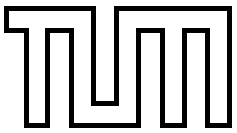


TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

LEHRSTUHL FÜR COMPUTER GRAPHIK UND VISUALISIERUNG

Interactive Sample-Based Rendering of High-Resolution Data Sets

Florian Reichl

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. D. Cremers
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. R. Westermann
	2.	Univ.-Prof. Dr.-Ing. M. Teschner, Albert-Ludwigs-Universität Freiburg

Die Dissertation wurde am 03.11.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 07.01.2015 angenommen.

To my family and friends.

Abstract

Visual feedback is an important tool in the analysis and verification of data in a wide range of scientific domains. Especially interactive 3D visualization methods heavily facilitate the exploration of data sets, as they provide immediate feedback to the user and allow for a fast and goal-oriented analysis by domain experts. With graphics processing units (GPUs) having developed into fully programmable, highly parallel data processing hardware, they have become a prime candidate for accelerating these tasks.

The last years have shown a rapid increase in the amount of data generated and handled in a scientific context. One example is the field of fluid simulation, where data sets as large as several hundred giga- or even terabytes are not uncommon, which exceeds the available memory of even the latest GPUs by several orders of magnitude. Data reduction is thus of utmost importance and is typically achieved by techniques such as level-of-detail. However, the process of building coarser representations is highly dependent on the application and data topology. For triangle-based data, which is a common choice for GPU-based rendering, this process is time-consuming, automatic methods are difficult to control and must rely on a very conservative approximation if a maximum error should be guaranteed.

Resampling the acquired data into a regular 3D sampling grid can simplify this process: Levels of detail can easily be constructed in a bottom-up manner, where coarser representations are created by spatially filtering the data present in the finer levels. Combined with a hierarchical scene representation, empty space can be removed and view-based culling can be performed at a fine-granular level. Interactive rendering with additional data reduction using occlusion culling is then performed by GPU ray-casting. However, a sufficient sampling resolution must be chosen to preserve geometric scene details, which often significantly increases the total memory requirements even though the currently visible working set is reduced for a single frame. Applying—possibly lossy—data compression is a typical way to lower the requirements in storage memory, yet the resulting representation can often not be directly used to render the data, making many compression schemes unsuitable for reducing the working set as well as the amount of data stored in GPU caches.

This thesis introduces a GPU-driven rendering system for the interactive visualization of high-resolution data sets. At its core are novel, memory-efficient sample-based data structures that facilitate direct rendering from a compact representation and, in this way, significantly reduce the memory requirements both at runtime and in terms of storage. In addition, they provide substantial decreases in rendering times compared to existing techniques. The system's scalability is demonstrated in context of massive particle-based simulations and scanned surface data, both of which are rendered at interactive rates on a single desktop PC. Furthermore, its use in dynamic scenarios such as rendering varying isosurfaces from scalar fields and online reconstruction using hand-held depth sensors is presented.

Zusammenfassung

Visuelles Feedback stellt ein wichtiges Werkzeug zur Analyse und Verifikation von Daten in einer Vielzahl von wissenschaftlichen Disziplinen dar. Besonders interaktive 3-dimensionale Visualisierungsmethoden unterstützen die Erkundung solcher Datensätze, da sie dem Anwender unmittelbares Feedback bieten und somit eine schnelle und zielgerichtete Analyse durch Experten erlauben. Seit ihrer Entwicklung hin zu frei programmierbarer, hochparalleler Hardware zur Datenverarbeitung bieten sich Grafikprozessoren (GPUs) mehr und mehr dazu an, diese Anwendungen zu beschleunigen. In den letzten Jahren ist ein stetiger Anstieg in den generierten und zu verarbeitenden Datenmengen in verschiedenen wissenschaftlichen Bereichen zu verzeichnen. Ein Beispiel hierfür sind hochauflösende Fluidsimulationen, die Speicher in der Größenordnung mehrerer hundert Giga- bis hin zu Terabytes belegen können, was den verfügbaren Speicher der aktuellsten GPUs um weit mehr als das Zehnfache übersteigt. Reduktion des Datenvolumens ist daher von entscheidender Bedeutung und typischerweise durch Techniken wie dem Generieren mehrerer Detailstufen ("level of detail") erreicht. Dieser Generierungsprozess ist jedoch stark von der Anwendung und der Topologie der Eingabedaten abhängig. Für Dreiecksnetze, die üblicherweise im Kontext des GPU-basierten Renderings verwendet werden, ist er zeitaufwändig. Vollautomatische Techniken sind zudem schwer zu kontrollieren und sind zu einer sehr konservativen Approximation der Originaldaten gezwungen, falls eine obere Fehlerschranke für die entstehenden Detailstufen garantiert werden soll.

Ein Resampling der Daten in ein reguläres 3D Gitter kann diesen Prozess vereinfachen: Verschiedene Detailstufen können trivial in einem "bottom-up" Ansatz erzeugt werden, indem die Daten der feineren Level räumlich gefiltert werden. In Kombination mit einer hierarchischen Szenenrepräsentation können sowohl der enthaltene Leerraum als auch unsichtbare Teile der Szenerie entfernt werden. Interaktives Rendering wird durch den Einsatz von GPU ray-casting ermöglicht, welches in der Lage ist, die benötigten Daten durch Berechnung von Verdeckungen weiter zu reduzieren. Die gewählte Gitterauflösung muss jedoch entsprechend hoch gewählt werden, um die feinsten Details der Szene zu erhalten. Dies führt in der Regel zu einem signifikanten Anstieg des Speicherbedarfs, selbst wenn die zum Erzeugen eines einzelnen Bildes benötigten Teildaten reduziert werden können. Typischerweise wird diesem Problem mit - potentiell verlustbehafteter - Datenkompression entgegengetreten. Obwohl hierdurch die Gesamtgröße der Datensätze reduziert werden kann, können die entstehenden Daten oftmals nicht direkt zur Visualisierung verwendet werden. Hierdurch sind viele Kompressionsalgorithmen nicht dazu geeignet, den zur Bilderzeugung benötigten Speicher sowie die Menge an Daten in GPU Caches zu reduzieren.

In dieser Dissertation wird ein GPU-getriebenes System zur Visualisierung hochaufgelöster Datensätze vorgestellt. Den Kern des Systems bilden neue, speichereffiziente, sample-basierte Datenstrukturen,

die das direkte Rendering aus einer kompakten Repräsentation ermöglichen und damit den Speicherbedarf sowohl zur Laufzeit als auch in der Datenhaltung deutlich reduzieren. Darüber hinaus bieten sie eine Verringerung der Renderzeiten verglichen mit existierenden Techniken. Die Skalierbarkeit des Systems wird im Bereich von partikel-basierten Simulationen sowie Oberflächenscans auf einem einzelnen Desktop PC demonstriert. Darüber hinaus wird der Einsatz in dynamischen Szenarios wie dem Rendern von Isoflächen in Skalarfeldern und der Online-Rekonstruktion mithilfe von mobilen Tiefensensoren präsentiert.

