	933	210	45	16
Lucy	94593	23130	5458	1215	285	68	18
Powerplant	40412	7607	1523	348	125	30	8
San Miguel	169648	34038	7026	1519	293	56	14
St. Matthew	97892	24088	5911	1423	332	92	16
Thai	133723	32140	7487	1645	355	76	22

Table 7: Block counts for the various data sets and resolutions. The simplification performance is serial, and data sets which are sparse in 3D space and contain many blocks are slower to process due to per-block overhead. The block size is 64^3; the two lowest levels of each data set contain 8 and 1 block. The block counts are for one level-of-detail simplification starting at the highest resolution.

and `San Miguel` for example consist of many blocks which contain only very few voxels, which still have to be read, processed and stored (see also Table 7). The per-block overhead could be easily mitigated by using adaptive block sizes. While the tree is written to disk, blocks could be merged until a reasonable amount of voxels is found. In the tests, the block size was fixed at 64^3, but it could be expanded to 256^3 without changes to the voxel data itself. In this case, fewer blocks would be created for sparse areas. However, cases in which this becomes necessary are likely to be severely oversampled and thus the correct solution is to simply use a lower target resolution. Accordingly, I have not optimized for this specific scenario.

The simplification is very quick and requires approximately the same time as the voxelization itself for all test scenes (see also Figure 24). If enough memory is available, the I/O time of a combined voxelization/simplification pass can be significantly reduced by keeping the voxels in memory instead of flushing them to disk and reading back.

Memory usage for the simplification is constant and requires worst-case storage for nine blocks, or a few MiB. Once a block is processed, it is immediately written to the output file and both the input and output is discarded.

4

GPU RASTERIZATION

4.1 INTRODUCTION

In the previous chapter, I described an efficient approach to voxelize and simplify triangle meshes and iso-surfaces. In this chapter, I will present the corresponding rendering algorithm. It takes advantage of the voxel representation to enable efficient, high-quality rendering. By combining both the pre-processing and the renderer, a very fast visualization pipeline can be created.

4.2 RELATED WORK

In the last years, there has been a lot of research into large octrees to render such voxel models [CNLE09, LK11a]. [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 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.

An interesting optimization for surface voxel models has been presented in [LK11a]: 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.

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 from 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 my technique, but there are several significant differences. The voxelization always produces a watertight surface and requires no blending between points. I can also integrate my rendering technique easily with other algorithms like shadow mapping, as the representation is view-independent for a given level-of-detail configuration. This guarantees that the voxel geometry matches exactly for different views.

4.3 RENDERING

The renderer is designed with the following goals:

- High quality: The generated image should have clean, smooth edges and no temporal or geometric aliasing artifacts should be visible.
- Interactive performance: The user must be able to explore the scene at interactive frame rates.
- Instant viewing: The viewer should start up immediately without having to read the complete mesh.
- Low memory usage: The renderer should use as little memory as possible.

The renderer uses the voxel representation created in the preprocessing as the level-of-detail simplification of the model. For close-up views, a hybrid rendering path is available which swaps in the source geometry where needed. High quality rendering is ensured by using the hardware rasterizer with anti-aliasing.

All of these features, including level-of-detail, occlusion culling and streaming are combined into a single, unified framework. Streaming is necessary to ensure that the mesh can be viewed instantly and that memory usage is kept minimal. As it is integrated with both occlusion culling and level-of-detail selection, the amount of streamed data can be minimized.

In this section, I will describe the run-time structure of the renderer, starting from the basic implementation. After that, the individual parts that comprise the renderer are explained in detail and a performance evaluation is presented.

4.3.1 *Overview*

The core data structure used in the renderer is an octree, similar to GPU based voxel raytracers. At load time, a single octree for the complete scene is created on the CPU and all voxel data associated with the root node is loaded.

The octree is the main data structure used to drive the rendering and managed completely on the CPU. At run-time, the following steps are executed per frame:

- Visibility determination, including culling
- Refinement and load request generation
- Loading
- Rendering

During the visibility determination, the tree is traversed to identify all visible nodes. Two different culling algorithms are used, which are described in Section 4.5.

Next, nodes which are missing yet visible or undersampled are requested for loading. To ensure high-quality rendering, it is necessary to guarantee that the maximum pixel error is below one, that is, that the projected size of a voxel never exceeds the size of a pixel. During the traversal, a conservative test is performed for each node; if it is found to be undersampled, all children are requested for loading.

During the loading stage, data is fetched from disk and uploaded into GPU memory. For optimal performance, all voxel data is placed into a single buffer, which makes it necessary to manually manage the memory (see also Section 4.4).

Finally, all loaded blocks are drawn by issuing draw calls to the GPU. In this stage, only small optimizations to the draw order and batching are performed to ensure the best possible performance.

4.3.2 *Voxel decompression*

As mentioned above, using the rasterizer is important to get high-quality rendering due to its hardware-accelerated anti-aliasing. However, the voxel storage format is not suitable for direct rendering on the GPU. It consists of a compressed representation for a single voxel. This has to be unpacked into multiple triangles before it can be drawn using the rasterizer.

The input consists of at least voxel coordinates, which can be stored in block or world space, a list of active faces per voxel and optionally additional attributes like color and normals. There are two possible data layouts for the voxel attributes: Per-voxel or per-face attributes. Per-voxel attributes are constant over a whole voxel and always consist of one datum per voxel. Per-face attributes are constant per voxel face and vary in count per voxel.

For per-face data it is thus necessary to separate the storage from the voxel buffer, which has to be traversed using uniform stride. In my implementation, a separate per-face buffer is created and the offset into that buffer is appended to each voxel.

It is also possible to combine both per-face and per-vertex data by using otherwise unused bits in the position. With 8-bit positions and a 6-bit active face mask, two bits are unused per voxel which can be used to mark the payload as either data or an offset into a per-face buffer. A potential scenario for this approach is higher-quality simplification: If voxels are combined with widely varying normals, a higher quality approximation can be achieved by separating the faces and storing averaged per-face normals for the affected voxels.

The standard voxel format consists of three 8-bit positions, followed by the 6-bit active face mask, packed into 32-bit. Normals are quantized to 10-bit per channel and stored in 32-bit as well. If color is present, I store it using 8-bit per channel, padded to 32-bit as well.

As I want to use the compact representation as much as possible, the ideal place to decompress the data is on the GPU. This can be accomplished by combining vertex, geometry and fragment shaders.

Let's start with the first stage, the vertex shader. I use it to perform two operations: Unpacking of per-voxel attributes and backface culling. In the vertex shader, the per-voxel position, which is stored in unsigned integers, is extracted and placed into floats which can be consumed by the GPU hardware. At the same time, the view direction from the camera to the center of the voxel is computed and used to cull the active face mask. This is a crucial step as it guarantees that a single voxel will produce at most three visible faces. Without the culling step, the geometry shader stage has to handle an expansion ratio of zero to six faces per voxel; which is halved to one to three if the vertex shader culling is used. This significantly improves the geometry shader performance, as less memory has to be allocated in the worst case. The GPU is then able to execute more instances in parallel, resulting in higher geometry shader throughput.

In the geometry shader, the triangles for each voxel face a generated. The geometry shader takes advantage of the vertex shader culling, which guarantees that per axis at most a single face is present. It is invoked three times using instancing, once per axis, and uses a single bit-test to identify whether it is active or not. If active, a single face is emitted and multiplied with the current view transformation. Running the shader three times over the input with instancing turned out to be most efficient on modern GPUs, which have very high memory throughput and low geometry shader performance at higher amplification rates.

Finally, in the fragment shader, the lighting is evaluated. If per-face attributes are present, the fragment shader will fetch them to avoid having to allocate memory per attribute in the geometry shader. In all cases, the attribute interpolation is turned off, as all attributes are constant over the face (see also Section 2.3).

Rendering using the rasterizer is not without problems. On the up side, it allows to use the hardware anti-aliasing, which is a key requirement to achieve high-quality rendering. On the down side, it exercises the GPU in two ways which are not well optimized. First, the geometry shader is used for expansion. This is a slow path in the GPU hardware due to very limited usage in games. Second, GPUs are inefficient when rendering pixel-sized geometry. As explained in Section 2.3, the rasterizer has to set up edge-equations for each triangle. In addition, to provide derivatives for shading, the rasterizer will always generate a quad of 2×2 pixels, even for sub-pixel geometry [FBH+10].

While it is possible to work around the geometry shader limitations by using additional scratch memory, the sub-pixel sized triangles will remain a problem. However, as this issue is becoming more and more apparent for games using tessellation, it is likely that the inefficiency for small triangles will be eventually addressed by the hardware vendors.

4.4 MEMORY MANAGEMENT

At run-time, blocks have to be paged in and out of memory. This is an usage pattern which is not well suited for graphics APIs like OpenGL and Direct3D, which are not optimized to create and destroy buffers per frame. Instead of relying on the driver, the rasterizer manually manages the GPU memory.

The memory is divided into three pools, one for the voxels, one for index data and one for vertex data. Each pool is managed separately and is allocated at the expected worst-case size. The memory inside each pool is managed using a linear allocator. The allocation strategy is as follows: A high mark, indicating the upper end of the allocated memory, is increment on each request. For each request, the block identifier, the start position and the size is recorded. Due to hardware requirements, allocations are also padded to satisfy alignment properties. This scheme makes allocations extremely efficient, as they only require to increment one pointer.

For a deallocation, the corresponding request is moved onto a free list and the block is marked as deleted, but uploaded. This makes it possible to quickly restore deleted blocks. Before a block is reloaded from disk or for upload to the GPU, I check if the block is on the free list. If so, it is simply removed from the free list and the loading

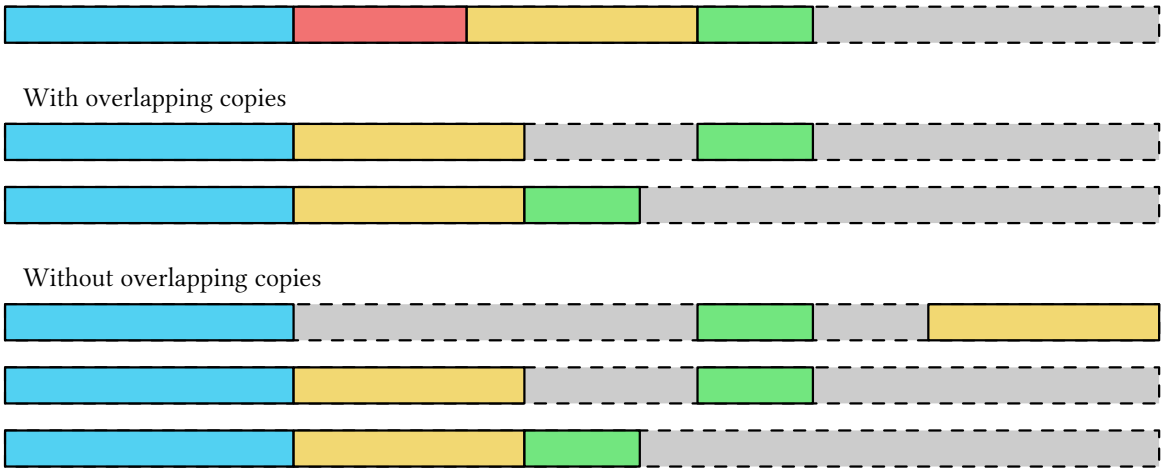
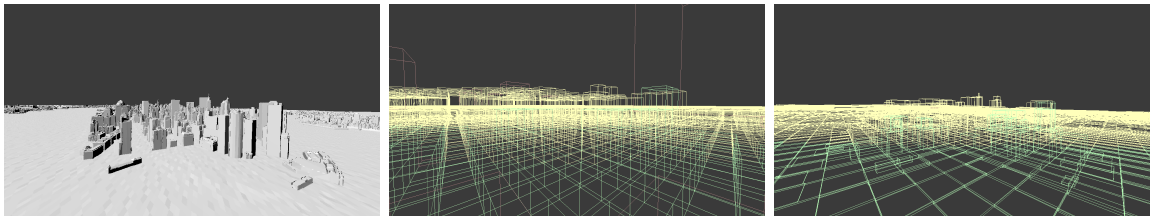


Figure 25: Buffer defragmentation: The red block is deleted. If overlapping copies are allowed, the yellow block is moved directly to the next start offset (middle). Otherwise, an intermediate copy is placed at the end of the buffer (bottom). In this case, one additional step is necessary before the green block can be moved to its target location.

is skipped. This is a very efficient method to avoid re-uploading of blocks which have been out of view for a few frames. Notice that allocations will prefer to not use the free list, but will be serviced from the upper end of the allocated range if possible. This guarantees tight packing of new data. In case of memory pressure, the allocator will start to evict blocks from the free list. As block sizes rarely match, this will result in fragmentation, which makes it necessary to defragment the memory pools regularly.

Once the buffer becomes sufficiently full or the fragmentation is too high, a defragmentation is triggered which compacts the data again. The recompaction purges the free list and moves all remaining blocks together. The packing processes sorts all allocations by their memory address. Once a gap is found, the next allocation is examined. If the size is less or equal to the gap size, it is moved directly into the gap. Otherwise, an overlapping copy is required. As this is not supported by all hardware and APIs, the memory manager will copy the block to the end of the pool and then copy it back if necessary. Eventually, all blocks are packed tightly again, with no gaps in between except when required by alignment (see also Figure 25.) The defragmentation process has a complexity of $\mathcal{O}(n)$, with n being the number of *visible* blocks in a frame.

For efficient uploading, one small – in the order of several MiB – staging buffer is used per pool. On the CPU side, the buffer is packed as tightly as possible and then transferred to the GPU. On the GPU, the data is scattered to the appropriate slots. Scattering on the GPU is more efficient than multiple round-trips from the CPU to



(a) Test scene (b) Octree node bounds (c) Precise node bounds

Figure 26: Comparison between precise and octree node bounds. Octree node bounds encompass the whole octree node and are much larger than the precise, content based bounds.

the GPU, as the CPU/GPU bus is optimized for large transfers and has a much lower bandwidth than GPU memory.

4.5 VISIBILITY

The renderer uses two culling algorithms to reduce the number of visible chunks. First, a frustum culling pass is performed, followed by occlusion culling.

4.5.1 *Frustum culling*

The frustum culling is integrated into the CPU side octree traversal. While descending through the tree to determine visible nodes, the bounding box of each node is also tested against the current view frustum. If a node is determined to be outside the view frustum, the traversal stops immediately and the node is marked as invisible. The frustum culling is conservative and only works on the node bounding boxes, which allows for very fast visibility tests. To improve culling efficiency, I use the precise node bounding box instead of the bounds which would be obtained directly from the octree. During the preprocessing, a tight bounding box over the contained voxels is pre-computed, which is generally much smaller than the octree bounds, especially near the leaf levels (see also Figure 26).

4.5.2 *Occlusion culling*

The occlusion culling has two goals: First, due to the approximative tests, the frustum culling misses many chunks which are close to the view frustum, but yet completely invisible. Second, for a typical scene, many chunks will be occluded by nearby geometry.

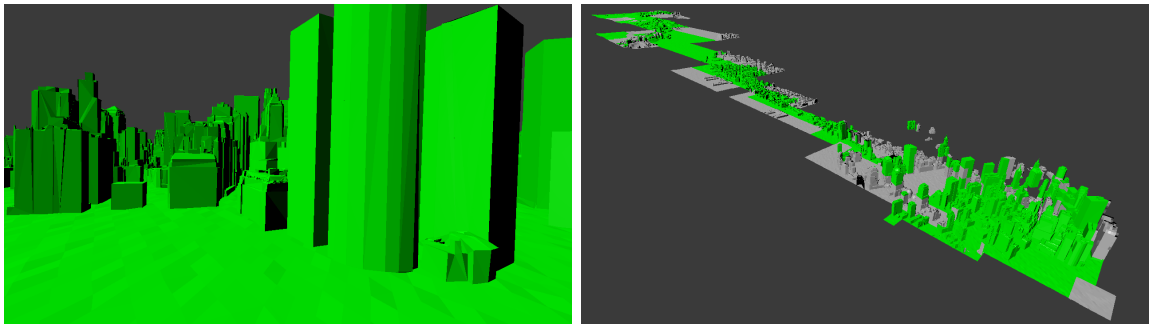


Figure 27: Top: Camera view into a city model. Bottom: Parts visible to the camera are highlighted in green, nodes rendered and determined invisible are gray. Notice that our occlusion culling algorithm pruned most of the scene and only few nodes adjacent to visible areas are rendered.

For example, every closed mesh will have roughly half its geometry occluded at any given time.

Occlusion culling using the rasterizer requires the use of occlusion queries. An occlusion query counts the number of fragments that pass depth and stencil tests for a set of draw calls. The renderer issues one occlusion query for each block that is rendered and reads back the results at the end of the frame. This requires a synchronization between the GPU and CPU, and as such, it is delayed as far as possible during the frame rendering.