Acknowledgments

I gratefully acknowledge the support of all people who made this thesis possible. First and foremost, I would like to thank my supervisor, Prof. Dr. Rüdiger Westermann, for allowing me to pursue my research interests under his guidance, for endless hours of discussions, valuable advice, constant feedback and guidance. I am truly grateful to you for being this involved in the research of all your PhD students, even at times where the deadlines of countless projects collide. It is certainly not to be taken for granted.

I would also like to thank all of my co-authors—Matthäus G. Chajdas, Marc Treib, Kai Bürger, Jens Schneider and Jakob Weiss—for supporting my work and sitting through those nightly deadlines, and of course all current and former colleagues at the Chair for Computer Graphics and Visualization: Florian Ferstl, Ismail Demir, Mihaela Mihai, Christian Dick, Stefan Auer, Marc Rautenhaus, Shunting Cao, Jun Wu, Johannes Kehrer, Tobias Pfaffelmoser, Raymund Fülöp, Roland Fraedrich, Joachim Georgii, Hans-Georg Menz, Andreas Klein, Mika Vaaraniemi, Nils Thuerey, Rachel Chu, Tiffany Inglis, and Sebastian Wohner. You have made these office a really welcoming and nice place to work at all times.

Furthermore, I want to thank everyone supplying data sets along with valuable support and insight. Thanks to Volker Springel from the Max Planck Society in Garching for providing the Millennium Run data set, the Digital Michelangelo Project at Stanford for the David statue, Enrico Gobbetti of the CRS4 Computing Group in Italy for the colored version of the scan, and to Prof. Teschner and Markus Ihmsen of Freiburg University for the beautiful fluid simulations and their patience in the exchange process.

Last, but definitely not least, a big thank you to all my friends for being so awesome and supporting me whenever I needed it. The same goes, of course, for my incredible parents. I certainly would not be where and who I am today without you.

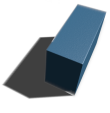
Contents

1 Introduction

1.1 Motivation

Visualization is an indispensable tool in a broad range of scientific domains. It allows domain experts a fast and accurate analysis of the presented data in an intuitive and goal-oriented way and greatly aids in conveying information to non-experts without burdening them with the underlying data itself. Especially interactive 3D visualization methods have become a well-adapted enrichment in many fields, as they provide immediate feedback to the user and thus facilitate exploration and a focused analysis of subsets of the input—either spatially, semantically, or both—which would be cumbersome or even impossible otherwise.
Visualized data can stem from a variety of sources, and the employed visualization methods and guidelines are as widespread as their applications. Most commonly, data is acquired in one of two ways. *Simulations* of real-world phenomena are employed to gain a deeper understanding of the processes present in nature. Examples include the broad field of computational fluid dynamics, seismic simulations, and weather forecasts. If available, *measurements* of the real world can serve as a starting point for simulations, or may be used for analysis on their own. 3D reconstruction of real-world geometry is gaining increased attention in fields like damage surveys, disaster management, construction monitoring and similar applications. Furthermore, it plays an important role in digital heritage initiatives striving to save cultural assets from destruction by preserving them as digital copies for further generations.

For nearly two decades, graphics processing units (GPUs) are capable of high-performance rasterization of triangle-based 3D data and are thus a prime choice for any application in need of interactive rendering. Over the last decade, they have evolved from simple co-processors capable of transforming and rasterizing triangles into fully programmable, highly parallel data processors. As of today, GPUs are utilized not only in all fields of visualization, but also in simulation and general-purpose computation. While they are in general not as flexible as generic CPUs, their high memory bandwidth and streamlined, parallel programming design make them superior in data processing, given a suitable parallel formulation of



the task at hand. However, the amount of fast memory is much more limited than the random access memory (RAM) present in CPUs. Even though a few recent models show a significant increase in the available GPU memory of up to 16 GB, the difference is still up to an order of magnitude in favor of CPU memory.