During rendering, I force early depth/stencil testing and write to a buffer with one entry per block for every pixel that passes the depth test. Using a single buffer instead of occlusion queries is important for performance, as I 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. In order to further improve the efficiency of the occlusion step, all blocks are sorted by distance to camera before rendering. Geometry blocks are also rendered before the voxels, as the geometry is guaranteed to be at least as close to the camera as the voxels.

The occlusion query buffer itself is implemented using the `ARB_image_load_store` extension in OpenGL respectively as Unordered Access Views in Direct3D 11. It is simply a large integer buffer with one entry for each submitted block. If the fragment shader is executed, it will write a 1 to the appropriate slot without any synchronization.

Once the results are back, I 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 visibility bound of its children. This is a major difference to algorithms like [BWPP04], which render object bounding boxes. In my algorithm, I can take easily advantage of the level-of-detail simplification, which creates a tight convex hull for the underlying

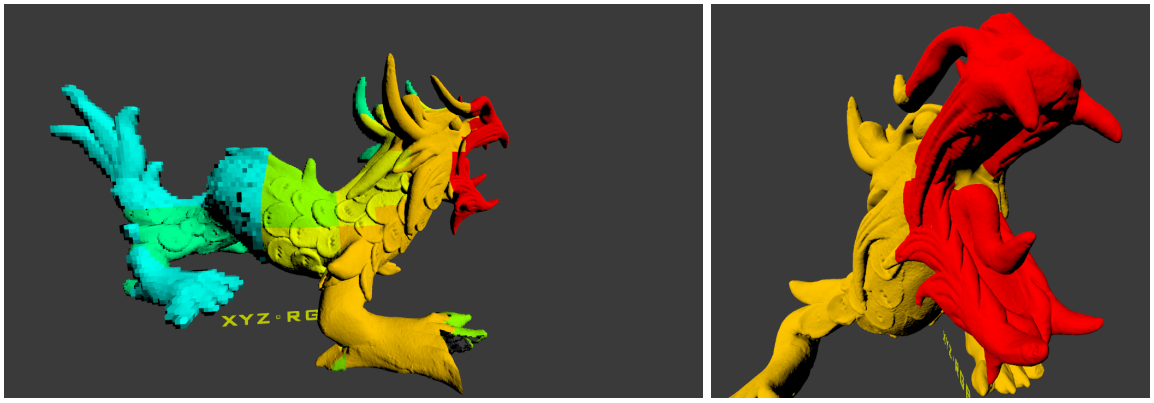


Figure 28: The octree traversal automatically reduces the level-of-detail for occluded areas. The inset shows the actual camera view, with colors indicating the level-of detail. In the side view, we can see that the occluded parts are rendered using drastically reduced level-of-detail.

geometry at any given level. As a result, a much tighter bound for the visible data set is obtained, as can be seen in Figure 27. This significantly improves the occlusion culling efficiency.

Once I determined that all children of a node are invisible due to occlusion, I 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, as the level-of-detail is conservative. As such, it guarantees that a visible block can never be missed. The effect of this optimization can be seen in Figure 28. The end result is that occluded geometry is reduced to the absolute lowest possible level-of-detail that still guarantees that nothing is missed, reducing memory usage and maximizing performance.

One issue with this method is flickering that occurs if a node is visible at a level-of-detail which requires refinement, but its children are not. In this case, the children will be rendered in one frame, determined to be invisible; in the next frame, the parent will be rendered, which results in a few visible pixels. In the next frame, the children will be rendered again, but as they cover no visible pixels, flickering will be visible every time the parent is rendered. I avoid this by disabling writes to the color buffer if I render a node only to determine visibility (see Figure 29.) This case can be identified by keeping a one-frame visibility history of each node.

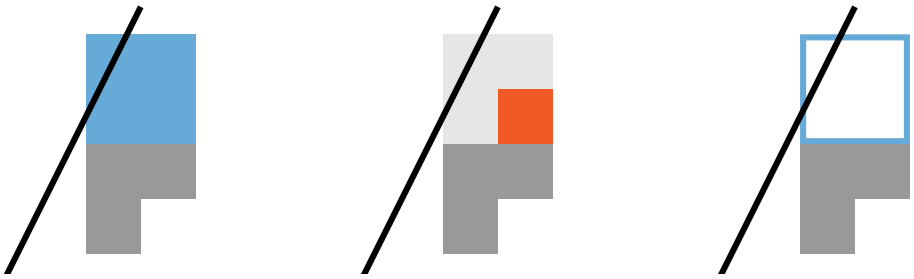


Figure 29: On the left, a view ray hits the blue octree node which should be refined. In the middle, the children of the blue node are highlighted in red. As all children are invisible, I mark the blue node for occlusion rendering only in the next frame. Otherwise, flickering will occur as the renderer will cycle through the first two states.

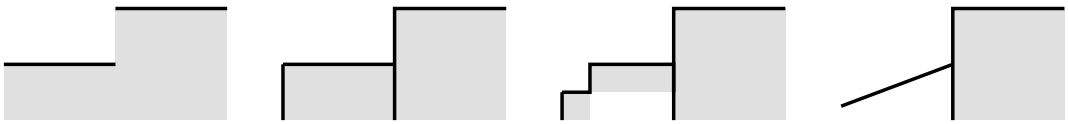


Figure 30: Transition between two blocks at different level-of-detail. On the left, the active faces are marked in black. Without any fixes, a gap may appear between the high- and low-resolution block. In the following two cases, the faces at the boundary have been closed using skirts, resolving the problem, even if the level-of-detail difference is higher than one. In the rightmost image, the left block has been refined to the source geometry, and the skirt ensures consistent rendering in this case as well.

4.5.3 *Level-of-detail*

Level-of-detail is implemented by taking advantage of the simplified voxel data from the pre-processing step. However, one minor modification is required to allow for crack free rendering. In the renderer, blocks of different levels will be placed adjacent to each other. This may result in visible cracks if the voxels in one block don't match up exactly with those from the neighboring block. Due to anti-aliasing, even tiny gaps will be easily visible and lead to visible flickering along block boundaries.

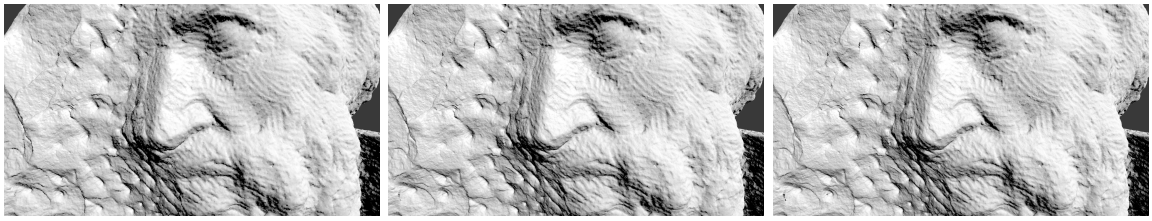
Fortunately, this problem can be solved using skirts. For each block, all voxel faces at the boundary are marked as visible. If the attributes are stored per-voxel, this change requires no additional memory, has only minimal impact on the rendering and resolves all potential cracks (see also Figure 30). Otherwise, additional per-face data is necessary. In general, the overhead is very minor as the skirts are only necessary for boundary voxels, and in particular at higher simplification levels, most of the boundary voxels will be closed anyway due to the simplification scheme.

4.6 STREAMING

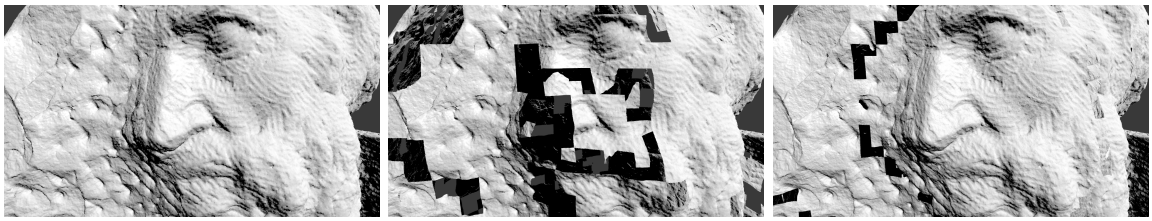
Streaming is necessary to reduce run-time memory usage and to allow for quick inspection of the data. Even though memory sizes have increased over the last years, making streaming for run-time memory reduction less important as it used to be, the relatively low increase in available bandwidth makes it still necessary. Especially if the data set is loaded from cold storage or over a network, it is crucial to only load the parts which are actually visible. This becomes even more important when level-of-detail is available, as the data set size for any given viewpoint is typically a fraction of the total data size.

Streaming is running asynchronously to the rendering and processing through a dedicated I/O thread. At the beginning of a frame, the scene tree is traversed and all nodes which have to be loaded are gathered. The load requests are then sent to the I/O thread, which proceeds to load data while the rendering continues. Once per frame, the data loaded by the I/O thread is transferred back to the main thread and uploaded to the GPU.

The I/O thread is synchronized once per frame to ensure that only missing data is requested. The process consists of three steps: First, the tree visibility update is performed which identifies all nodes that are needed for the current frame. Next, the I/O thread results are read and uploaded to the GPU. These may include blocks which have been requested by the previous frame which are no longer visible. While checking



(a) Refinement with full updates



(b) Refinement with partial updates

Figure 31: The two refinement strategies.

and omitting these blocks is trivial, uploading them to the GPU has the advantage that no disk I/O bandwidth is wasted. After the upload, the load requests for the current frame are processed. All blocks which have not yet been loaded are added to a request queue and sent to the I/O thread, replacing the current load queue.

The upload to the GPU can be optionally limited to ensure smooth frame rates. Without limitation, all data that has been loaded by the I/O thread will get uploaded, which can lead to long stalls and interrupt the rendering. In general, it is not necessary to have all results available immediately. Especially at high frame rates, waiting for multiple frames has very little to no visible impact.

As the streaming is driven from the tree traversal, it is integrated directly into the level-of-detail and occlusion culling schemes. In the first frame, the visible parts of the scene are determined and the required level-of-detail is computed. Parts which are not yet resident in GPU memory are requested and loaded from disk. This request-driven loading guarantees that only visible parts are loaded. Due to the level-of-detail refinement, which makes it necessary to load intermediate levels before the next level is found to be necessary, the data cannot be loaded immediately at the target level. Instead, the tree has to be refined in multiple steps until all data is present.

Streaming requires that the rendering side can handle missing data. In my implementation, the user can choose between two different strategies (see also Figure 31). The first one guarantees that a closed model is displayed at all times. If missing data is about to be displayed, the level-of-detail is reduced instead to ensure that some geometry is visible for all parts. While this guarantees that no holes are visible, it can lead to brief flickering when the traversal branches at a high level of the tree. If a path

(a) Zoom out with full updates

(b) Zoom out with partial updates

Figure 32: Zooming out from a dataset. The full update refinement strategy reduces the level-of-detail, while the partial update strategy simply omits missing geometry.

starting from the root node becomes visible, this approach has to reduce the resolution to the root level before it can refine again. This typically happens when the user zooms out, as new geometry becomes visible in this case. Zooming in onto the geometry does not lead to any flickering.

The second approach allows for missing data. In this case, partially loaded nodes are rendered with holes while blocks are being streamed in. The advantage of this approach is that no flickering occurs due to level-of-detail reduction, at the expense of missing geometry in some cases. This approach is complementary to the first one. During zoom-in, the first approach guarantees a closed mesh and good navigation, while the second approach exhibits holes. During zoom-outs, the first approach results in flickering, while the second one generally allows for a stable navigation with minor artifacts, at the expense of occasionally missing data (see also Figure 32).

Both modes can be easily switched at run-time, as they only affect the tree traversal. In the first case, visibility and loading information is gathered bottom-up to ensure a consistent configuration. Internally, a “visibility front” is computed which consists of the nodes with the highest depth in the tree that are fully loaded. This may require moving back up in the tree until a loaded intermediate node is found. In the second case, information is only propagated top-down. The tree is traversed depth-first and each loaded leaf is rendered. Intermediate nodes are always skipped. For both approaches, it is necessary to keep the root node in memory at all times to ensure a fallback path.

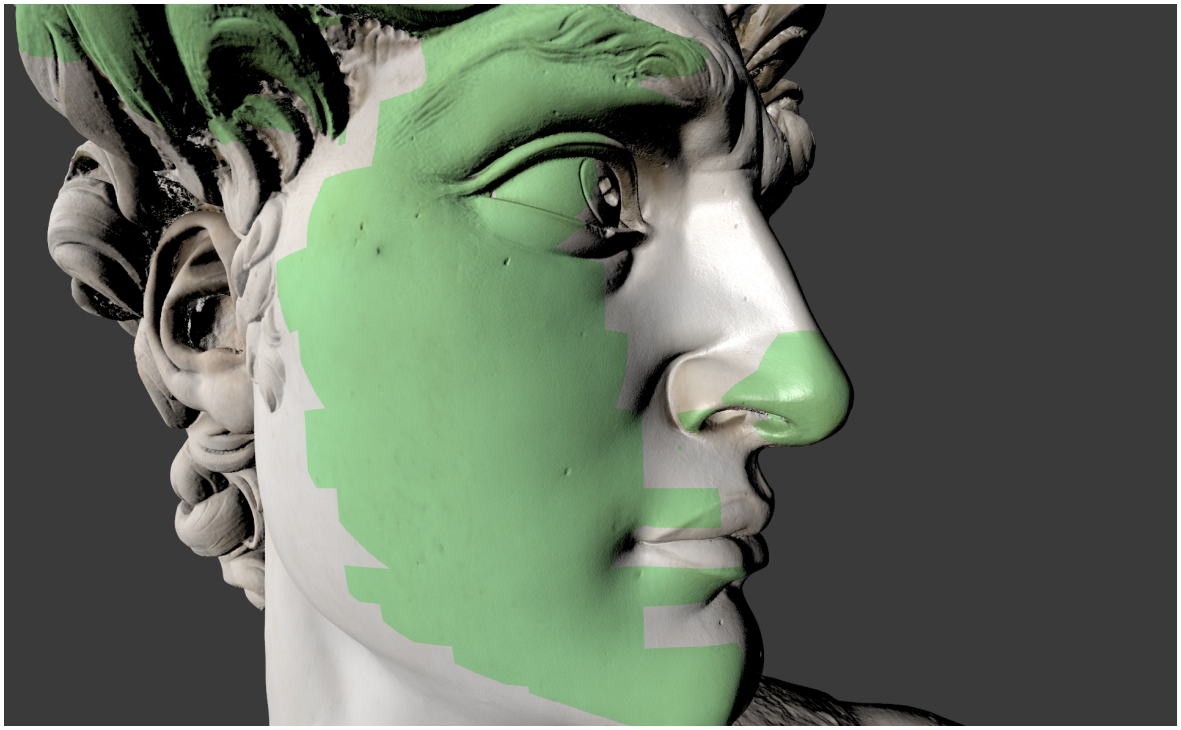


Figure 33: For close-up views, the source geometry, tinted in green, is rendered instead of the voxel representation. Due to the sub-pixel error and block skirts, no transition between the source geometry and the voxel representation is visible.

4.7 HYBRID RENDERING

The renderer can combine both source geometry and voxels for high-quality rendering. Once a voxel block at the finest level-of-detail requires refinement due to the error metric, it is replaced by the actual source geometry.

The replacement makes it necessary to split the mesh into blocks. Fortunately, this can be done at the same time as the voxelization itself. In the voxel rasterizer, each triangle is successively clipped until the leaf level. It is easily possible to simply emit all triangles fragments for a block "on the way" with minimal impact on the performance.

At runtime, the renderer uses two additional caches for vertex and index data, similar to the voxel cache with the same management strategy. Separating the caches makes it easy to balance memory usage between the different data types. During the visibility tree traversal, blocks which are at the highest resolution and still require refinement are marked for geometry rendering. The I/O thread then requests the geometry, and, once ready, the renderer switches to geometry rendering. For maximum efficiency, all geometry draw calls are batched and executed before the voxel rendering. This minimizes the amount of buffer bindings and state changes that need to be performed.

For each block at the finest level, both geometry and voxel data is kept in memory, even if only the geometry is visible. In general, the voxel data is much more compact – a block containing thousands of triangles can be associated with a block comprised of a few voxels. Keeping it in memory does not result in severe memory pressure and allows the user to quickly zoom out without having to re-upload the voxels.

4.8 RESULTS

I have measured the performance on three large 3D scans: `Atlas`, `David` and `St. Matthew`. For details on the data sets, see Section 3.3.1. For the rendering tests, the data sets have been resampled into 16384^3 resolution.