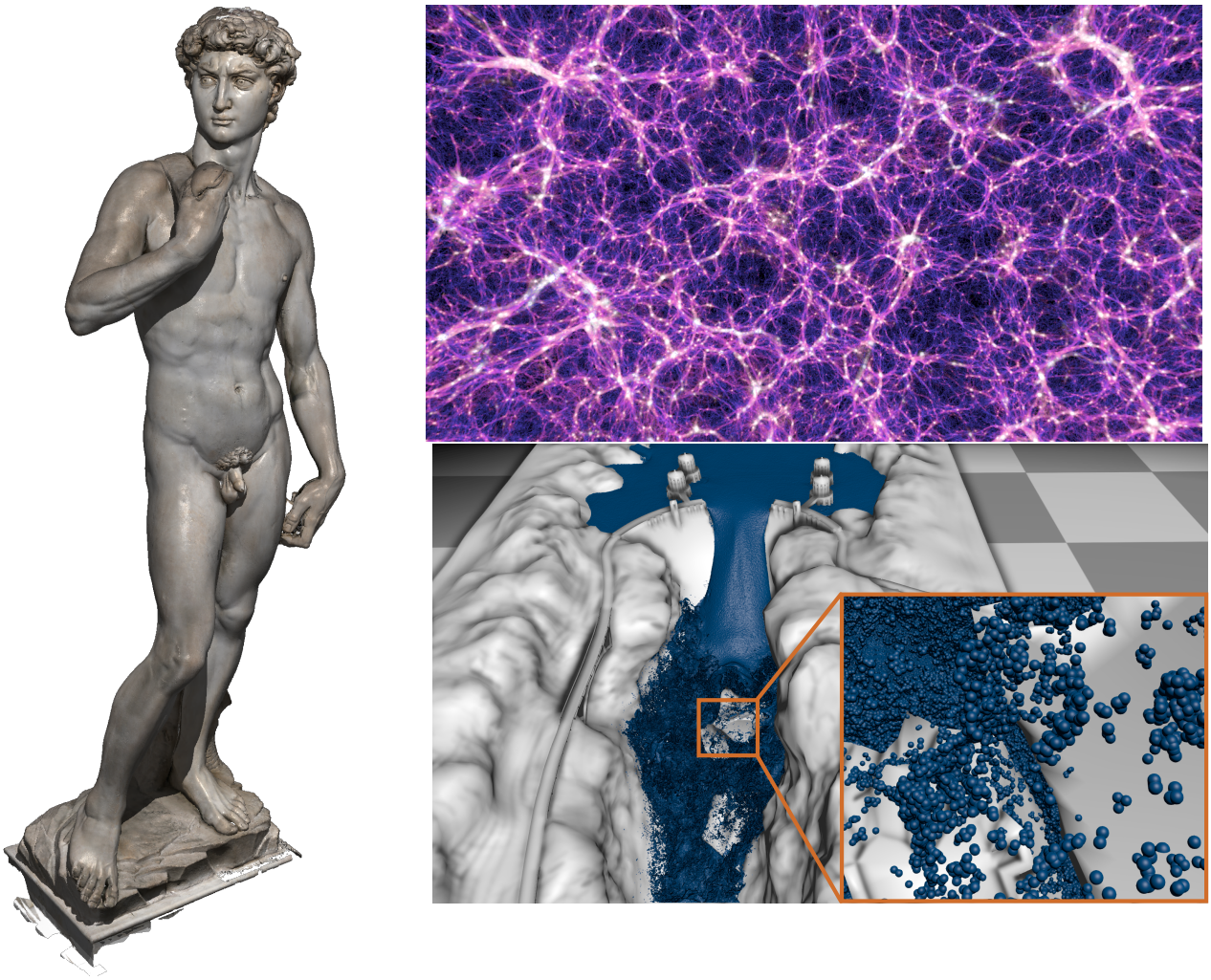


Figure 1.1: Examples of the data sets this thesis is concerned with: A triangle representation of a surface scan comprising about a billion triangles (left, 22 GB input data size), a large-scale gas dynamics simulation of galaxy formations (top right, 225 GB), and a time-dependent fluid simulation consisting of over 200 million particles and 350 time steps (bottom right, 780 GB). Our system allows highly interactive explorations of all data sets using a single GPU.

Due to advances in algorithms as well as hardware for data acquisition, processing, simulation and storage, the last years have shown a rapid increase in the amount of generated information. Data sets in the size of several hundred giga- or even terabytes are not uncommon and pose an increasingly difficult challenge for interactive visualization systems as they exceed the amount of available fast CPU and particularly GPU memory by orders of magnitude. For example, the particle-based gas dynamics

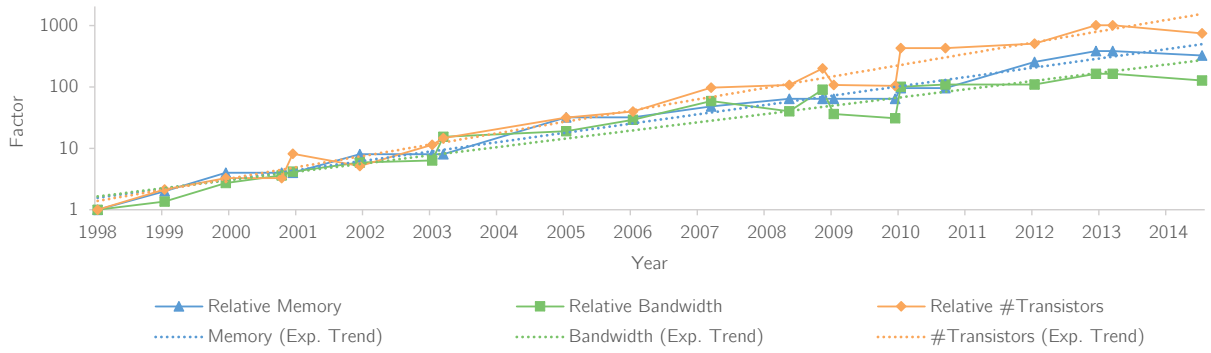


Figure 1.2: Development of GPU memory and performance, exemplary shown for NVIDIA GPUs. Due to the highly varying performance characteristics of GPUs published concurrently, the data for all "high end" devices of each generation are shown. Exponential trends are given as dotted lines. Data source: [Wik].

simulation known as the Millennium Run shown in Fig. 1.1 (top right) consists of 64 time steps, each of which requires about 225 GB of storage memory for over 10 billion particles.

1.2 Big Data Challenges

In recent years, the term *Big Data* emerged in a broad range of domains to describe the problems and solutions that arise when dealing with growing amounts of data. Commonly, Big Data systems face the challenge of efficient and scalable storage and processing of information which is either extremely large and complex or of dynamic nature and frequently changing.

1.2.1 Memory Hierarchies

One fundamental problem for big data applications is the inverse proportionality of storage memory access speed to density: Fast memory, like CPU caches, relies on storing data in active feedback loops driven and accessed by multiple transistors which cannot be densely packed. It is thus limited by the available area to cover on a CPU die and cannot exceed sizes of a few megabytes. Larger memory which relies on different types of storage hardware such as capacitors or magnetic devices, on the other hand, is comparably slow in terms of read- and write access, but can be packed extremely dense. In addition, the rate at which data can be moved from one memory area to another is often severely limited by the available bandwidth.
To make matters even worse, as shown in Fig. 1.2, there exists widening gap between the growth GPU processing power on the one hand and memory sizes as well as bandwidths on the other, though both

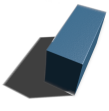


exhibit an exponential increase as predicted by Gordon Moore in 1965 [Moo65]. Thus, the presented problems will subsist and even increase in future hardware generations.

In the context of this thesis, three main levels of memory are considered:

1. **Storage memory:**
   External storage, hard disks (HDDs) or solid state disks (SSDs). Indefinite data storage and "unlimited" capacity (i.e. sufficient at any time), but comparably slow access.
2. **Main memory:**
   CPU main memory (RAM). Temporary storage, data is lost when powered off. Fast access, but limited capacity.
3. **GPU memory:**
   Memory accessible from the GPU. Temporary storage, data is lost when powered off. Very fast access, but even more limited capacity. GPUs require data to be stored in this memory area for interactive rendering.

Typical sizes and bandwidths are illustrated in Fig. 1.3. Several lower levels of faster, smaller memory such as CPU and GPU registers and caches or GPU shared memory have not been included in this overview for simplicity as they are not of impact on the high-level system design, but only considered in algorithmic details. Higher levels—such as local or global network storage—are treated as local storage memory.

1.2.2 Data Streaming and Caching

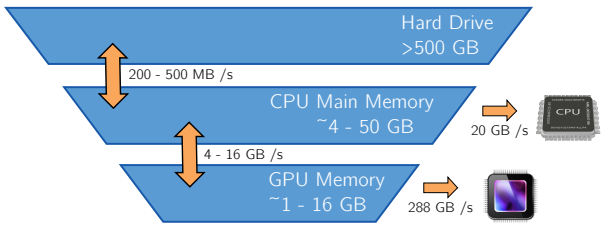


Figure 1.3: Typically available memory and approximate bandwidths from storage and caches.