The performance has been measured on an AMD FirePro W9100 GPU with 16 GiB of memory and a dual-Xeon X5650 machine (2× 6C/12T) with 24 GiB of main memory. All tests have been run using OpenGL under Linux. Data was stored on a Samsung 840 EVO SSD with 1 TiB capacity formatted as ext4. Using `hdparm`, I have measured an uncached read performance of 265 MiB/sec. Unless noted otherwise, the sizes of the voxel-, index- and vertex-cache have been set to 1536 MiB.

I have measured the performance for several zoom-ins onto the data sets as well as a camera rotation around the `Atlas` mesh. For all tested data sets, the rendering achieves interactive frame rates between 50 and 100 ms per frame. The per-frame upload is typically in the order of a few MiB for all tested scenes while voxels are loaded. Once the geometry is requested, the upload becomes quickly a bottleneck (see Figure 35 and Figure 37). In the unlimited cases, the asynchronous loading has been completely disabled. This means that for each frame, all required data is loaded from disk which results in occasional stuttering.

As previously mentioned, this behavior can be mitigated by using an upload limit. In this case, the asynchronous load thread is used to fetch data from disk as fast as possible while the per-frame upload is limited. In general, the disk I/O is slower than the GPU upload, as can be see in Figure 35 between frame 0 and 800. After the 800th frame, the GPU upload is limited to ensure a smooth frame rate. This has only minimal visual impact but improves the interaction performance significantly, as the camera movement is more predictable due to the consistent frame rate. Increasing the resolution from 1280×720 to 1920×1200 only results in more requested data for this test case, as everything fits into the caches (see Figure 34 and Figure 35).

In the `David` scene, increasing the resolution from 1280×720 (see Figure 36) to 1920×1200 requires a defragmentation of the memory during the zoom-in. This can be see in Figure 37 around frame number 975. It is triggered as the vertex cache

becomes more than 95% full. This is the high-watermark at which I perform the defragmentation to ensure that there is enough space left for moving blocks back and forth, if needed.

Rotations or other coherent movements around an object require minimal upload per frame. This can be easily seen in Figure 39, where the camera was fixed on the rotating `Atlas` mesh. Except for the initial loading, less than 1 MiB of data is uploaded per frame during the whole sequence. Towards the end of the rotation (approximately around frame 700) the initially seen parts of the mesh come into view again, further reducing the upload bandwidth.

The GPU upload limit is rarely hit, even though it is set to very conservative 1 MiB. To validate this, I have tested the `St. Matthew` scene with an upload limit of 4 MiB. As mentioned, the upload limit only affects the CPU to GPU transfer; if the disk I/O is slower, the limit has no effect. In Figure 38, the upload limit is only reached once geometry is loaded. A key point here is that even at 1 MiB/frame, the disk I/O thread is fast enough to load all required data until approximately frame 900. This can be easily seen by the total amount of cached voxel data, which reaches 400 MiB in both the unlimited and limited case.

4.9 MODIFICATIONS & DYNAMIC CHANGES

Block rendering is performed using the rasterizer, which does not require a per-block acceleration structure. For voxel-only data sets, it is thus possible to quickly perform changes like applying stencils or cutting out voxels. In this section, I will briefly cover a few of the possible extensions. The key assumption here is that the geometry is represented using only voxels.

A very simple modification is the application of stencils, or "cut-outs". This can be performed at multiple stages: Either on the CPU, by removing the affected voxels, in GPU memory by clearing the active face mask per voxel or in the vertex shader. If a stencil is applied only once, performing the operation in CPU memory is most efficient. For run-time changes, for example, when moving a "see-through" brush, the changes should be performed in the vertex shader to minimize the amount of memory traffic.

For small volume data, it is also possible to combine the surface extraction with the rendering and run it in real-time on the GPU. For example, a "build" volume can be created using an 8-bit density volume which is then edited using a 3D brush. On every modification, the surface extraction is performed and the resulting data is rendered (see also Figure 40). Level-of-detail can be then lazily built starting from the highest resolution blocks. Using an underlying volume is necessary in this case, as the surface

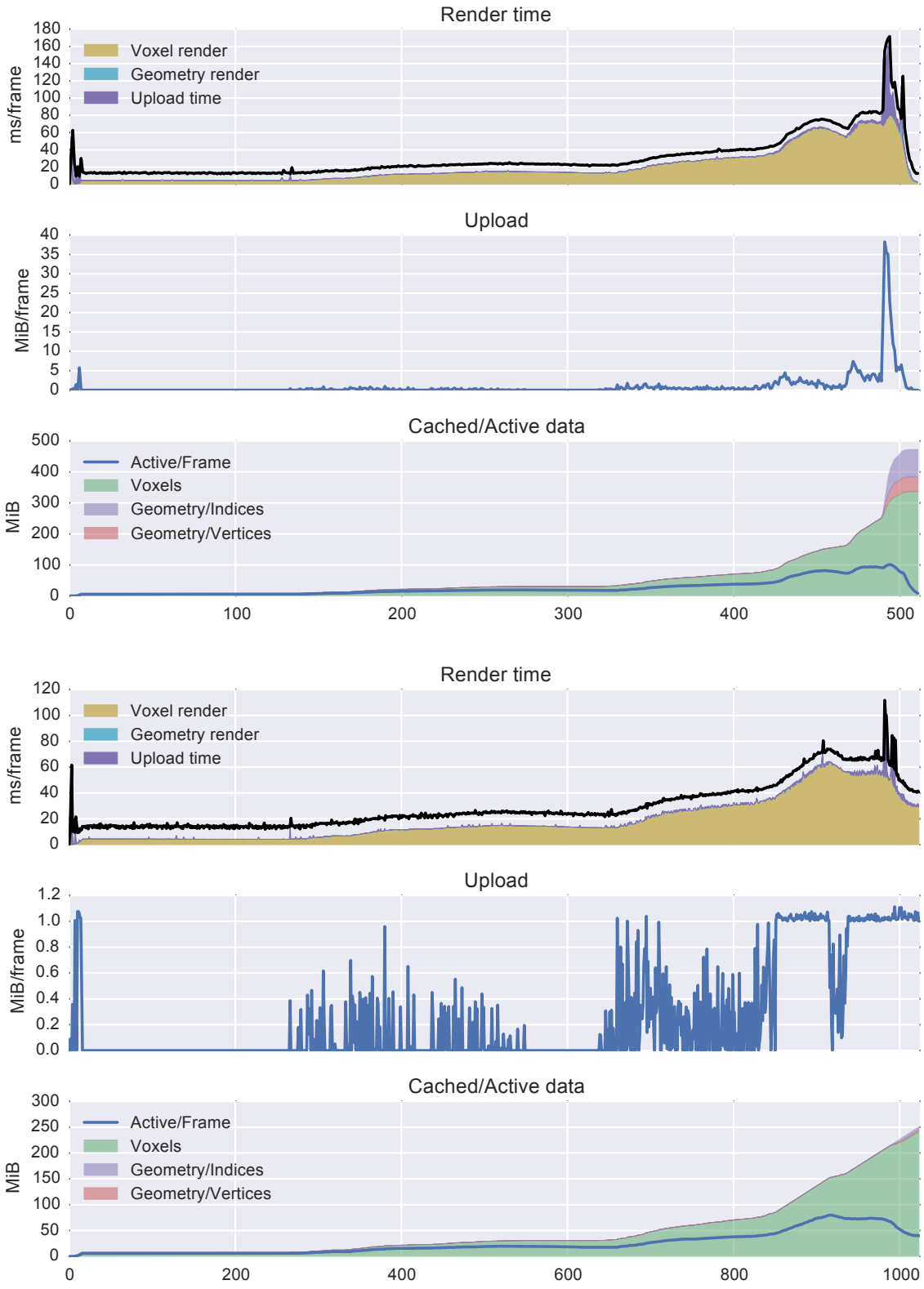


Figure 34: Measurements for a zoom-in onto the `Atlas` data set at 1280×720 and $8\times$ MSAA. Top is with upload limitation, bottom without.

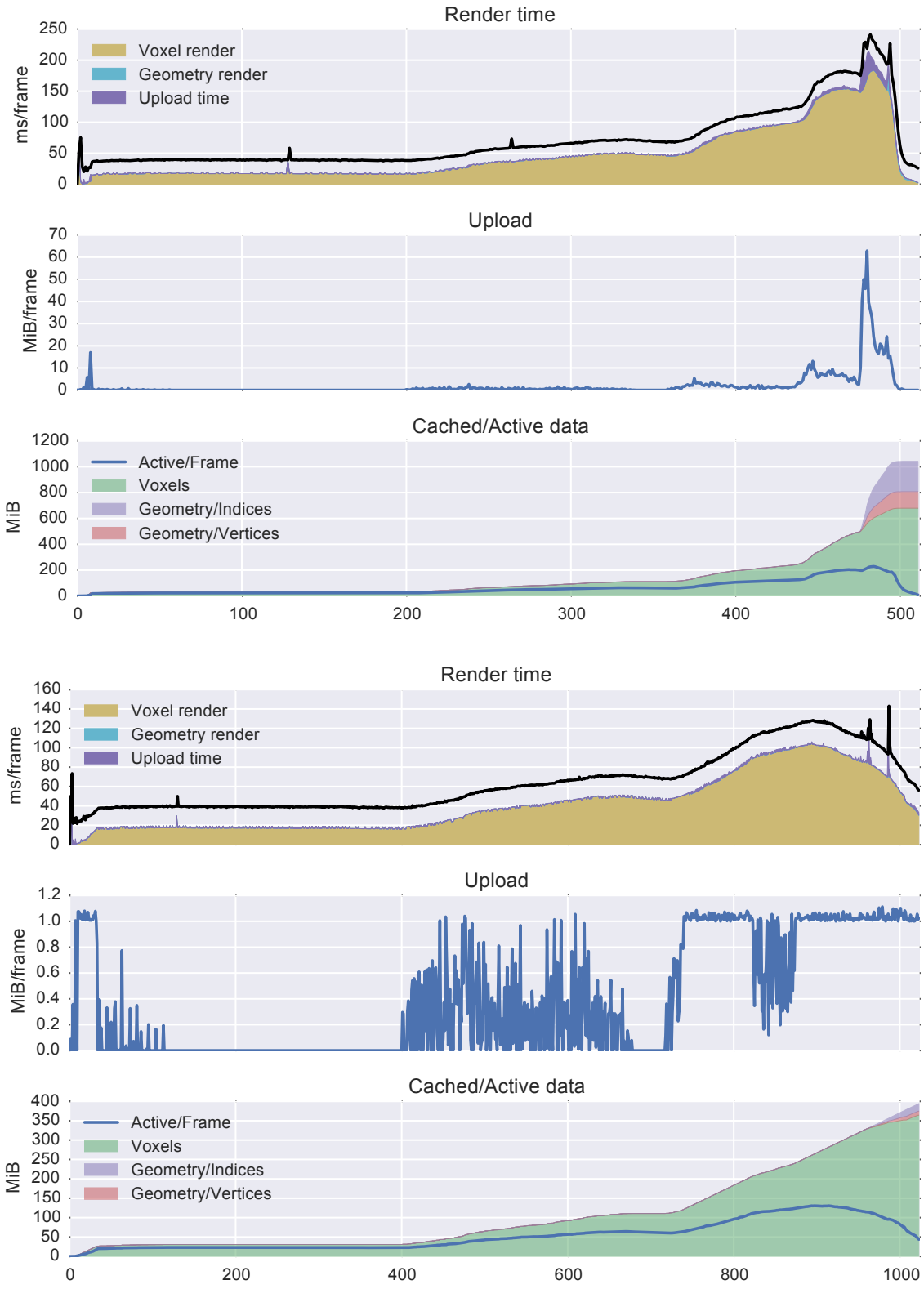


Figure 35: Measurements for a zoom-in onto the `Atlas` data set at 1920×1200 and $8\times$ MSAA. Top is with upload limitation, bottom without.

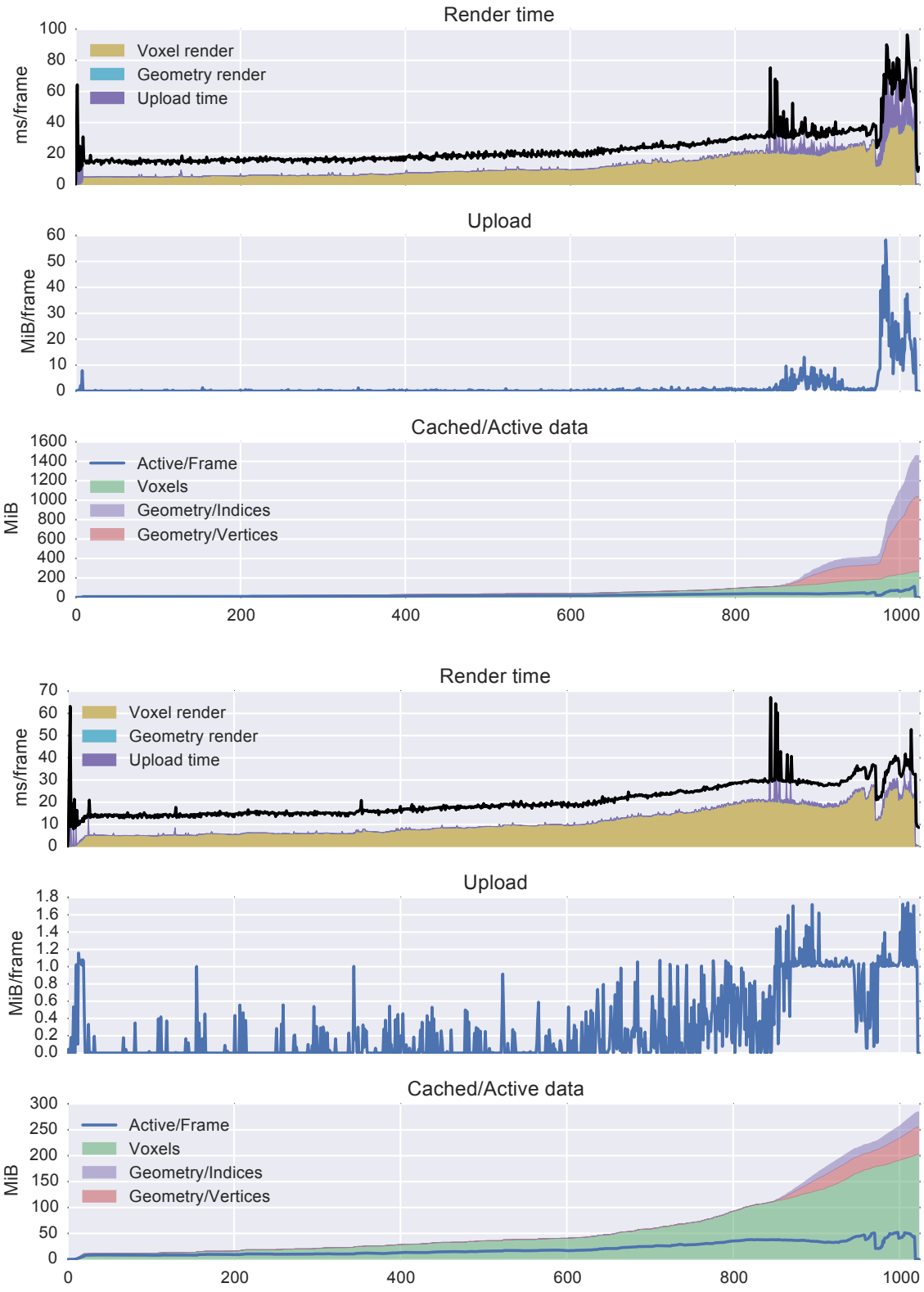


Figure 36: Measurements for a zoom-in onto the `David` data set at 1280×720 and $8\times$ MSAA. Top is with upload limitation, bottom without.

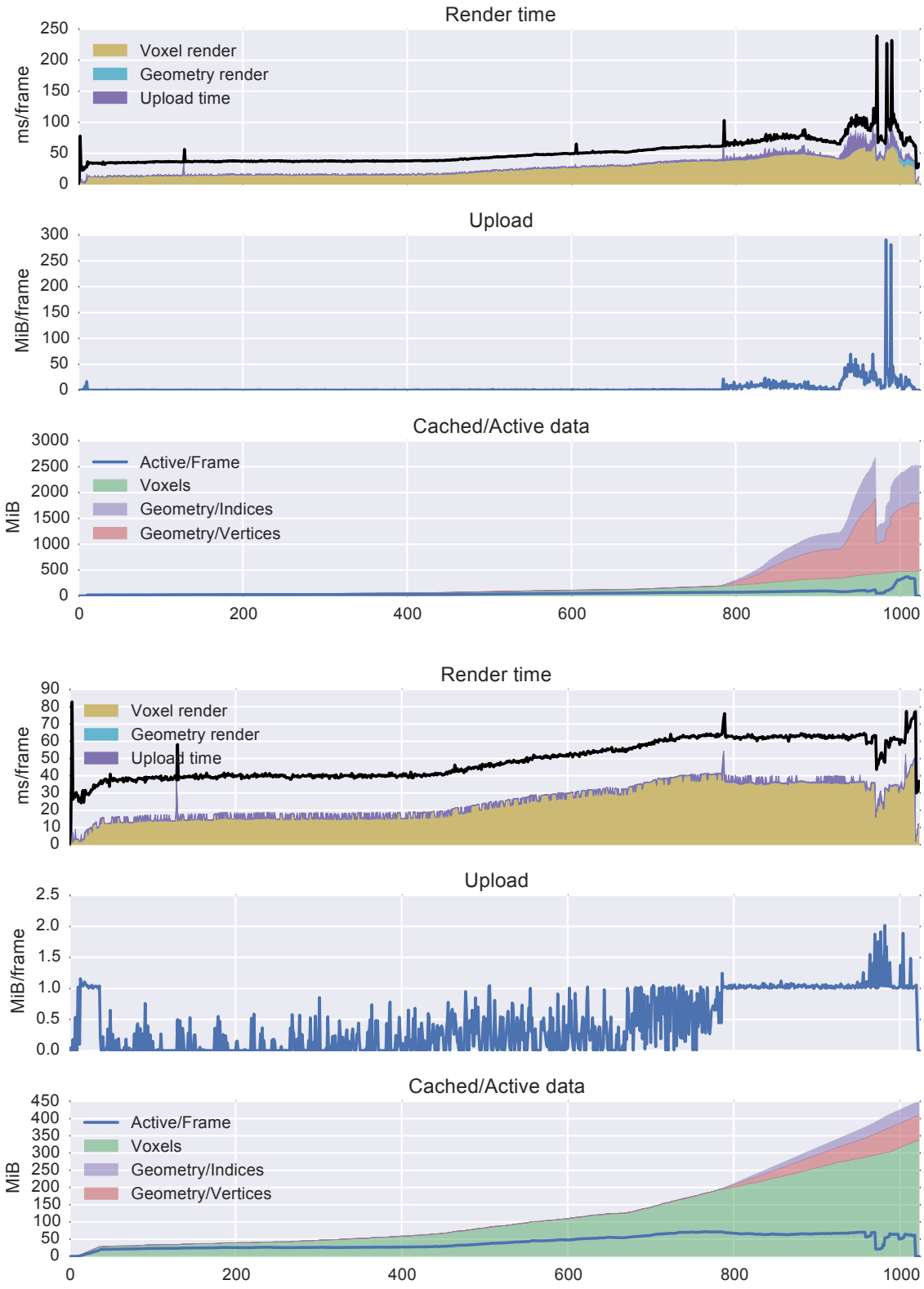


Figure 37: Measurements for a zoom-in onto the `David` data set at 1920×1200 and $8\times$ MSAA. Top is with upload limitation, bottom without.

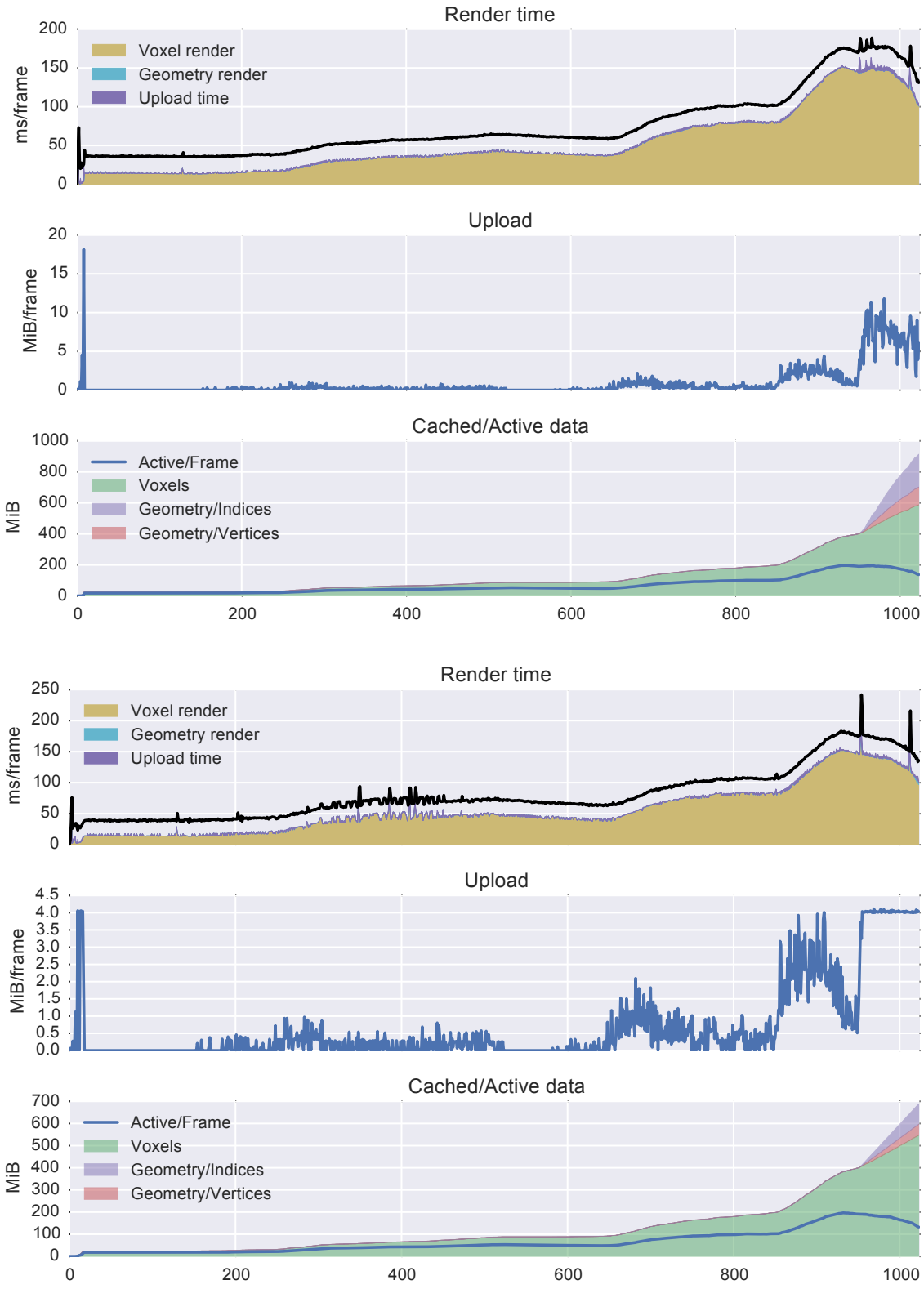


Figure 38: Measurements for a zoom-in onto the `St. Matthew` data set at 1920×1200 and $8\times$ MSAA. Unlike for the other scenes, the upload limit has been set to 4 MiB in this scene. Top is with upload limitation, bottom without.

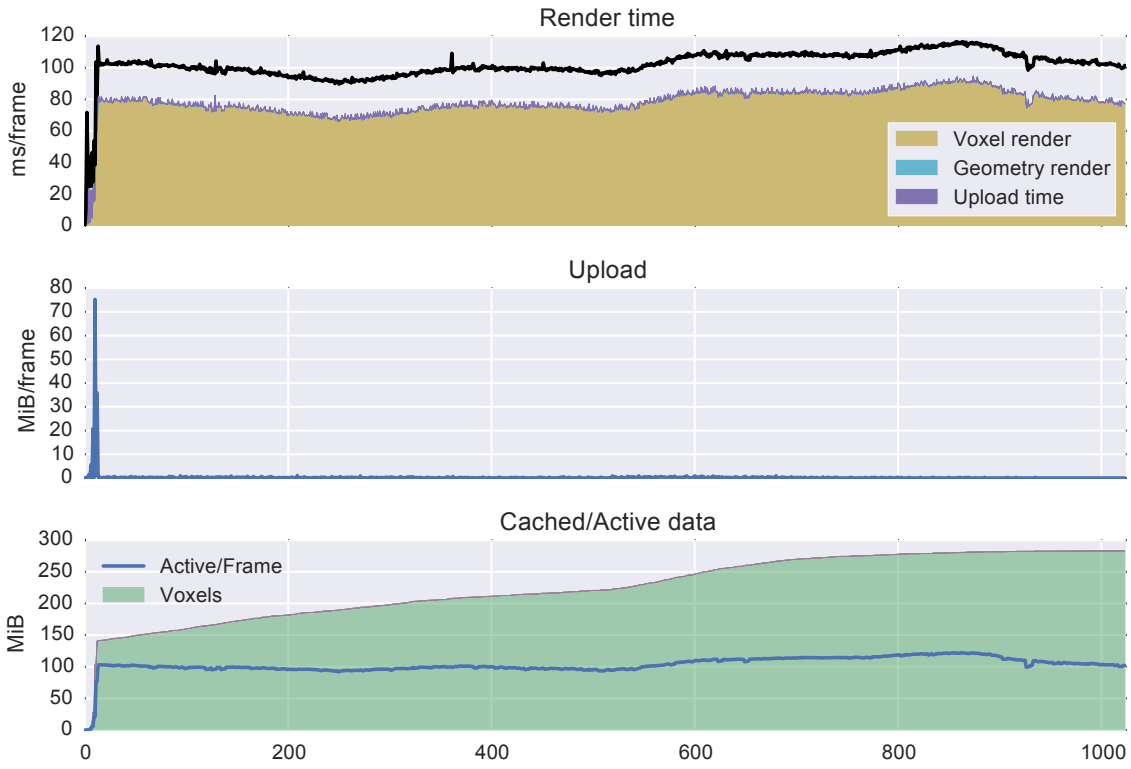
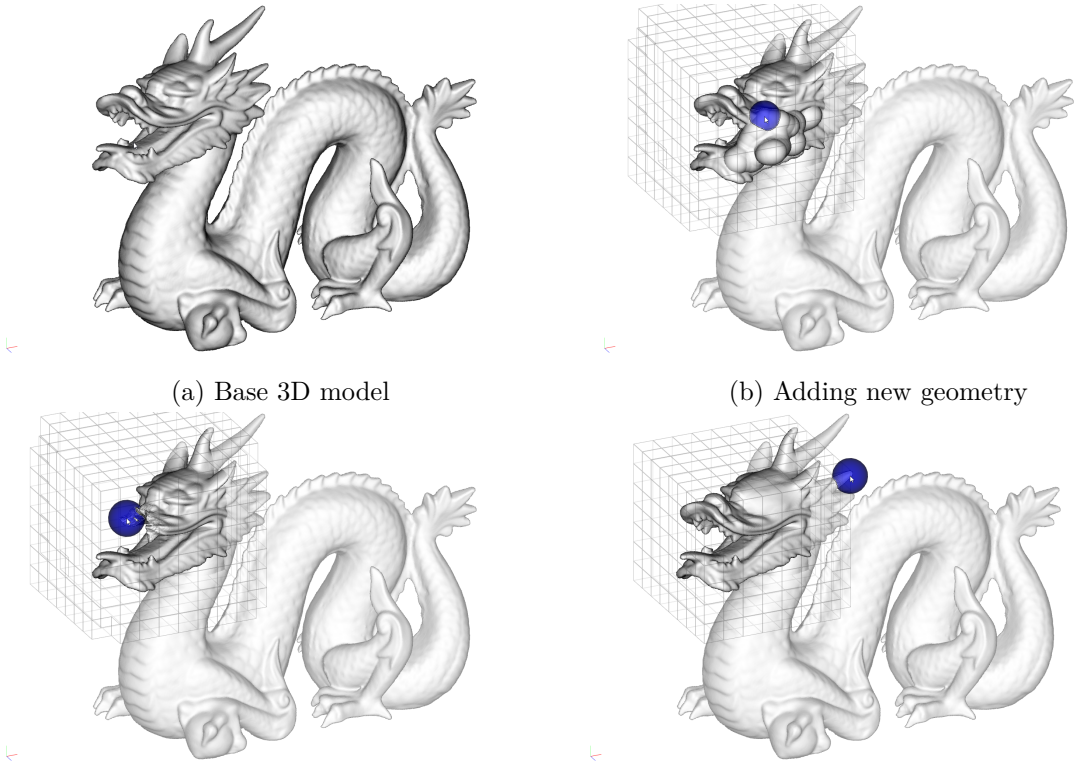


Figure 39: Measurements for a rotation around the `Atlas` data set at 1920×1200 and $8\times$ MSAA. This scenario has very high coherency, which leads to very low upload bandwidth requirements.



(a) Base 3D model

(b) Adding new geometry

(c) Removing geometry

(d) Smoothing the underlying volume

Figure 40: Dynamic editing in VOTA [CRS13], a 3D modeling package which uses a rendering approach derived from the technique described in this work.

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

Table 8: Rendering time for the `Atlas` data set at different multi-sampling levels and graphics cards onto a 1280×720 viewport. Both cards exhibit only minimal slowdowns with higher multi-sampling levels.

voxels have no topological information and no connectivity and as such performing editing operations on them directly is not possible. For triangle meshes, this makes it necessary to perform a conversion into a solid voxelization, for example, using wavelet rasterization [MS11].

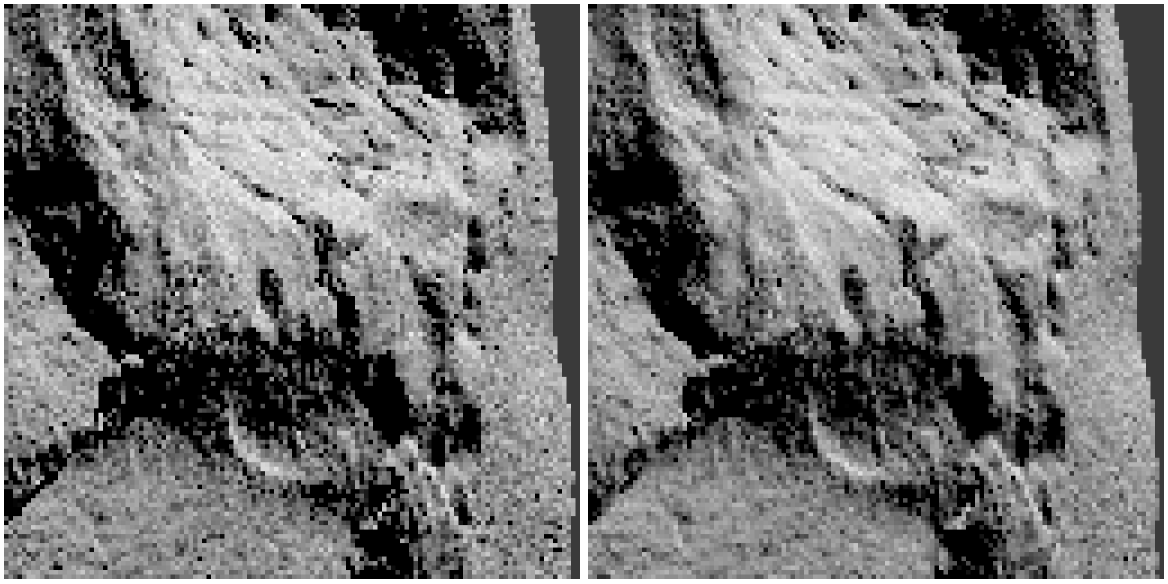
Finally, it is also possible to deform and animate the mesh. As long as the deformation is applied uniformly to all vertices, the mesh will remain connected and can be rendered correctly. For instance, the model can be easily sheared or displaced using a vector field as long as voxel vertices remain connected to each other. The only required change to the view pipeline is an adjustment for the new boundaries during the frustum culling phase.

4.10 ANTI-ALIASING

Anti-aliasing is crucial to obtain a stable and clean rendering. In the case of the high-resolution 3D scans, aliasing stems from two sources: Aliasing caused by the high-frequency geometry as well as aliasing from the rasterization. For the former, the voxel based resampling and simplification acts as a low-pass filter over the geometry. However, even when seen at the correct level-of-detail simplification, the `Atlas` data set still exhibits aliasing, as can be seen in Figure 41. This effect is even more pronounced under motion. This kind of aliasing can be resolved by using additional samples during the rendering.