To handle growing amounts of data, a visualization system must be able to utilize the lower levels of the memory hierarchy as far as possible. These lower levels typically act as inclusive caches for the higher levels, e.g. the data residing in GPU memory is a portion of the data in CPU main memory. If the available memory on a level is exhausted, a replacement policy determines how to free space for newly required data. In an optimal case, this policy is able to predict the usage of data in the future and can discard information that will not be accessed for the longest time. In practice, such a policy—commonly known as Bélády's Algorithm [Bel66]—cannot be implemented as a prediction of future usage is impossible or at least impractical. Thus, caching algorithms often rely on a data management based on the *past* usage, for example a (chunk-based) *least recently used (LRU)* policy: All data is divided into chunks of a certain size, and each chunk is supplied with a timestamp indicating its last usage. When a new chunk

needs to be inserted, the one with the oldest timestamp is replaced if no more memory is available. With this process, all required data is kept in fast memory as long as possible; it is often referred to as *paging*, with chunks termed as *pages*. Other similar policies include *most recently used*, *random replacement*, and *least frequently used*. In the end, the choice of replacement policy is in general application-specific and heuristic.

A critical bottleneck in rendering performance is the available amount of GPU memory, as this is the only memory modern GPUs can render data from. Upon exhaustion, image generation must be halted until the missing pages are delivered from main memory. Due to the asynchronous nature of GPUs, a few MiB of data can be transferred between two frames without noticeable impacts on rendering performance, but in-frame transfers must be avoided if possible since they force the GPU to wait for the transfer to complete. If the requested pages are not yet residing in CPU memory and need to be streamed from storage, the delay may well lie in the order of seconds and cannot be tolerated in the context of an interactive system. In this case, rendering an incomplete image or a coarser representation of reduced detail can keep the system responsive. To minimize the occurrence of this case, *prefetching* of information possibly required in the future can be employed based on a heuristic determination of the user's exploration behavior.

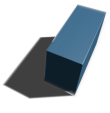
1.2.3 Data Reduction

Data reduction mechanisms are of utmost importance for any system aiming at interactive rendering in all stages of the pipeline to reduce the required bandwidth as well as the number of rendered primitives. In the context of visualization, we can classify two approaches to achieve data reduction:

1. **Compression:**
   Techniques that reduce the overall size of the data by employing—lossless or lossy—compression algorithms.

2. **Visibility-based reduction:**
   Techniques that reduce the required amount of data based on the information to be currently displayed, e.g. by discarding data outside of the camera's view frustum (*frustum culling*), discarding parts of the scene invisible due to occlusion (*occlusion culling*), or utilizing coarser representations of the data (*level of detail, LoD*) for regions further from the viewer.

Data compression is commonly used for data residing in storage memory to reduce the overall memory requirements. In many cases, general-purpose techniques can be applied in a non-application-specific way without any deeper knowledge of the underlying data. Entropy coding, run-length encoding and dictionary techniques—or a combination thereof—are popular choices for a lossless encoding of arbitrary data. These techniques are often combined with a more specialized form of transform coding such as a *discrete wavelet transform* when dealing with locally coherent data such as images, CT scans, or audio



streams to achieve optimal compression. For a comprehensive review of data compression which goes beyond the scope of this thesis, we refer the reader to [Say12].
While these approaches can greatly reduce the storage memory requirements and therefore also the bandwidth limitations when streaming the data from external storage, the data set generally can not be rendered directly from this representation. A vast part of the rendering performance of current GPUs originates from their highly parallel processing pipelines optimized for some form of regular, evenly-sized data chunks (such as vertex data present in triangle meshes). Many approaches for data compression, however, seek to reduce and combine similar or identical data as far as possible and often introduce inter-dependencies—e.g. an incremental encoding of successive chunks—to achieve this, making parallel random access impossible or at least unfeasible. Thus, it is necessary to decompress the data before rendering, which leaves the run-time memory requirements for image generation unchanged.

To avoid this need for intermediate decompression, big data visualization systems instead often opt for custom tailored, special-purpose compression schemes that are tightly coupled to the underlying data representation. Examples include point cloud run encoding (DuoDecim, [KSW05]), triangle strip compression for ray-tracing (ReduceM, [LYTM08]) or incremental encoding of vertex positions for terrain rendering applications [DKW09], to name just a few.

Visibility-based reduction, on the other hand, relies on knowledge of the underlying data and the currently rendered scene to restrict the amount of displayed information to the parts of the scene actually visible to the user. In large data sets, *culling* algorithms are performed on coarse bounding volumes rather than on the actual primitives, since element-wise visibility determination is impractical for millions of points or triangles. The granularity is chosen as a good trade-off between accuracy and efficiency, and visibility testing has to be performed conservatively on groups of elements of the chosen granularity. This usually requires the data set to be broken down in smaller, locally coherent chunks in a preprocess.

Culling mechanisms are often combined with LoD techniques. Mimicking the human visual system, rendering is mostly performed under a perspective projection: objects are projected onto a smaller area of the screen with increasing distance from the viewer as shown in Fig. 1.4. Yet current monitors can display the resulting image at a fixed granularity of one pixel at most, essentially discarding all information below pixel size. LoD techniques exploit this fact by displaying coarser resolutions of parts of the scene at distances where the differences can not be observed.

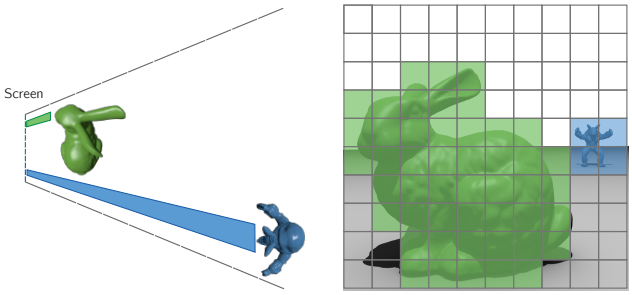


Figure 1.4: Perspective projection of two objects onto the screen. The blue object is projected to significantly fewer pixels will convey less detailed information to the viewer.

A vast number of visibility algorithms exist for specific scenarios such as urban environments, landscapes, or applications like (soft) shadows and sound propagation. For a general overview on the broad topic of visibility culling, we refer to the surveys by Cohen-Or et al. [COCSD03] and Bittner and Wonka [BW03]. In general, two classes of algorithms can be employed, both of which try to compute potentially visible sets (*PVS*) for a specific scene under certain conditions. *From-region* methods such as cells and portals [TS91] rely on extensive preprocessing of data sets to determine connected regions and their mutual visibility. They come with little to no runtime overhead since the complete visibility information is precomputed, but they are difficult to implement, restricted to static scenes and very dependent on the scene geometry. While the current state of the art [BMW*09] achieves results that are competitive to algorithms that evaluate visibility during runtime, it requires multiple minutes of preprocessing and comes at the trade-off of a slight pixel error. In addition, it can only deal with a limited number of subdivision cells, which reduces its efficiency for large data sets.

From-point methods, on the other hand, try to determine a minimal PVS for the current position of the viewer in every frame during runtime. Two simple heuristics that come with little overhead are *frustum culling* and *backface culling* (see Fig. 1.5). The former works by discarding geometry outside of the current view frustum, the latter is specific to triangle-based rendering and discards primitives deemed as *backfacing* depending on the order of their vertices. Both mechanisms are standard in today's GPU rasterization pipeline and provide an early-out of primitives to relieve the rasterizer and shading units. Frustum culling, in addition, can be performed in an efficient way on precalculated bounding volumes of parts of the scene—or more complex hierarchies thereof—on the CPU to avoid sending invisible objects to the GPU pipeline at all.