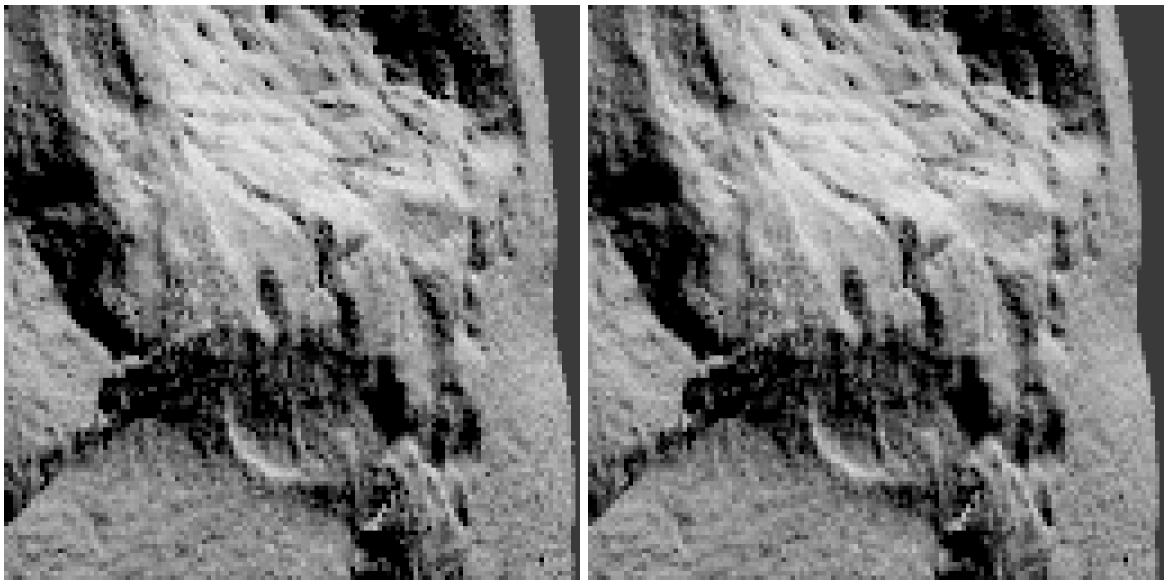
Due to the use of the rasterizer hardware, the scene can be easily and efficiently anti-aliased using hardware-accelerated multi-sample anti-aliasing. Good quality can be obtained with $4\times$ MSAA (see also Figure 41). Unlike for GPU raytracers, the anti-aliasing is also very cheap and has only minimal impact on the performance (see Table 8).

The performance can be further optimized by separating the shading frequency from the sampling frequency. As the geometry is at most pixel-sized, many triangles are part of a single pixel which have to shaded at least once. Yet many of those triangles



(a) 1 sample per pixe

(b) 2 samples per pixel



(c) 4 samples per pixel

(d) 8 samples per pixel

Figure 41: Comparison between different anti-aliasing levels. In the image, a 128×128 crop of the `Atlas` data set has been magnified. At least $4\times$ MSAA is required to obtain a smooth, clean image.

are part of the same voxel, and result in exactly the same color value after the shading has been performed.

If more complex shaders are used, this can be exploited by subpixel-reconstruction anti-aliasing (SRAA). The key idea behind SRAA is to reconstruct which shading samples correspond to the same input primitive and shade those only once.

Originally, SRAA was designed for deferred rendering in games. It is a post-process anti-aliasing algorithm which takes shaded input at low frequency – typically, one sample per pixel – and a super-resolution geometry buffer – four samples or more. It assumes that the shading is the most expensive part of the rendering pipeline and thus tries to minimize the amount of shader execution.

At filtering time, SRAA tries to associate geometry samples with shading samples. For low-resolution geometry, where each pixel is fully covered by a single triangle, SRAA is equivalent to MSAA, albeit running as a post-process and a slightly larger filter.

In the context of voxel rendering, SRAA can be used to enforce a shading frequency of approximately one sample per pixel at the cost of a separate shading pass. For complex, physically based shader models and scenes with multiple shadow maps, a deferred shading system combined with SRAA for voxel models is likely to outperform forward rendering.

5

GPU RAYTRACING

In this chapter, I will describe my analysis of raytracing acceleration structures on wide architectures with a focus on voxel rendering. Voxels are an interesting analysis target due to their unique requirements on the underlying data structure.

5.1 INTRODUCTION

In recent years, voxel raytracing has become increasingly popular. In the previous chapter, I have shown how rasterization can be a viable alternative, yet there are many use cases where pure raytracing is desirable.

So far, octrees have been the data structure of choice for raytracing thanks to their simple traversal routines and the easy integration of level-of-detail schemes. This stands in stark contrast to triangle raytracing, where bounding volume hierarchies and kD trees are the dominant data structures due to their superior spatial partitioning properties. The choice of octrees for voxel rendering was mostly motivated by the level-of-detail aspect, but the question whether octrees are a the best data structure for highly regular data like voxels remains open.

One aspect which has not been well researched yet is how data structures affect traversal coherency on architectures with very wide vector units such as GPUs. Previous approaches to investigate the coherency have been performed using SIMD simulation, but not measured on real GPU hardware. The problem with simulations is that caches, switching between different work groups and other secondary aspects are very hard to simulate and often omitted. Fortunately, it is now possible to get precise measurements of various metrics like execution coherency, cache usage, cache hit rates and bandwidth usage directly from the hardware through *hardware counters*. This method has been standard on CPUs, where it is widely used for profiling, but has only been recently available on GPUs.

In this work, for the first time, I use the hardware counters on a GPU to provide comprehensive and accurate measurements of the behavior of different acceleration structures. In particular, I measure and analyze how bounding volume hierarchies, kD trees and octrees behave in a wide range of scenarios. My analysis shows how coherency, performance and data structure design are directly related. With the pro-

vided information, I hope to pave the way towards coherency-optimized GPU data structures.

Besides its current popularity, one reason for analyzing voxel raytracing is that the used acceleration structures, unlike those typically used for triangle raytracing, do not have overlaps at the leaf level. In triangle scenes, it is very likely that leaf nodes overlap and each, and the handling these overlaps has significant impact on the overall performance. In many cases, the tree has to traversed multiple times before a hit can be reported, as all overlapping nodes have to be processed before the closest hit can be identified.

In voxel raytracing, most of the execution time is spent in tree traversal, making it the ideal vehicle to focus the analysis on the acceleration data structure. Additionally, I have chosen a very wide SIMD architecture which puts further emphasis on the coherency.

5.2 RELATED WORK

Voxel raytracing has been performed on GPUs since many years now [CNLE09, LK11a]. So far, all GPU voxel raytracers use an octree as the underlying acceleration structure due to the simple integration of level-of-detail. Interestingly, nearly no work has been done on exploring alternative data structures for voxel raytracing.

This is particularly surprising as kD trees and bounding volume hierarchys have been the gold standard for triangle raytracing for many years. This has carried over to GPU triangle raytracing as well [WMS06, WIK+06, PGSS07, HSHH07, GPSS07]. An early work which focused on a detailed analysis of GPU raytracing is [AL09]. In their work, a SIMD simulator was used to identify the best traversal pattern and subsequently validated on actual hardware. The focus has been placed on the traversal, and not on the data structure, for which a standard bounding volume hierarchy was used. While the results were validated on the actual hardware, only total execution time was measured.

One of the main differences between GPUs and CPUs are the very wide vector units. On the GPU, there is no choice but to use packet raytracing with 32 rays or more. Such wide packets have significant impact on the obtainable performance. Even on CPUs, with 4-8 wide vector units, packet raytracing is not always feasible. [BEL+07] found that packet traversal was only beneficial for primary rays. For incoherent rays, [WWB+14] show that it may be even more efficient to switch to single-ray traversal. These coherency problems observed on relatively narrow vector architectures become more pronounced on 32 & 64-wide vector units as used by current GPUs. Several

techniques have been proposed to improve performance. The focus has been placed on the actual execution of the traversal, i.e. by re-scheduling of rays as proposed in[AK10, BAM14], but not by investigating how the acceleration structure affects coherency.

Data structures themselves have only come into attention very recently, in particular, [AKL13] has investigated in detail why bounding volume hierarchies do not perform as good as expected on GPUs. They introduce two important metrics beside the standard SAH cost metric: The *end-point overlap* (EPO) and the *leaf-count variability* (LCV). The EPO describes how many nodes any given point in the scene overlaps. For triangle bounding volume hierarchies, this numbers is typically greater than zero, as leaf nodes are highly likely to overlap. The LCV describes the standard deviation of the number of leaf nodes intersected by a ray. The higher the variance, the more likely it is that the intersection kernel will become more incoherent as some rays will intersect the leaf while the rest will be traversing the tree. In this work, only one data structure is used, but many different builders are compared. Together with the SAH, the EPO and LCV provide a much better explanation for the observed GPU performance.

One important difference between their analysis and this thesis is that I investigate voxel raytracing, which has an EPO of 0 for all data structures. This means that unlike a normal triangle ray-tracer, which has to spend a significant amount of time to resolve overlapping leafs, the raytracing kernels described in this work spend nearly all of their time in the tree traversal. The impact of leaf intersections on the measurement is thus significantly reduced, allowing me to focus on the acceleration structure characteristics instead.

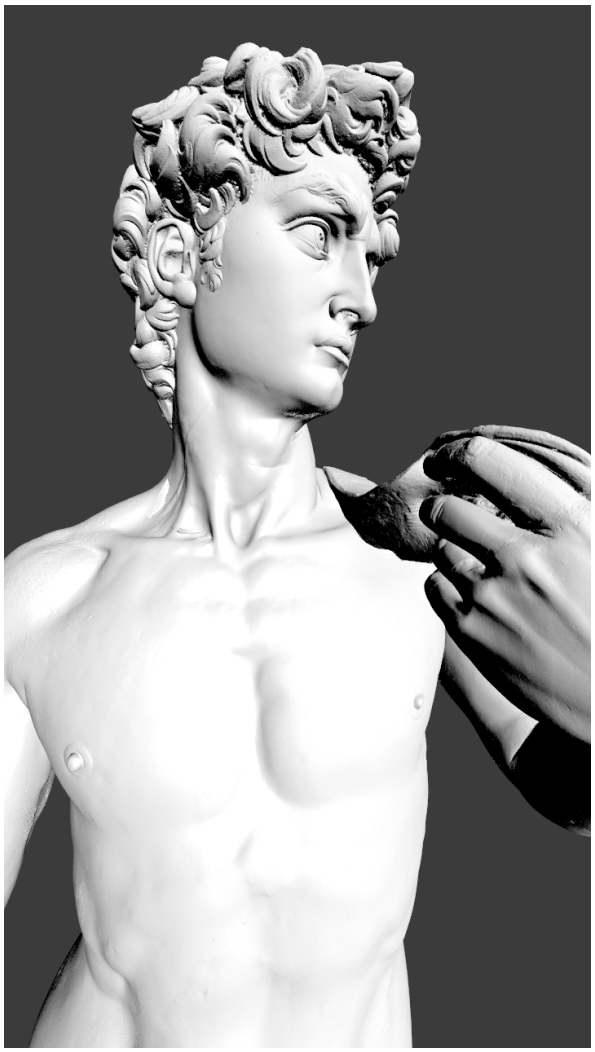
5.3 IMPLEMENTATION & TESTING METHODS

In this section, I will describe my test methodology in detail. This includes the test scenes as well as the tree traversal and building routines.

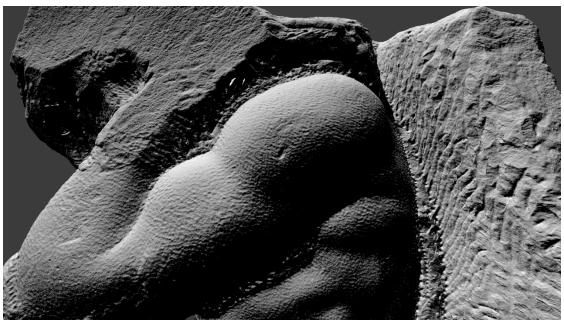
The test scenes (see also Figure 42) are created from a six-separating voxelization to minimize the number of generated voxels (see also Table 9). The data sets have been generated separately for each resolution and not obtained from a simplification.

Each voxel consists of the world-space position, a normal obtained from the interpolated input geometry and a color. High-quality per-voxel normals are necessary to obtain a correct hemisphere for ambient occlusion ray-tracing. For simplicity, all data sets are stored with color, even though only `David` has actually a color channel. For the other data sets, the color is set to a constant white.

I have limited the maximum scene size to 65535^3 voxels. This allows me to use 16-bit bounding boxes for the BVH and 16-bit plane offsets for the kD tree.



(a) David



(b) Atlas

(c) Conference room

(d) San Miguel

Figure 42: The test scenes used for the analysis.

The surface normal and color are stored in a separate buffers. This maximizes the cache usage when tracing "any"-intersection rays, as those don't have to fetch the voxel normal. In total, three buffers are used: The voxel buffer storing 16-bit, unsigned integers, the normal buffer storing quantized 10-bit per channel normals, padded to 32-bit and the color buffer storing 24-bit color, padded to 32-bit.

To simulate level-of-detail, I have voxelized each test scene at different resolutions. For the target rendering resolution (1280×720), a voxel resolution between 2048^3 and 4096^3 results in the correct level-of-detail, while 1024^3 is undersampled and 8192^3 is oversampled. For two data sets, I have added a 16384^3 voxelization to identify the impact of highly oversampled data.

For a fair comparison, all data structures use world-space voxels. While it is possible to further reduce the size of a voxel by storing it in node-relative coordinates for the octree, this would require a separate intersection routine and bias the results by improving the cache efficiency on the octree. Hence, all voxels are stored using 16-bit coordinates and the same leaf intersection code is shared between all data structures.

The data structures are generated in an offline process on the CPU and linearized before uploading to the GPU. As the GPU has a separate address space, this step is necessary to ensure that the links between the nodes remains valid. Just before upload, the pointers are replaced by 32-bit integer offsets into the respective buffers. Additionally, the data structure is split into separate buffers for the leaf and interior nodes. As most of the traversal time is spent accessing interior nodes, splitting the tree results in improved locality and reduces memory usage.

I have implemented all traversal kernels using OpenCL [Khr12] and integrated them into a common, OpenCL based raytracing framework. For the actual measurement, I have instrumented the framework using the AMD `GPUPerfAPI`. This is the same API used by AMD's profiling tools. On the tested target architecture the hardware counters provide accurate vector unit utilization, cache hit rates as well as memory usage.

I have tried to keep the traversal routines short and as divergence-free as possible. In particular, I have not used CPU-focused optimizations which introduce additional branching. For instance, I avoid filtering of `NaN` results with branches. Instead, I rely on careful ordering of operands and the IEEE mandated `NaN` handling of the min/max instructions.

Another example is the empty node handling, which is necessary for the kD tree and octree. I chose to push them during interior traversal and let the leaf intersection routine handle empty nodes. This improves coherency as other rays are likely to traverse through a leaf in such a scenario as well. The alternative would be to skip an empty leaf directly in the interior node intersection code. This requires an additional branch

Dataset	Resolution	# Voxels	Size
Atlas	1024^3	1197215	16
	2048^3	4857871	65
	4096^3	19832781	265
	8192^3	81202823	1084
	16384^3	332965775	4446
Conference	1024^3	3185580	43
	2048^3	12989135	173
	4096^3	52280443	698
	8192^3	209928684	2803
David	1024^3	615490	8
	2048^3	2477190	32
	4096^3	10007754	134
	8192^3	40725349	544
	16384^3	166481909	2223
San Miguel	1024^3	2648049	35
	2048^3	10788306	144
	4096^3	43676406	583
	8192^3	176316146	2355

Table 9: Overview of the tested data sets. The size is the total size of all voxels including normal and color data in MiB.

(a) Primary rays

(b) Ambient occlusion

(c) Shadows

(d) Soft shadows

Figure 43: The four ray types used in the tree analysis.

and stack pop call in the already branch-heavy interior traversal code, which reduces coherency.

For each scene, four separate tests are run to measure the coherency of primary and secondary rays (see Figure 43). For secondary rays, I have used three different tests with varying levels of coherency. For high coherency secondary rays, I use shadow rays to a directional light source. For medium coherency, I trace shadow rays to a directional light source in a narrow cube, and for low coherency rays, I trace ambient occlusion rays in a hemisphere. The soft shadow rays and ambient occlusion rays use 36 rays per pixel. For the ambient occlusion, the maximum ray length is set to $\frac{1}{3}$ of the scene.

5.4 ACCELERATION STRUCTURES

I have used a bounding volume hierarchy, an octree and a kD tree in my testing. All of the data structures have been adapted to voxel raytracing.

5.4.1 *kD tree*

The kD tree is built using a standard SAH builder (see also Section 2.5.4). This turned out to be crucial to get a high-quality kD tree which can be rendered efficiently. For example, a middle-split kD tree results in roughly 10× slower raytracing performance. The builder uses a binned SAH estimation with 32 bins. At each step, voxels are binned along their major axis and the SAH cost is computed for each bin.

I use a straightforward adaption of the triangle SAH cost heuristic for voxels. Each surface voxel uses the same surface area, that is, each voxel is assumed to be a cube with six faces which are equally likely to be hit. With this assumption, the cost of a voxel is simply normalized to one and only the number of voxels is taken into account. This could be optimized further by using the active surface mask of a voxel and estimate the surface area using the number of active faces.

A split may create an empty leaf node. I mark these nodes using a special `null` marker.