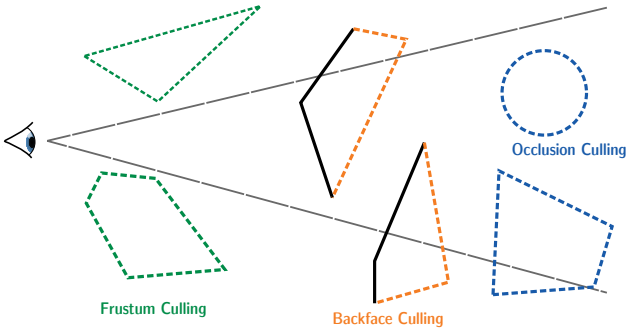
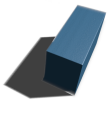


Figure 1.5: Frustum, backface and occlusion culling. Only the solid primitives need to be rendered. The gray lines depict the view frustum.

Much more complex to integrate into the standard rasterization pipeline is *occlusion culling* which exploits the fact that primitives may be hidden by larger objects in the scene even though they are in the current view-frustum and front-facing. The determination of a PVS based on occlusion culling, however, is a non-trivial task for complex scenes. Depending on the granularity of the bounding volumes, a huge number of potential combinations of occluders and occludees must be considered in each frame. To speed up this task, all modern GPUs offer support for hardware *occlusion queries*. This feature allows the developer to determine if at least one pixel will be affected if a draw call for a certain object were issued and is the standard way to determine occlusion in GPU-based rasterization. Due to the required read-back of information from the otherwise asynchronously working GPU and the long graphics pipeline, these queries must be used with care to avoid unnecessary CPU / GPU synchronization points resulting



from latency. Otherwise, CPU stalls and GPU starvation may occur and in fact drastically decrease the performance instead of producing the desired speed-up. In addition, a fine-granular subdivision of the scene forces a high number of individual draw calls and GPU state changes, which come with a negative performance impact.
Bittner et al. [BWPP04] were the first to introduce hierarchical occlusion culling with temporal coherency to address the problem of GPU starvation. They traverse a scene hierarchy in front-to-back order and rely on the visibility information obtained from the last frame to hide the query latency. Only previously visible leaves and invisible boundary regions are queried, and visible parts of the scene are assumed to stay visible and rendered immediately. The overhead of large numbers of visibility queries may however still decrease the performance in comparison to simple frustum culling for scenes with relatively few and localized occlusions. This is determined and improved upon by Guthe et al. [GBK06] by introducing an additional statistical occlusion model and a hardware calibration step. A few years later, Mattausch et al. [MBW08] introduce CHC++ which also addresses the issue of an increased number of state changes due to occlusion culling by batching visible nodes during rendering. They also improve the previous model by eliminating the hardware calibration step, whose test scenarios may not by applicable to all possible real-world settings.

1.3 Rasterization Limitations

Nowadays, *rasterization* is the predominant algorithm for interactive GPU-based rendering. The algorithm starts from a primitive which is projected onto the screen according to the current camera parameters. All covered pixels are then determined by rasterization of the primitive surface, a task for which current GPUs are highly optimized. They can currently process over 4 billion primitives per second in this way under optimal conditions. For a more in-depth introduction and a detailed coverage of pipeline stages, let us refer to [AMHH08].
A triangle-based representation can be computed for a variety of input-data as long as a reasonable surface inside the data exists, see e.g. the marching cubes algorithm for volume data [LC87]. Even if this is not possible, the rasterization power of GPUs can be utilized by rendering the data as *proxy geometry*: For example, 2D screen-space aligned quads can appear as perfect spheres by performing ray-sphere intersection in a pixel shader and can, thus, be used in particle rendering. However, triangle data and rasterization comes with a few inherent drawbacks making it less suitable for massive model rendering than the raw numbers may suggest.

1.3.1 Influence of Small Triangles

In practice, the theoretical maximum triangle throughput is hard to reach and influenced by a number of factors. For highly tessellated models, triangles quickly project to areas of a single pixel or even

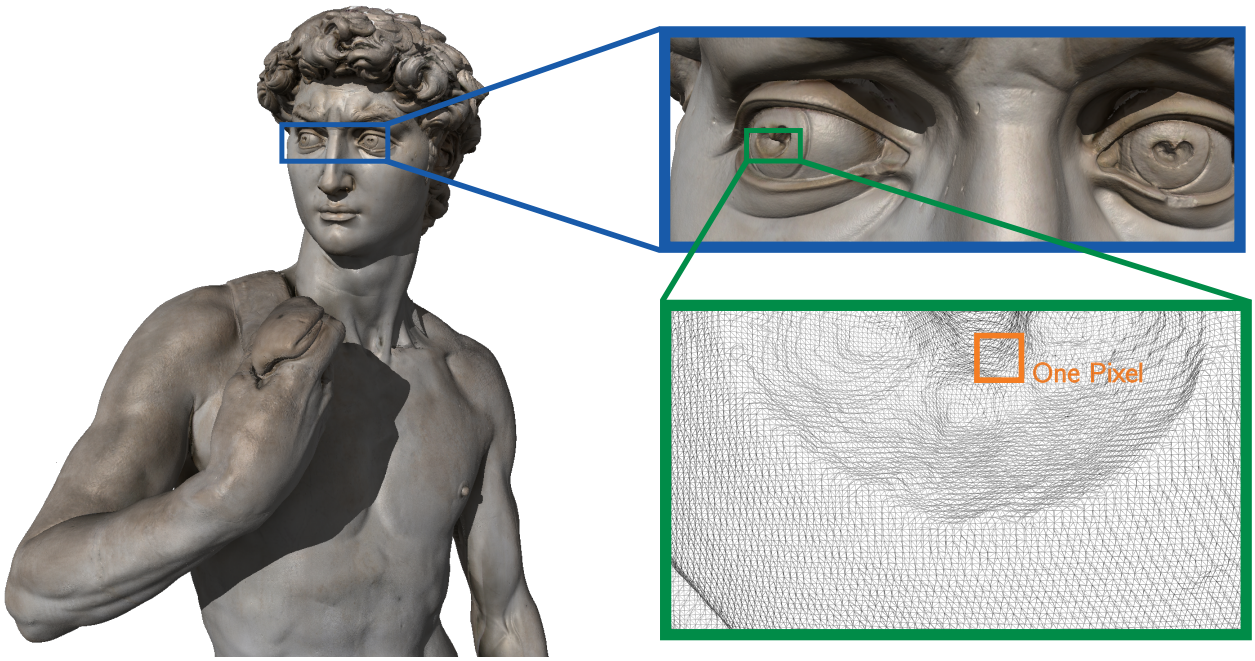
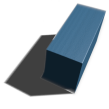