The traversal routine (see also Algorithm 5.1) computes the intersection between the ray and the current node's splitting plane. If an intersection has been found, the traversal continues into the closer node. The other node, which is guaranteed to be further away is pushed onto the stack, together with the current ray maximum. Just like the bounding volume hierarchy and octree traversal, once a leaf node is reached, the ray is intersected with all primitives. If no intersection is found, the stack is popped; if the stack is empty, a restart is performed.

If a `null` node is encountered, the traversal skips the leaf intersection and directly continues to pop the stack. A major difference between this traversal routine and the default kD traversal (see Algorithm 2.1) is that the algorithm stops on the first voxel hit. As the traversal is guaranteed to be front-to-back for each ray, and there is no potential overlap, the first hit is also guaranteed to be the closest hit.

5.4.2 *Octree*

The octree is built using a top-down process. The octree builder takes a list of surface voxels from the voxel rasterizer. The voxels are then sorted in-place by their morton order index. To compute the index, I simply combine the x, y and z coordinates by interleaving the bits. Once the data is sorted, every voxel is already placed at the right offset and the interior nodes have to be created. I use a simple recursive subdivision on the sorted voxel list. The first three bits of every voxel position are used to assign the

```
1  function TRAVERSE(r, t)
2      next = root
3      s_max = r_max
4      loop
5          if ISINTERIOR(next) then
6              node = t[next]
7              if INTERSECT(r, node, t) then
8                  close, far = SORTCHILDREN(r, node)
9                  PUSH(t, r_max, far)
10                 next = close
11                 r_max = t
12             else
13                 next = CLOSECHILD(node, ray)
14         else
15             if ISLEAF(next) then
16                 node = t[next]
17                 if INTERSECTLEAF(node, hit) then
18                     return hit
                                                  ▷ otherwise, empty node
19             if EMPTY(stack) then
20                 r_min = r_max
21                 r_max = s_max
22                 next = root
23             else
24                 r_min, r_max, next = POP(stack)
25             if r_min ≥ r_max then
26                 return miss
```

Algorithm 5.1: kD tree kernel used in the voxel raytracer. Compared to the standard kD tree traversal, this algorithm adds a short-stack with restart.

voxels into children, then one level of the tree is created and the recursion continues with the next three bits.

The recursion stops if the number of voxels is eight or less. As this may create leaf nodes at higher levels of the tree, the voxels are stored with their complete position, allowing them to be freely positioned inside a leaf. Similar to the kD tree, the octree may contain empty leaf nodes. During linearization, a special `null` marker index is emitted for such nodes.

The traversal (see also Algorithm 5.2) computes which interior planes are intersected by the current ray and traverses into the closest child node. If an interior plane is hit, both the current node index and the current ray minimum/maximum is pushed onto the stack. Pushing the current node instead of computing all intersected children and pushing them allows for better usage of the short stack and simplifies the traversal routine. Similar to the bounding volume hierarchy, the traversal stack is popped once a leaf node is reached; if the stack is empty, a restart is performed. Just like in the kD tree, a leaf intersection stops traversal immediately.

The special `null` marker is checked when a leaf is intersected, in this case, the traversal skips the leaf intersection. The logic here is the same as for the kD tree.

The octree does not use level-of-detail during traversal. While simple level-of-detail is a major feature of octrees, I didn't integrate it into the traversal to allow for a fair comparison of the data structures. Instead, as mentioned earlier, I have voxelized the scene at different resolutions.

5.4.3 *Bounding volume hierarchy*

I use a standard bounding volume hierarchy builder with a middle-split heuristic. Compared to an SAH based builder, the middle split builder creates a more balanced tree while providing the same raytracing performance. The bounding volume hierarchy is built top-down by identifying the longest axis first, and then splitting exactly in the middle. This is performed recursively until a leaf-size of eight or less voxels. Unlike the octree and the kD tree, the bounding volume hierarchy has no empty nodes, so a dedicated `null` marker is not required.

During the linearization, the tree is packed to a BVH2 (see also Section 2.5.3) by moving the bounding boxes up to their direct parents. That is, every interior node contains the two bounding boxes of its children. This enables an efficient traversal and allows me to store the leaf nodes without bounding boxes.

The traversal routine (see also Algorithm 5.3) is very similar to the standard bounding volume hierarchy traversal. The main difference stems from the BVH2 merging and

```
1  function TRAVERSE(r, t)                ▷ r has been already intersected with scene bounds
2      next = root
3      offset = (0,0,0)
4      level = t_depth
5      s_max = r_max
6      loop
7          if ISINTERIOR(next) then
8              node = t[next]
9              s = 2^(level−1)
10             c_max = r_max
11             result = INTERSECT(r, node, t)
12             if MULTIPLEHITS(result) then
13                 PUSH(stack, node, c_max, level)
14             octant = CLOSESTCHILD(r, node)
15             next = GETCHILD(node, octant)
16             level = level − 1
17             UPDATEOFFSET(offset, octant, level)
18         else
19             if ISLEAF(next) then
20                 node = t[next]
21                 if INTERSECTLEAF(node, hit) then
22                     return hit
                                                  ▷ otherwise, empty node
23             if EMPTY(stack) then
24                 r_min = r_max
25                 r_max = s_max
26                 next = root
27                 offset = (0,0,0)
28                 level = t_depth
29             else
30                 r_min = r_max
31                 next, r_max, level = POP(stack)
32                 CLEARBITS(offset, level)
33             if r_min ≥ r_max then
34                 return miss
```

Algorithm 5.2: Octree traversal for the voxel raytracer. Compared to the standard octree traversal, a short-stack with restart has been added.

```
1  function TRAVERSE(r,t)                                  ▷ r is the ray, t the tree
2      next = root
3      loop
4          if r_min ≥ r_max then
5              return miss
6          if ISINTERIOR(next) then
7              node = t[next]
8              result = INTERSECT(r, node, t_0, t_1)
9              if BOTHHIT(result) then
10                 close, far = SORTCHILDREN(r, node)
11                 nextMin = max(t_0, t_1)
12                 next=close
13                 PUSH(far)
14             else if FIRSTHIT(result) then
15                 next=FIRSTCHILD(node)
16             else if SecondHit(result) then
17                 next=SECONDCHILD(node)
18         else if ISLEAF(next)) then
19             node = t[next]
20             if INTERSECTLEAF(node, hit) then
21                 return hit
22             if EMPTY(stack) then
23                 r_min = max(r_min, nextMin)
24                 next = root
25             else
26                 next = POP(stack)
```

Algorithm 5.3: BVH traversal used in the voxel raytracer. Compared to the standard BVH traversal, this algorithm is designed to run on a BVH2 and uses a short-stack with restart.

Tree	Ray type	With stack	Without stack
BVH	Primary	5.1	13.0
	Shadow	5.3	21.0
	Soft-shadow	216.5	789.2
	AO	369.9	811.1
kD	Primary	6.2	11.3
	Shadow	8.8	12.9
	Soft-shadow	301.3	476.1.9
	AO	481.5	797.4
Octree	Primary	6.5	10.0
	Shadow	6.5	8.5
	Soft-shadow	306.2	416.9
	AO	420.0	634.7

Table 10: Execution time in milliseconds with and without a short-stack. For the bounding volume hierarchy, a short-stack with 8 entries is used. For the kD and octree the stack size is 4, due to larger stack entries. In every case, total execution time is improved by the addition of a short-stack. I have also measured a corresponding increase in coherency.

the fact that the voxel tree has no overlaps. Instead of intersecting with the bounding box when a node is entered, the bounds are only checked when an interior node is visited. If both are hit, the traversal continues into the closer one and the further away node is pushed onto a short traversal stack. Once a leaf node has been reached, the stack is popped; if the stack is empty, a full restart is performed. Unlike for triangle raytracing, where overlapping nodes have to be resolved using auxiliary storage, I can perform a restart once the stack becomes empty. Just as for the octree and kD tree, a leaf intersection immediately stops traversal.

5.4.4 *Short stack and restart traversal*

For all traversal kernels, I use a short stack combined with restarts. The stack is placed in local memory and pre-allocated. If the stack runs empty, a restart is performed. For voxel rendering, this change is possible as there are no overlapping nodes in any of the acceleration structures which would require additional storage to resolve.

Adding a short stack is beneficial for all algorithms (see also Table 10). The short stack is implemented as a circular buffer. Pushing onto the stack always succeed. If

the stack is full, the oldest pushed element will be overwritten. During pop, either the last pushed item is retrieved, or, if the stack is empty, an error is reported which triggers a traversal restart.

Tree		BVH				kD				Octree			
Scene	Res	# I	Size	# L	Size	# I	Size	# L	Size	# I	Size	# L	Size
Atlas	1024	186028	6	186029	1	316342	4	193637	1	105406	3	367221	3
	2048	768075	26	768076	6	1302189	15	798005	6	428138	13	1486192	11
	4096	3205277	110	3205278	24	5384670	62	3314389	25	1741729	53	6057483	46
	8192	13394650	460	13394651	102	22377048	256	13780426	105	7071909	216	24618817	188
	16384	55817473	1916	55817474	426	92735073	1061	57171972	436	28704198	876	99822901	762
Conference	1024	493343	17	493344	4	611691	7	537589	4	236386	7	772894	6
	2048	2012348	69	2012349	15	2517517	29	2201666	17	995400	30	3263460	25
	4096	8060684	277	8060685	61	10201124	117	8863479	68	4186866	128	13288870	101
	8192	32327634	1110	32327635	247	42111409	482	35642710	272	17187681	525	53408132	407
David	1024	101587	3	101588	1	171112	2	105765	1	54047	2	188035	1
	2048	409964	14	409965	3	695008	8	425792	3	218576	7	756084	6
	4096	1662811	57	1662812	13	2823240	32	1720653	13	882934	27	3043292	23
	8192	6820183	234	6820184	52	11573444	132	7031430	54	3575402	109	12327749	94
	16384	28093398	965	28093399	214	47657376	545	28964514	221	14489447	442	50013999	382
San Miguel	1024	455873	16	455874	3	552202	6	457005	3	193407	6	662241	5
	2048	1850685	64	1850686	14	2134505	24	1858228	14	843955	26	2699159	21
	4096	7474030	257	7474031	57	8300898	95	7512657	57	3509074	107	10968532	84
	8192	30083854	1033	30083855	230	32922155	377	30320213	231	14337661	438	44446093	339

Table 11: Sizes of the acceleration structures for the various test scenes. # I is the number of interior nodes, # L the number of leaf nodes, size the size in MiB for the node type, and Res the resolution of the voxelization.

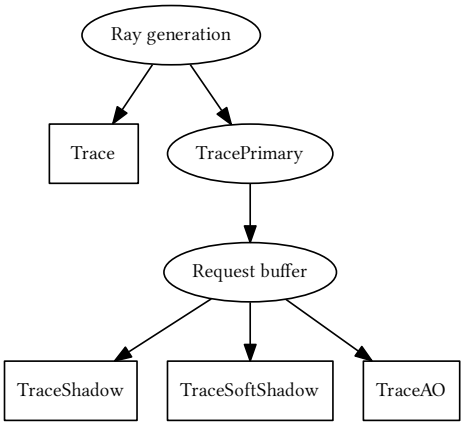


Figure 44: The rendering pipeline for the analysis. Boxes indicate kernels which are instrumented. Two rendering paths are provided in order to eliminate the primary ray-tracing impact for the secondary ray measurements.

5.5 RESULTS & ANALYSIS

I have measured the performance on an AMD FirePro W9100 GPU with 16 GiB of memory. The host machine was a Dual-Xeon X5650 machine, with 12 cores/24 threads total and 24 GiB of memory. The tests were run using Windows 8.1 and Direct3D 11.1.

For each test scenario, the test runner starts the test application from scratch and renders the test scenario 32 times. Profiling results are temporarily recorded in memory and dumped to the console at the application end. The test runner then gathers the results, stores them, terminates the application and starts a new instance for the next test. The execution of the test kernel is instrumented using the `GPUPerfAPI`. Due to limitations on how many counters can be written in a single pass, the test kernel had to be executed multiple times to gather all performance counters.

The analysis is split into multiple passes (see Figure 44). For primary rays, the ray-generation is part of the kernel. For secondary rays, a modified primary ray kernel is executed which records the hits for each ray and writes them into a buffer. In the second pass, each thread reads one request from this buffer and traces the secondary rays. In all cases, the rays are grouped in screen-space tiles to maximize coherency.

In some cases, the total number of executed wavefronts measured is less than the number of requested wavefronts, indicating that the counters have not run for the complete kernel execution. As the number of wavefronts is known and constant, any

result with a different number is rejected. For the remaining counter results, I have used one representative measurement.

As the instrumented runs require multiple passes, the execution time is measured in a separate run. In this case, the total frame time is measured, which is more stable than the actual kernel execution time. The overhead is minimal, as it solely consists of the drawing of a single textured quad to the framebuffer. For the execution time, I have taken the median value as the representative value.

5.5.1 *Measurement results*

I have tested four different scenes: `Atlas`, `David`, `Conference` and `San Miguel`. Atlas is a 3D scan with a lot of surface detail as well as noise. Unlike the other scenes, Atlas exhibits many tiny holes in the surface, which complicates traversal. David is a very clean 3D scan without any holes, allowing nearly all rays to traverse once to the leaf level and terminate.

San Miguel and Conference are in-door scenes. In both cases, the camera is placed within the boundaries of the data set. This leads to a completely different traversal behavior compared to the 3D scans, where the camera is positioned outside of the object. In the scenes where the camera is outside, the traversal starts from the tree root and descends once to the leaf node level. In the interior scenes, the traversal has to descend immediately to a very low level in the tree and then continues traversal near the leaf nodes. This puts additional pressure on the tree traversal and the back-track performance.

Tree		BVH				kD				Octree			
Scene	Res	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO
Atlas	1024	59.7	52.6	47.4	33.5	40.7	30.9	28.0	16.2	52.4	40.4	37.5	22.9
	2048	50.5	42.9	40.5	32.2	32.2	22.5	21.3	14.5	44.5	31.3	29.9	20.5
	4096	44.3	37.0	36.0	31.5	27.1	17.4	17.0	13.4	39.4	24.7	24.2	19.0
	8192	41.0	33.5	33.0	30.9	24.5	14.4	14.2	12.5	35.8	20.4	20.2	17.8
	16384	39.2	31.3	30.9	30.4	23.1	12.4	12.4	11.9	33.4	17.5	17.5	16.8
David	1024	57.9	58.5	58.2	35.6	41.4	39.2	39.9	18.4	53.3	49.2	49.5	25.2
	2048	50.0	50.3	50.6	34.6	33.5	29.4	30.2	16.5	46.8	39.7	40.4	23.0
	4096	45.0	45.1	45.6	33.8	28.7	23.2	23.9	15.0	41.9	32.6	33.4	21.1
	8192	42.1	42.2	42.7	33.2	26.0	19.5	20.1	13.9	38.4	27.7	28.4	19.6
	16384	40.3	40.5	40.9	32.7	24.5	17.1	17.6	13.1	35.9	24.2	24.9	18.5
Conference	1024	70.0	56.9	55.5	35.8	43.7	39.3	40.3	19.1	48.8	44.0	43.6	21.8
	2048	67.8	50.3	49.9	35.9	43.6	34.3	35.4	17.9	47.3	38.6	38.4	20.9
	4096	66.6	49.9	49.6	36.2	41.9	31.1	32.1	17.0	46.7	36.0	35.5	20.7
	8192	66.2	49.6	49.2	36.4	40.5	28.8	29.6	16.3	46.0	33.7	33.0	20.3
San Miguel	1024	65.2	52.7	43.8	27.2	46.1	34.0	28.1	14.1	51.3	40.4	34.9	18.7
	2048	56.5	43.7	37.4	26.3	36.0	24.6	21.0	12.1	42.4	31.9	27.9	16.9
	4096	50.2	37.4	33.9	26.1	29.4	18.7	16.8	10.7	36.7	26.0	23.4	15.6
	8192	46.8	34.9	32.7	26.5	26.4	15.6	14.5	10.0	33.3	22.0	20.5	14.8
Average		53.3	45.0	43.2	32.2	33.9	25.1	24.6	14.6	43.0	32.2	31.3	19.7

Table 12: Measured coherency across various test scenes and ray types. Res is the resolution of the scene, `Prm` are primary rays, `Shd` shadow rays, `Soft` are soft-shadows and `AO` are ambient occlusion rays. For soft-shadows and ambient occlusion, 36× as many rays have been casted as for the primary and shadow rays tests.

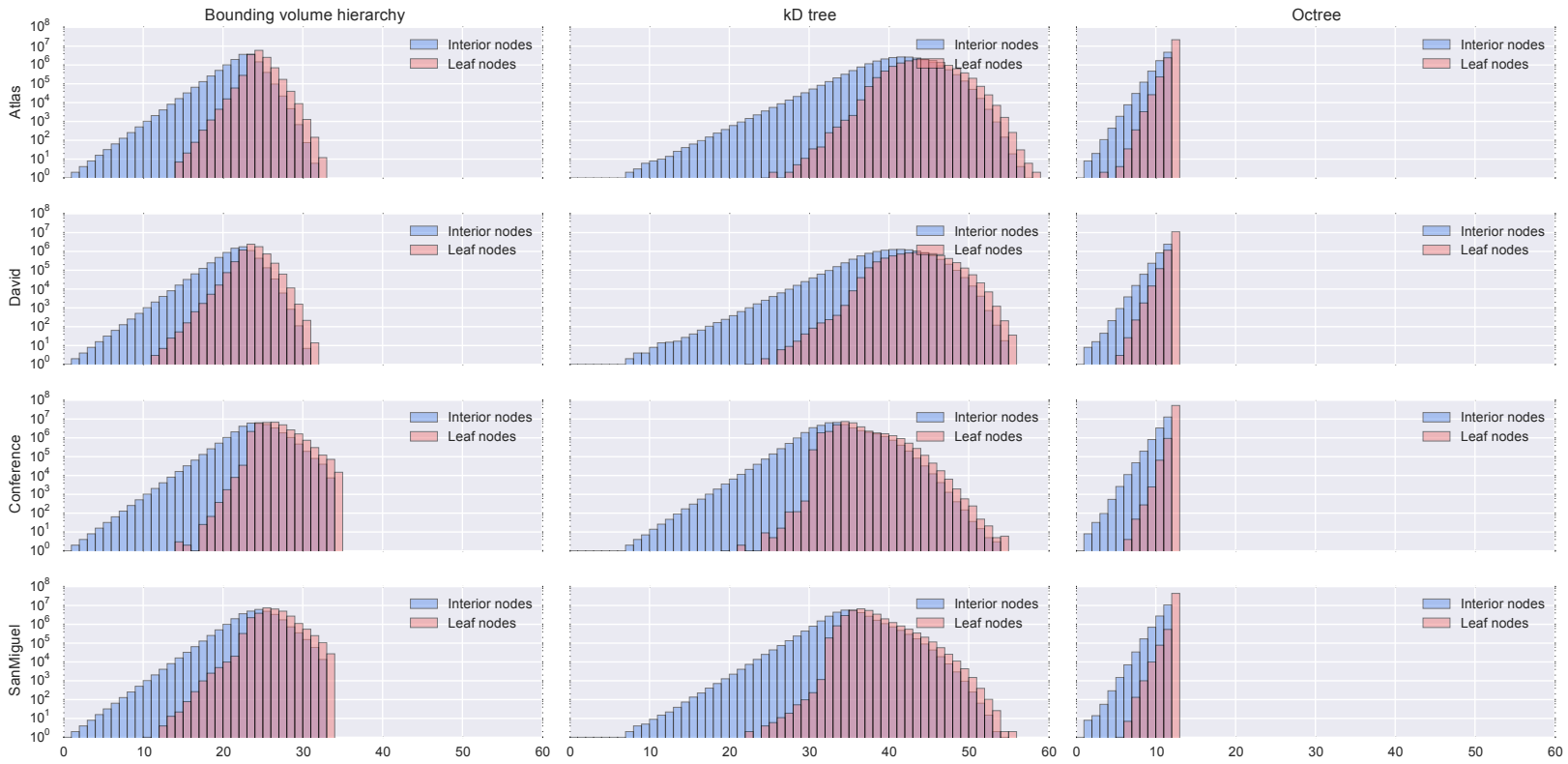


Figure 45: Histograms of the tree depth and number of leaves/interior nodes at a particular depth for the data sets at 8192^3 resolution. The y-axis is in logarithmic scale. The first few levels of the kD tree bound the scene and have one only interior child. Afterwards, the split count is comparable to the bounding volume hierarchy. The octree has $8\times$ more nodes at each level and produces a much more shallow tree.

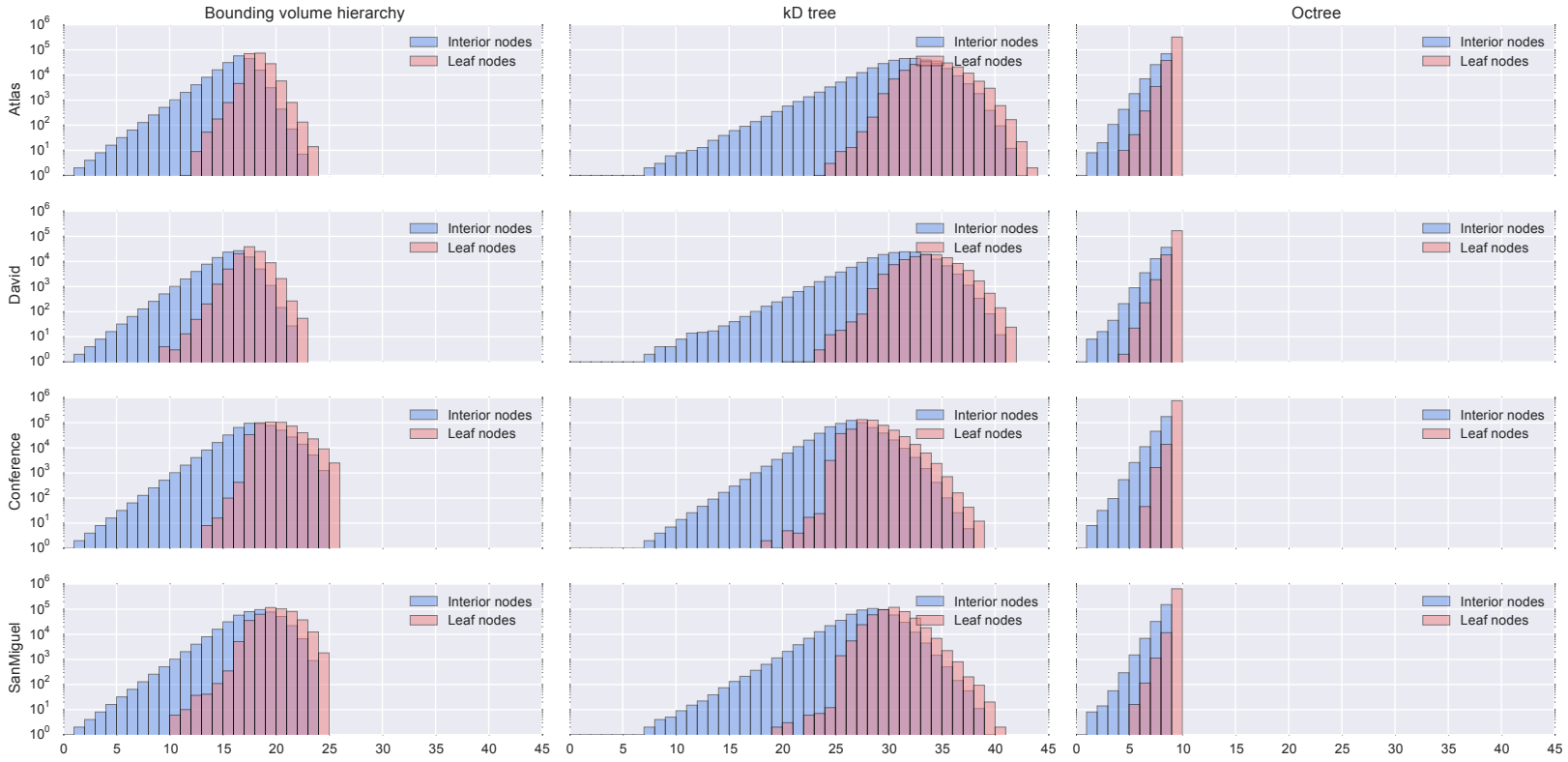


Figure 46: Histograms for the trees at 1024^3 resolution.

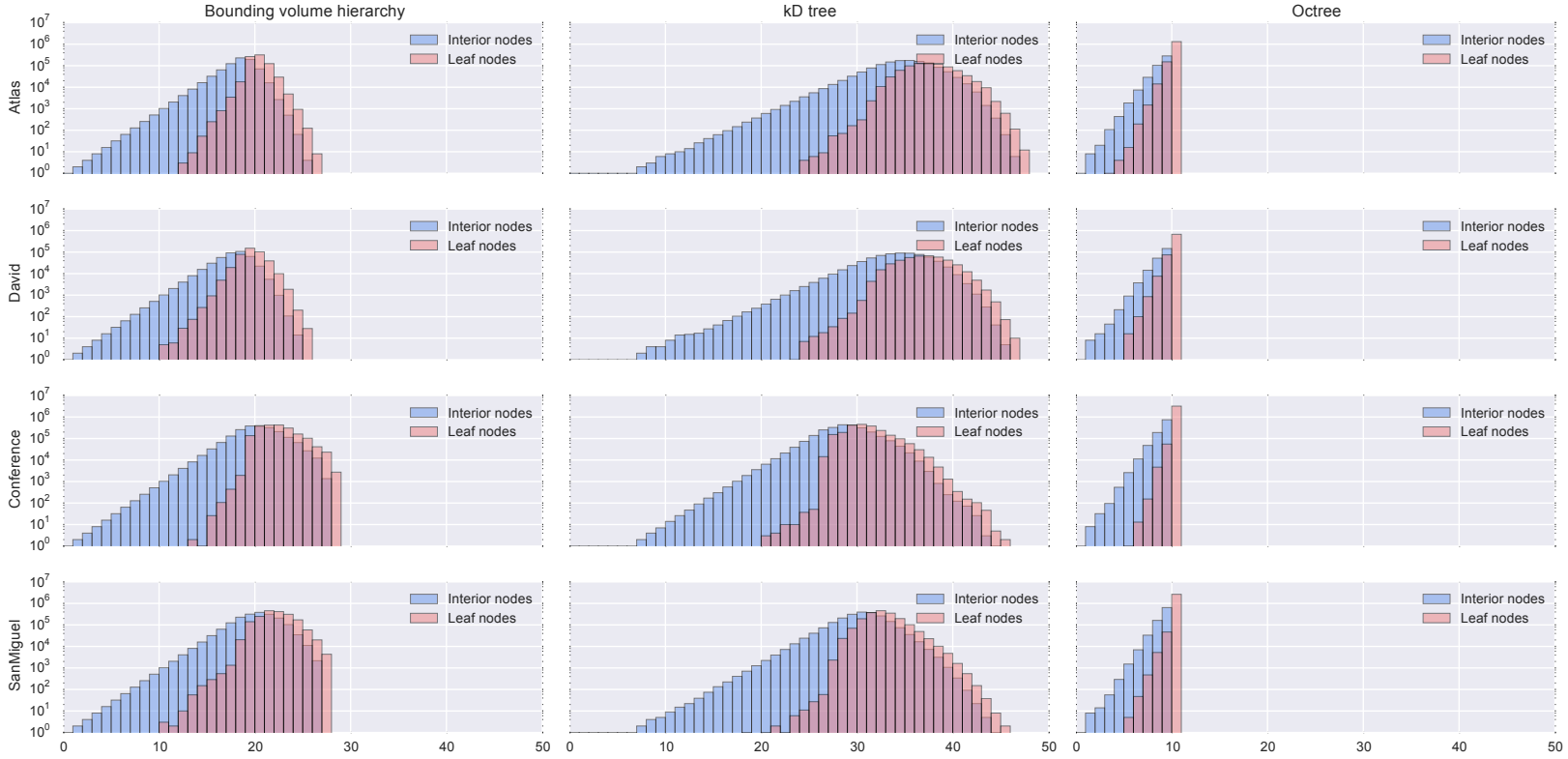


Figure 47: Histograms for the trees at 2048^3 resolution.

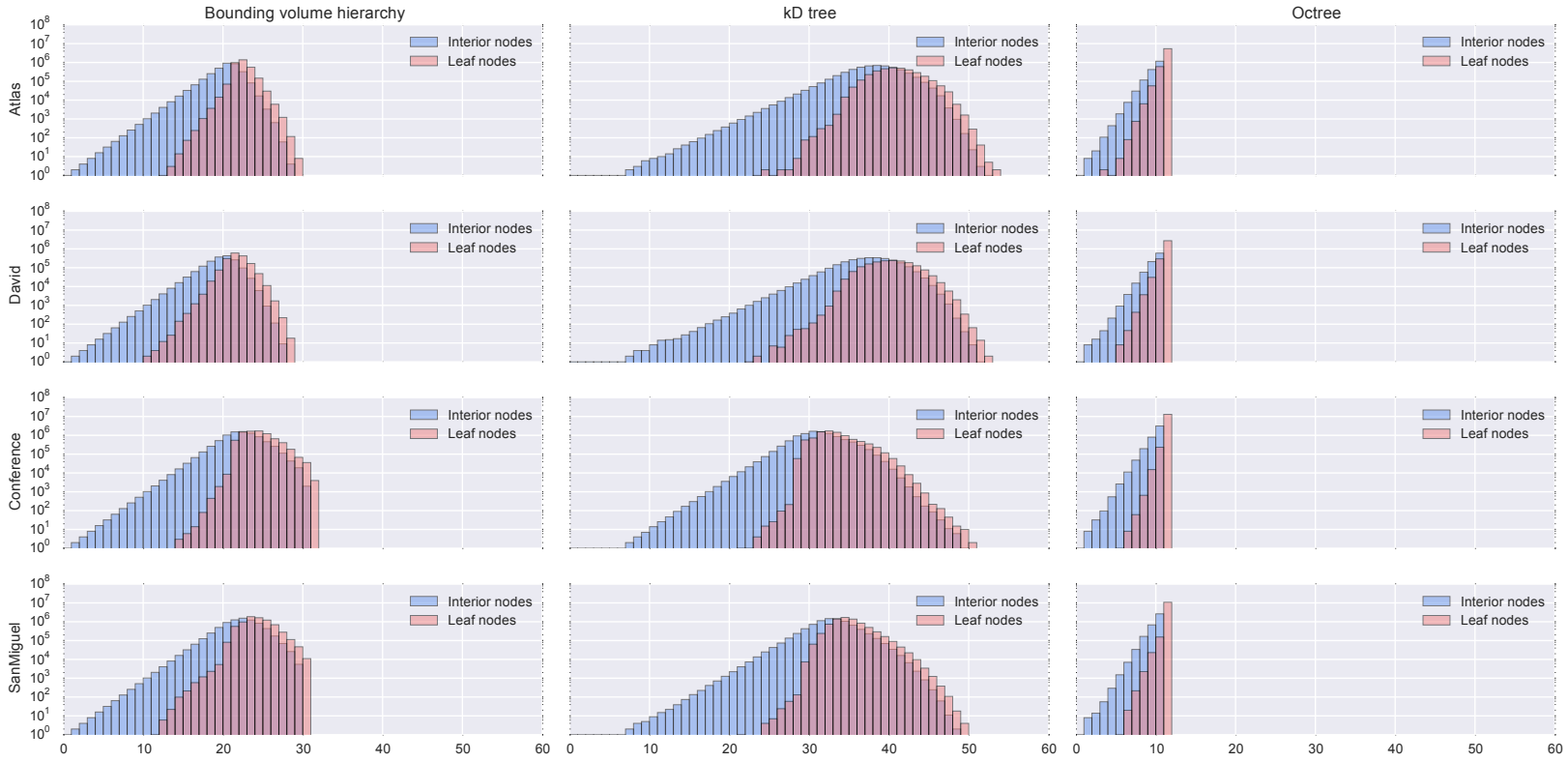


Figure 48: Histograms for the trees at 4096^3 resolution.

As can be seen in Table 12 and Table 13, on average, the bounding volume hierarchy has the highest coherency across all test scenes and the best traversal performance. The coherency of the bounding volume hierarchy is high even for very large trees, unlike the kD tree and the octree. It also degrades less for incoherent rays, maintaining at least 30% efficiency for the largest data sets, or, translated to the 64-wide vector unit, approximately 19 active vector lanes.

Coherency is also related to the tree depth. Very deep trees are likely to exhibit lower coherency, as long execution chains can be created. I have measured the average depth at which leafs nodes are present (see also Figure 49 and Figure 45). Compared to the bounding volume hierarchy, the kD tree is between 38% to 92% deeper. The bounding volume hierarchy is 120% to 170% deeper than the octree. Notice that the first six levels of the kD tree, corresponding to 13%-20% of the total tree depth, are the global bounds of the scene and thus highly coherent. I cannot conclude that a higher tree depth has significant impact on the coherency, as the kD tree exhibits similar coherency to the octree, despite much deeper trees. For instance, in the `Conference` scene, the kD tree reaches 40% coherency for primary rays, compared to 46% for the octree. For comparison, the bounding volume hierarchy, with an average depth right between the kD and the octree, achieves 66% coherency.

We can also see that primary ray coherency is up to twice as high as for secondary, incoherent rays. This result is similar to the observations made in [AL09], however, in their work, a 32-wide architecture was used which is half as wide as the hardware used in my test.