Figure 1.6: Triangle sizes of the David statue when rendered to a 2560×1440 viewport.

considerably smaller. For example, the David statue depicted in Fig. 1.6 (left) consists of nearly a billion triangles in total. The inset shows the geometry of the statue when rendered onto a 2560 × 1440 viewport. Clearly the majority of triangles is below pixel size, with high numbers of unique primitives falling onto the same fragment as shown in the marked pixel of the inset. In this case two major problems arise:

- **GPU Pipeline Inefficiency:**
  GPUs are optimized for fast processing and rasterization of scenes with relatively large triangles at high frame rates. In the case of sub-pixel sized triangles, the overhead of transforming and projecting each individual primitive exceeds the amount of work performed by the rasterization hardware, which is in turn stalled while waiting for previous parts of the pipeline to complete. Even if this was not the case, several optimizations such as hierarchical rasterization and parallel coverage tests cannot unfold their potential [Cra11, FBH*10].

- **Overdraw:**
  All surfaces more complex than a simple plane will always introduce *overdraw*. Rasterization algorithms output a number of *fragments*, which are basically potential screen-pixels with depth information. If multiple primitives overlap in the same pixel, a fragment will be generated for each primitive, and collisions are handled in screen-space by utilizing the *z-Buffer* [Cat74]. The number of fragments generated can, thus, vastly exceed the number of finally visible pixels. This also introduces *over-shading* since each fragment is shaded individually by a potentially complex shader.



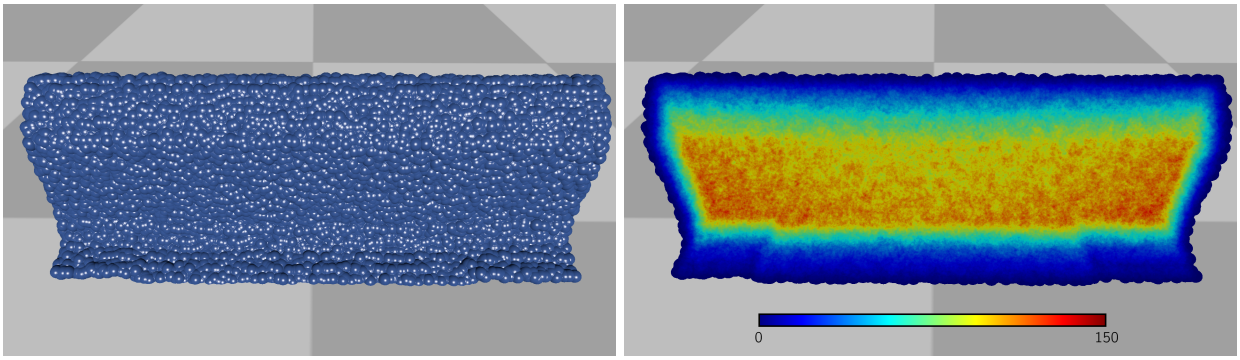


Figure 1.7: Amount of over-shading for small triangles. 100,000 particles of a fluid simulation are rendered as screen-space aligned quads of two triangles with ray-sphere intersections performed in a pixel shader. The left image shows the corresponding surface, the right the color-coded amount of overdraw for each pixel.

Depending on the order of the fragments passing through the pipeline, early-z tests in modern hardware can diminish the amount of over-shading by rejecting fragments before they are passed to the fragment shader, but cannot reduce the work performed by the rasterizer. Even more so, deferred shading [ST90] can effectively shade each finally visible pixel only once, but cannot deal with transparent surfaces.

Figure 1.7 demonstrates this in the context of rendering particle-based fluids where each individual particle of a fluid simulation comprising 100,000 particles in total is shown. Each particle is rendered as a screen-space aligned quad of two triangles, with ray-sphere intersections performed in the pixel shader. In this early time step of the simulations, the particles are tightly clustered, resulting in a vast amount of pixels being over-shaded as often as over 100 times, even though only a comparably small number of triangles are visible in the final image.

In addition to the resulting performance implications, the severe undersampling of the geometry also degrades the image quality due to aliasing artifacts under a moving camera. This will be discussed in more depth in section 2.1.1.

1.3.2 Mesh Simplification

To overcome this problem, triangle-based massive model rendering systems employ mesh simplification algorithms to drastically reduce the number of rendered primitives. In combination with LoD techniques, objects more distant to the camera can be represented with only a few faces, even if the original model comprised hundreds of millions of triangles. Building and rendering a LoD representation of a massive scene, however, is a challenging problem. A variety of techniques and systems exist, but each suffers from its own drawbacks and is specifically designed for a selected use-case [GKY08]. For example, terrain rendering systems [CGG*03, DKW09] are able to interactively visualize vast landscapes from

highly compressed representations, but specifically rely on the 2D parametrization of the underlying height field.

Popular techniques for mesh simplification work by iteratively collapsing edges [HDD*93] or removing vertices [SZL92] (Fig. 1.8). Conceptually, the algorithms minimize an energy function that captures the amount of discrepancy between a tight geometric fit compared to the original mesh and a compact representation. The error introduced by replacing a mesh by its coarser representation is often based on quadric error metrics [GH97, SG01].

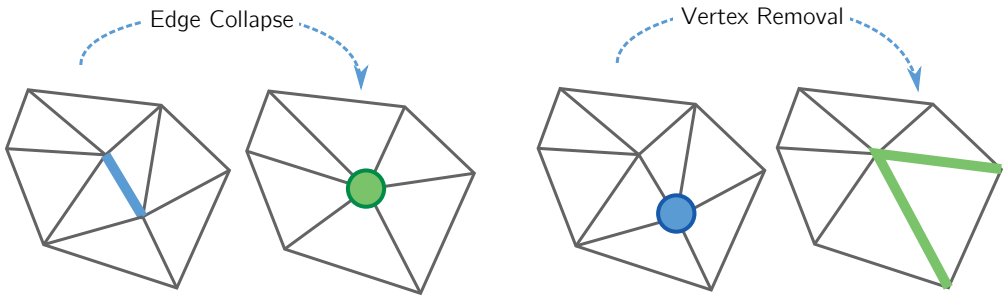
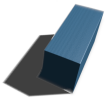


Figure 1.8: Edge collapse and vertex removal.

Rendering and LoD selection are tightly coupled to the mesh decimation process. A popular technique for the generation and handling of coarser representations of triangle data is *progressive meshes* [Hop96, Hop97, Hop98] due to its good balance of generality, quality and error-control. Hoppe introduces the representation of a model as a base mesh and a series of collapse-operations to generate meshes of desired triangle count which can then be used for view-dependent refinement. A similar approach was pursued by [XV96]. Based on these early works, a number of adaptive, view-dependent level of detail rendering and simplification systems have been proposed that sacrifice either scalability, visualization speed or decimation quality [CGG*04]. In particular, existing out-of-core methods for geometric simplification [Lin00] mostly target smooth, uniformly tessellated and topologically simple surfaces [GKY08] but do not scale well to arbitrary scenes.