Figure 49: The average depth and standard deviation for the different trees. The octree has near zero variance at the leaf, which are mostly placed at the last or second last level. The average variance of the interior and leaf depth for kD tree is approximately twice that of the bounding volume hierarchy, which is again twice as high as that of the octree. In all cases, the variance is nearly independent of the tree size.

Tree		BVH				kD				Octree			
Scene	Res	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO	Prm	Shd	Soft	AO
Atlas	1024	3.0	4.1	168.9	190.8	4.4	6.1	219.8	271.3	3.7	4.3	201.9	216.3
	2048	4.0	5.9	233.1	244.7	5.6	10.1	336.3	378.1	5.0	6.6	300.4	291.0
	4096	5.3	7.9	315.7	295.6	7.4	14.7	501.3	490.6	6.2	9.7	434.3	369.6
	8192	6.5	9.9	399.2	348.3	9.0	20.8	712.0	620.9	7.8	13.6	601.4	457.0
	16384	7.5	12.0	480.7	447.8	10.4	28.0	949.9	873.6	9.0	18.2	795.2	551.8
David	1024	2.3	1.1	43.7	108.6	3.1	1.7	55.0	144.8	3.4	1.4	61.4	132.0
	2048	3.2	1.7	63.4	140.0	4.3	2.9	91.4	200.7	4.4	2.0	89.3	172.2
	4096	4.0	2.3	88.1	172.0	5.6	4.5	143.1	268.6	5.4	3.0	128.1	218.7
	8192	4.9	3.1	115.6	203.5	7.2	6.5	205.5	342.7	6.7	4.2	177.2	271.8
	16384	5.7	3.9	142.3	236.7	8.2	8.8	282.8	427.0	7.8	5.8	236.0	328.4
Conference	1024	3.7	2.3	96.0	331.2	2.8	2.0	69.3	274.6	5.5	2.9	138.0	397.2
	2048	4.3	2.9	125.0	364.1	3.4	2.6	90.8	322.5	6.2	4.0	185.6	489.5
	4096	4.9	3.4	143.1	393.1	3.9	3.1	110.0	366.5	6.5	4.8	227.1	545.9
	8192	5.5	3.8	158.3	423.3	4.4	3.9	132.9	411.4	6.9	6.0	274.5	602.0
San Miguel	1024	4.5	4.5	209.5	516.5	5.0	5.0	199.5	499.0	4.7	3.8	199.1	370.8
	2048	6.1	6.4	292.6	638.6	7.1	8.3	321.9	714.4	6.7	6.1	314.1	532.4
	4096	7.8	8.5	378.0	736.1	9.1	12.5	469.1	929.6	9.1	8.9	448.4	714.3
	8192	9.3	10.9	443.1	866.9	11.2	17.0	621.6	1129.8	11.4	12.5	611.6	900.0
Average		5.1	5.3	216.5	369.9	6.2	8.8	306.2	481.5	6.5	6.5	301.3	420.0

Table 13: Measured execution time in milliseconds. The captions are the same as in Table 12. Notice that `Soft` and `AO` trace 36× more rays.

Tree	Ray type	Cache hit rate (%)
BVH	Primary	59.0
	Shadow	49.8
	Soft-shadow	72.3
	AO	56.2
kD	Primary	64.2
	Shadow	51.5
	Soft-shadow	73.4
	AO	69.0
Octree	Primary	59.8
	Shadow	47.8
	Soft-shadow	75.1
	AO	66.7

Table 14: Average level 2 cache hit rates for the different scenes, as measured on the AMD FirePro W9100 with 1 MiB of L2 cache.

Besides the coherency, I have also measured the execution time to understand how the coherency translated into actual ray-tracing performance. As can be seen in Table 13, the coherency is a good indicator for the achievable performance. This is a strong hint that all of the tested traversal algorithms are limited by execution speed and not by memory bandwidth. The hardware counters also indicate that the kernels never block on memory accesses. For very low coherency values, the execution time increases non-linearly – I assume that we can see the effects of few, very long running rays in this case.

I have also measured the cache usage (see also Table 14) to identify the impact of large nodes in the acceleration structure. The bounding volume hierarchy has the biggest interior nodes, consisting of two bounding boxes, two pointers and the split dimension. In total, an interior node requires 33 byte of data (padded to 36 byte). A kD interior node stores the split position, dimension and two pointers and requires 11 bytes (padded to 12). Finally, an octree node stores 8 child pointers each and requires 32 bytes. The leaf nodes occupy 8 bytes for all acceleration structures and consist of the start primitive offset and the primitive count. The complete tree sizes can be seen in Table 11. Even though the kD tree has roughly twice as many interior nodes as the bounding volume hierarchy, it is the smallest data structure due to the compact nodes. The octree is approximately twice as big as the kD tree. The bounding volume

hierarchy, which has similarly sized nodes as the octree, but requires more nodes, is the largest data structure.

As expected, the kD tree shows the best cache efficiency due, yet it does not translate into better performance. Interestingly, shadow rays exhibit by far the worst cache efficiency, even worse than primary rays. The reason for this is that shadow rays all start at different positions in the tree, unlike primary rays, which traverse through the same part of the tree for the first few intersection steps. Soft-shadows and ambient occlusion show very good cache usage as many rays originate from the similar points in the tree and only traverse small regions of the tree.

5.5.2 *Analysis*

My analysis shows that the data structure has a significant impact on the traversal coherency. Interestingly, the data structure with the most complex spatial partitioning performs best from both coherency and performance points of view. This can be explained by investigating how precise the spatial partitioning is. In general, a single step in the bounding volume hierarchy traversal performs 12 plane intersections (6 for each child); a single step in the octree performs 3 plane intersections and the kD tree only plane intersection. As can be seen, the probability that a bounding volume hierarchy node will result in a push is very low compared to both the octree and kD tree.

We can get a better intuition into this problem by looking at the probabilities of each traversal step. For example, the kD tree traversal has only two outcomes: Either the split plane is intersected or not. The chance that some SIMD lanes take one path and others take the other path is thus very high, leading to overall low coherency, as the execution tree will have many branches. On the other hand, the bounding volume hierarchy has a very low probability that a node is hit. It is very likely that all paths will hit only one node and execute the same code path. The octree is in-between – initially, it would seem that its behavior should be equal to a kD tree, but the fact that the ray is shortened by potentially up to three planes per step significantly reduces the chance of an intersection. The key observation is that a long and complicated intersection test does not put a data structure at a disadvantage, coherency wise. As long as the test can be formulated in a branch-free manner, which is easily done for a bounding-box intersection, the resulting traversal will benefit from higher coherency due to the improved selectivity of the test.

Another important observation is that even the most incoherent acceleration structure never dips below 12% coherency in the worst case – that is, 8 units of the 64-wide

vector unit can be always filled. This strongly suggests that for CPUs, which only support 4 & 8-wide SIMD, efficient re-packing of rays would allow to reach near perfect utilization of this comparatively narrow vector units.

6

CONCLUSION AND FUTURE WORK

In this thesis I have described a voxel-based rendering pipeline for high-resolution geometry. After covering the fundamental concepts, I have presented the two core ingredients of my approach: A highly scalable pre-process, which includes voxelization and simplification, and a corresponding renderer. Due to its focus on rasterization, the renderer allows for high-quality viewing, low memory usage and run-time modifications. Finally, I have analyzed the performance of GPU voxel raytracing in detail on modern hardware to gain insight into which algorithm characteristics are most important for performance.

The pre-processing can be further improved in multiple ways. First of all, an out-of-core path for very high resolution output would be desirable. This can be implemented by storing intermediate results and replaying the voxelization process. Second, the voxelizer currently always uses the input triangle mesh. It would useful to implement support for other surface types, in particular subdivision surfaces or NURBS with displacement maps. The evaluation of the limit surface could be integrated into the voxelization process, avoiding the need for very finely tessellated input meshes. Finally, the preprocessor currently works exclusively on the CPU. Especially with an out-of-core pre-pass, it should be possible to implement the "leaf" rasterization stage on the GPU for even higher performance.

The rasterization algorithm described in this thesis can be also optimized to improve performance. The usage of the geometry shader for expansion is not well supported by current graphics hardware and is one of the main bottlenecks of the presented algorithm. It is likely that a two-pass algorithm, which first decompresses into temporary memory and then renders without any expansion will result in higher triangle throughput. A key insight here is that the output topology for a voxel is either one quad consisting of two triangles, if only one face is visible, or a triangle strip consisting of four triangles, for both two or three visible faces. This makes it possible to efficiently identify the cases and handle them using only two temporary buffers and draw calls.

The rendering quality can be also improved for iso-surfaces by integrating hybrid rendering for those as well. In the current implementation, hybrid rendering for iso-surfaces makes it necessary to extract an iso-surface as geometry in a preprocess. For best quality, the renderer should be modified to use direct volume ray-tracing in this case.

On the ray-tracing side, I have identified that a voxel-optimized bounding volume hierarchies results in superior performance compared to the commonly used octrees if no level-of-detail is used. An interesting question in this context is how level-of-detail can be integrated into a bounding volume hierarchy. As the detail level is just another dimension, it seems feasible to build a 4D tree which directly integrates level-of-detail.

I have also briefly explored the effect of data structures on ray-tracing coherency. For bounding volume hierarchies and kD trees, it seems possible that the builder can actually exploit this to build coherency-optimized acceleration structures. This will requires changes the cost heuristics which take the resulting "tree coherency" into account.

In conclusion, I have presented a very fast pipeline for the rendering of very large meshes. By focusing on the data flow from the input mesh to the final renderer, I was able to exploit optimization opportunities and different stages along the pipeline. In the end, my approach makes it possible to quickly investigate and explore even the most complex meshes on commodity machines.

BIBLIOGRAPHY

[Abr97] Michael Abrash. *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash.* Coriolis Group Books, Scottsdale, AZ, USA, 10th edition, 1997.

[AK10] Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics 2010*, pages 113–122, 2010.

[AKL13] Timo Aila, Tero Karras, and Samuli Laine. On quality metrics of bounding volume hierarchies. In *Proc. High-Performance Graphics*, 2013.

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[AMA05] Tomas Akenine-Möller and Timo Aila. Conservative and tiled rasterization using a modified triangle setup. *Journal of Graphics Tools*, 10(3):1–8, 2005.

[AMD13] AMD. *AMD Accelerated Parallel Processing*, 2013.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition.* A. K. Peters, Ltd., Natick, MA, USA, 2008.

[App68] Arthur Appel. Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[BAM14] Rasmus Barringer and Tomas Akenine-Möller. Dynamic ray stream traversal. *ACM Trans. Graph.*, 33(4):151:1–151:9, July 2014.

[BEL+07] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007*, GI '07, pages 177–184, New York, NY, USA, 2007. ACM.

[Ben75] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[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 5th High-Performance Graphics Conference*, HPG '13, pages 27–32, New York, NY, USA, 2013. ACM.

[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 *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. to appear.

[COK95] Daniel Cohen-Or and Arie Kaufman. Fundamentals of surface voxelization. *Graph. Models Image Process.*, 57(6):453–461, November 1995.

[COK97] Daniel Cohen-Or and Arie Kaufman. 3d line voxelization and connectivity control. *IEEE Comput. Graph. Appl.*, 17(6):80–87, November 1997.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 137–145, New York, NY, USA, 1984. ACM.

[CRS13] Matthäus G. Chajdas, Matthias Reitinger, and Jan Sommer. Vota, 2013.

[DHK08] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays.

In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR'08, pages 1225–1233, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[ED06] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 71–78, New York, NY, USA, 2006. ACM.

[ED08] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI '08, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[FBH+10] Kayvon Fatahalian, Solomon Boulos, James Hegarty, Kurt Akeley, William R. Mark, Henry Moreton, and Pat Hanrahan. Reducing Shading on GPUs Using Quad-fragment Merging. *ACM Trans. Graph.*, 29(4):67:1–67:8, July 2010.

[FGH+85] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes. *SIGGRAPH Comput. Graph.*, 19(3):111–120, July 1985.

[FH08] Kayvon Fatahalian and Mike Houston. A Closer Look at GPUs. *Commun. ACM*, 51(10):50–57, October 2008.

[FPE+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 79–88, New York, NY, USA, 1989. ACM.

[FS05] Tim Foley and Jeremy Sugerman. KD-tree Acceleration Structures for a GPU Raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, pages 15–22, New York, NY, USA, 2005. ACM.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Gie11] Fabian Giesen. A trip down the graphics pipeline. 2011.

[GPSS07] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007. `http://www.mpi-inf.mpg.de/~guenther/BVHonGPU/index.html`.

[Gre96] Ned Greene. Hierarchical polygon tiling with coverage masks. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 65–74, New York, NY, USA, 1996. ACM.

[Hen07] Justin Hensley. Close to the Metal, 2007.

[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.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive K-d Tree GPU Raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[HYFK98] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, pages 119–126, New York, NY, USA, 1998. ACM.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[JKSH13] Alec Jacobson, Ladislav Kavan, , and Olga Sorkine-Hornung. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH)*, 32(4):33:1–33:12, 2013.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

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

[Khr12] Khronos. *The OpenCL specification, version 1.2, revision 19*. Khronos OpenCL Working Group, 2012.

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

[KSKAC02] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. A simple and robust mutation strategy for the metropolis light transport algorithm. In *Computer Graphics Forum*, volume 21, pages 531–540. Wiley Online Library, 2002.

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

[LB03] Adriano Lopes and Ken Brodlie. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):16–29, January 2003.

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[Liu77] Hsun K. Liu. Two- and three-dimensional boundary detection. *Computer Graphics and Image Processing*, 6(2):123 – 134, 1977.

[LK11a] Samuli Laine and Tero Karras. Efficient Sparse Voxel Octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17:1048–1059, 2011.

[LK11b] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 79–88, New York, NY, USA, 2011. ACM.

[LLVT03] Thomas Lewiner, Helio Lopes, Antonio Wilson Vieira, and Geovan Tavares. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of Graphics Tools*, 8:2003, 2003.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *Proceedings of ACM SIGGRAPH 2000*, pages 131–144, July 2000.

[LW93] Eric P Lafortune and Yves D Willems. Bi-directional path tracing. 1993.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, May 1990.

[Mea82] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[Mie09] Christopher Mielack. Isosurfaces. 11 2009.

[MKWF04] Gerd Marmitt, Andreas Kleer, Ingo Wald, and Heiko Friedrich. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *in Proceedings of Vision, Modeling, and Visualization (VMV*, pages 429–435, 2004.

[MM00] Joel McCormack and Robert McNamara. Tiled polygon traversal using half-plane edge functions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00, pages 15–21, New York, NY, USA, 2000. ACM.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[MS11] Josiah Manson and Scott Schaefer. Wavelet rasterization. *Computer Graphics Forum (Proceedings of Eurographics)*, 30(2):395–404, 2011.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 3 2008.

[NCKG00] László Neumann, Balázs Csébfalvi, Andreas König, and Eduard Gröller. Gradient estimation in volume data using 4d linear regression. *Computer Graphics Forum*, 19(3):351–358, 2000.

[NH91] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Proceedings of the 2Nd Conference on Visualization '91*, VIS '91, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[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.

[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.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. `http://www.mpi-inf.mpg.de/~guenther/StacklessGPURT/index.html`.

[Pin88] Juan Pineda. A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, 22(4):17–20, June 1988.

[RB13] Randall Rauwendaal and Mike Bailey. Hybrid computational voxelization using the graphics pipeline. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):15–37, March 2013.

[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.

[RUL00] J Revelles, Carlos Urena, and Miguel Lastra. An efficient parametric algorithm for octree traversal. In *WSCG*, 2000.

[RW80] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph.*, 14(3):110–116, July 1980.

[Sam89] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics*, 13:445–460, 1989.

[Sam90] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[SS10] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 179:1–179:10, New York, NY, USA, 2010. ACM.

[SSKLK13] Dave Shreiner, Graham Sellers, John M Kessenich, and Bill M Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley Professional, 2013.

[ST86] Roger W. Swanson and Larry J. Thayer. A fast shaded-polygon renderer. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 95–102, New York, NY, USA, 1986. ACM.

[VG95] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 419–428, New York, NY, USA, 1995. ACM.

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis Light Transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70, 2006.

[Whi80] Turner Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, June 1980.

[WIK+06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, pages 485–493, 2006. http://doi.acm.org/10.1145/1141911.1141913.

[WMS06] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for hardware accelerated ray tracing of dynamic scenes. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 67–77, New York, NY, USA, 2006. ACM. http://doi.acm.org/10.1145/1283900.1283912.

[Woo04] Sven Woop. A Ray Tracing Hardware Architecture for Dynamic Scenes. Technical report, Saarland University, 2004.

[WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A Ray Tracing Solution for Diffuse Interreflection. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, pages 85–92, New York, NY, USA, 1988. ACM.

[WWB+14] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 33(4):143:1–143:8, July 2014.

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