Systems that can handle complex surfaces in an out-of-core manner often suffer from LoD popping artifacts unless a very conservative error estimation is used [Lev02]. They also need to take special care during model partitioning as the resulting non-localized memory access quickly becomes problematic for the loading and rendering performance [YSGM04], and potential cracks between adjacent parts of different LoDs need to be considered. Largely varying triangle sizes also force the used error metrics to be evaluated in larger areas of the scene, increasing the processing times. Many approaches trade runtime complexity for preprocessing times, such as the Quick-VDR system of Yoon et al. [YSGM04], where preprocessing of a 300 million triangle data set takes over 2000 hours. The same data set is processed in about 11 hours in a more recent work by Derzapf et al. [DG12]. For the same task, the adaptive TetraPuzzles system [CGG*04] requires about 20 hours, but can be fully parallelized to a cluster of multiple nodes. On the downside, it requires huge amounts of temporary memory. In some applications,



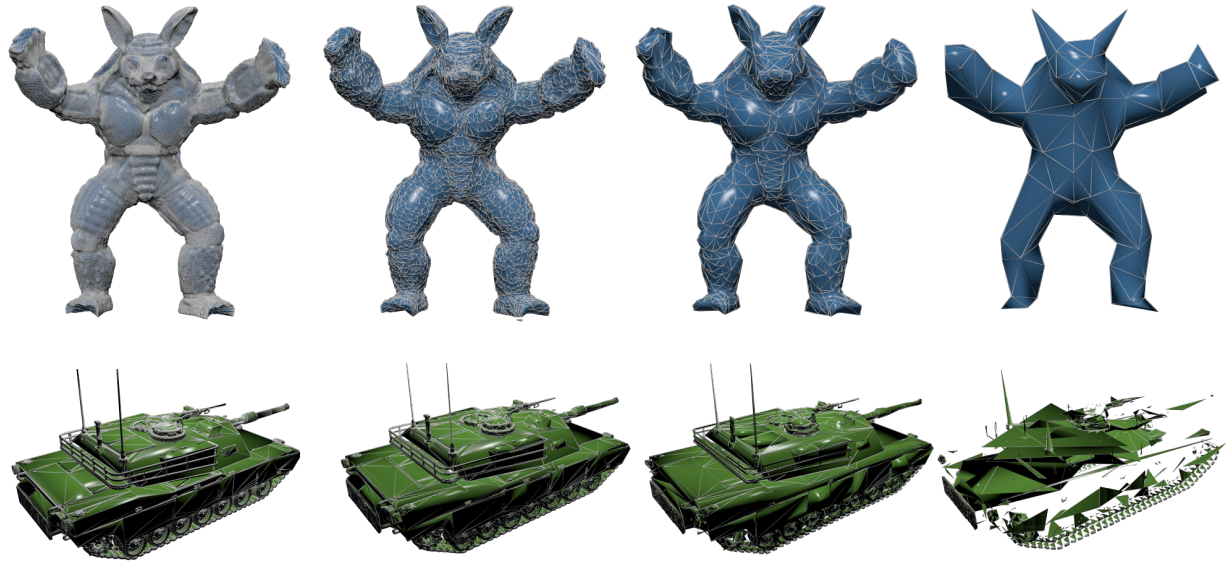


Figure 1.9: Downsampling of a uniformly triangulated triangle mesh (top) and a more general mesh (bottom).

simplification can be performed on the basis of single objects in a scene consisting of a large number of models [EMI01, PC12]. While [EMI01] is also able to split larger models into sub-objects, the approach exhibits severe LoD popping artifacts due to the use of static display lists.

Most systems based on progressive meshes also require a large amount of CPU interaction as many operations are hard to parallelize and thus not well suited for the GPU. This introduces CPU/GPU synchronization points and possible pipeline stalls which degrades performance.

1.3.3 Appearance Preservation

The error resulting from a mesh simplification is generally measured by geometric distances and can then be used to guarantee a certain maximum geometric error in screen space during rendering. In addition to this error, however, the appearance of the final mesh may be heavily influenced by small changes in surface orientations: With slightly differing light scattering behavior due to fine details—or the lack thereof—, highlights can rapidly change position when switching between LoDs that otherwise preserve a maximum geometric error, especially if highly reflective materials are used (Fig. 1.9).

Appearance preserving simplification tries to incorporate this change into the LoD representation. A prominent example is baking of high-frequency surface details into normal maps [COM98], which can then be downsampled and mapped onto the coarser representations to account for the loss in surface detail. To achieve this, a parametrization of the underlying mesh is required, which is often not present in the case of simulated or scanned data. Automatic generation of a good surface parametrization, in

turn, is a hard problem on its own especially if consistency over multiple LoDs is required (see 3.1). Appearance preserving mesh simplification error metrics such as the modified quadric error metric [GH98] or curvature-based estimations [Lin03], on the other hand, are unable to maintain a guaranteed screen space error [GBBK04].

For these reasons, alternative LoD representations that can maintain a visual as well as a geometric error have been sought after. An early example is presented by Guthe et al. [GBBK04], who combine triangle- and point-based LoDs to heavily simplify huge models while still preserving the visual cues indicating a highly detailed surface. A number of approaches follow their example, which we will review in section 3.4.

1.4 Sample-based Representations

Since the beginnings of computer graphics, numerous sample-based representations have been proposed to either enhance existing geometry by fine-scale detail, or to completely replace it to offer a simplified appearance preserving LoD generation and selection. In these approaches, the source data is resampled into a discrete uniform 2D or 3D Cartesian grid, which we will in general refer to as a *sample-based representation (SBR)*. Especially recent approaches [CNLE09, LK10] rely on view-independent 3D grids made up from evenly sized cubic volumetric elements (*voxels*).

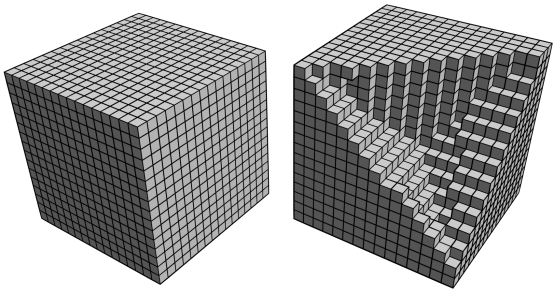
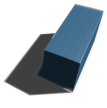


Figure 1.10: Voxel grids.

This regular structure can be exploited to efficiently generate coarser representations bottom-up by spatially filtering the data present in the finer levels. Appearance information such as colors, normals or material properties are stored along with the geometry and appropriately filtered along multiple levels to maintain the fine-granular visual properties of the input data. Combined with a hierarchical scene representation, empty space can be removed and view-based culling can be performed at a fine-granular level. On modern programmable graphics hardware, this representation can then be rendered directly using ray-casting, which inherently provides additional data reduction in the form of occlusion culling at very little cost.

For densely sampled volumetric data present in e.g. weather forecasting or medical CT scans, a 3D data structure is a perfect fit and an intuitive, widely used solution. However, many applications generate sparsely distributed data—for example, scanned data in digital heritage systems represents a set of surfaces and is consequently exported as triangle data or directly rendered from point elements of varying spatial extent. Leveraging this sparse data into a 3D domain drastically increases the overall memory



requirements as the sampling resolution is determined by the finest geometric detail across the scene if no loss of quality should occur. Once again, compression can be used to diminish this problem. Lossy compression can often not be employed or must be highly application-specific, as a maximum visible error in the resulting representation must be guaranteed. Thus the amount of data reduction by generic compression is limited and, as stated, cannot use the current visible set in a single frame, which may already exceed the available GPU memory for larger view-ports and high-resolution input data.

1.5 Contributions

This thesis provides solutions for the stated problems in interactive 3D big data visualization. It introduces a versatile rendering framework with multiple highly scalable, compact, sample-based data representations at its core. While the underlying principles are applicable to a large range of possible input data representations, we specifically introduce solutions for two wide-spread scientific fields which commonly deal with extremely large data sets: particle-based simulations and 3D surface scans given either as point-representations or as high-resolution triangle meshes. In contrast to previous sample-based rendering systems, the presented data structures are more application-specific and thus able to provide highly optimized, directly renderable representations with a significant reduction of memory requirements in comparison to the source data or a regular resampling, combined with an increase in rendering speed. We describe the design of all stages of the pipeline of a massive out-of-core rendering system and highlight the commonalities as well as differences in handling particle- and triangle-based data. All parts of the pipeline are designed to run on a single desktop PC equipped with a single GPU and do not require clusters for preprocessing or rendering.

The framework is tailored to suffice three primary requirements:

1. **Scalability:**
   Support for big data sets, both in spatial extent as well as local sampling resolution, without sacrificing interactivity. At the same time, the data representations must provide efficient compression to minimize the memory footprint.

2. **Interactivity:**
   This includes low rendering times, low latency for visual feedback of requests, and a smooth exploration without noticeable stalls while data is generated or delivered.

3. **Flexibility:**
   Building upon a common rendering framework, a variety of data must be supported.

As already hinted at in this section, resampling the data onto a discrete 3D Cartesian grid to obtain a SBR provides a good fit for all three goals: Scalability is provided by the integration into a ray-guided, out-of-core ray-casting system, which allows for fast visibility determination and culling with little overhead and

offers a seamless appearance-preserving level of detail integration, if desired. Interactivity is achieved by utilizing the massive parallel power of modern GPUs, which lend themselves very well to the required operations. And finally, flexibility results from an easily customizable data representation. By the use of a variety of custom-tailored SBRs, we do not restrict our system to surfaces, but also provide support for truly volumetric data.

The specific contributions include:

1. A new approach for the rendering of big data sets from SPH simulations, which is based on a novel combination of lossy compression, adaptive top-down level of detail generation and direct volume rendering. We demonstrate preprocessing and rendering of a 10 billion particle data set.

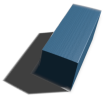
2. A highly compact data structure based on binary grids for the visualization of particle-based fluid simulations. This data structure complements the one introduced for a similar application in 1) as it offers completely I/O bound, lightweight preprocessing during loading times.

3. The *Run-Length Encoded Sample Buffer*, a new data structure for compact storage of surface data. It provides fast conversion and rendering of arbitrary surfaces. We demonstrate its versatility by integration into a hybrid triangle / voxel rendering system and an isosurface ray-caster for volumetric data sets.

4. A memory-efficient online scene reconstruction algorithm based on binary grids that provides large-scale reconstruction from hand-held depth sensors at comparable quality to existing methods while reducing the memory requirements for surface and color data by an order of magnitude.

1.6 Outline

The remainder of this thesis is structured as follows. Chapter 2 provides the reader with the relevant fundamentals for creation of and rendering from SBRs, with a focus on interactive techniques utilizing modern graphics hardware to further establish the suitability of our approach for big data visualization. In chapter 3, we review existing work in this context, and we then derive the design of our out-of-core rendering framework based on these publications in chapter 4. Chapters 5 and 6 address the rendering of particle simulations. We first discuss possibilities and limitations in the direct volume rendering of astrophysical gas dynamics simulations, which are characterized by huge, varying particle radii. Chapter 6, on the contrary, presents a data structure for the interactive rendering of giga-particle fluid simulations with small and constant particle radii. Moving from particle-based to surface data, chapter 7 introduces a compact, flexible data structure for the rendering of triangle data obtained from high-resolution surface scans and shows its application in a hybrid triangle / sample-based rendering pipeline. Finally, chapter 8 showcases the usage of our system in online surface reconstruction from hand-held depth sensors for



data acquisition. The thesis is then concluded with a summary of the results and an outlook on future work in Chapter 9.

1.7 List of Publications

Part of this thesis is based upon research results originally published or submitted as the following peer-reviewed conference papers:

1. Reichl F., Chajdas M. G., Bürger K., Westermann R.: Hybrid sample-based surface rendering. In *Proceedings of Vision, Modeling and Visualization 2012* (2012), pp. 47–54
2. Reichl F., Treib M., Westermann R.: Visualization of big sph simulations via compressed octree grids. In *Proceedings of IEEE Big Data 2013* (2013), pp. 71–78
3. Reichl F., Chajdas M., Schneider J., Westermann R.: Interactive rendering of giga-particle fluid simulations. In *Proceedings of High Performance Graphics 2014* (2014)
4. Reichl F., Weiss J., Westermann R.: Memory-efficient scene reconstruction from depth image streams. Submitted to Eurographics 2015

2 Fundamentals

2.1 Sample-based Representations

Digital storage and processing of information requires the data to be converted and, thus, discretized to a certain maximum available precision, yet our real world consists of continuous signals as perceived by human senses. In most cases, no implicit description of these signals can be derived. Stemming from the world of signal processing, *sampling* refers to the reduction of a continuous signal into a discrete one by storing support points in regular intervals. In the context of this thesis, we refer to a representation as *sample-based* whenever the data is given at regular sampling points on a Cartesian grid in any number of dimensions. Sampling theory and digital filtering on itself is a complex topic and has been intensively covered in countless publications. This introduction can only provide a brief overview to establish a basic understanding on the fundamentals of sampling and reconstruction. For a more thorough investigation of the subject, we refer the reader to [GW06, PM96, Wol90]. A more computer graphics related and shorter overview can also be found in [AMHH08] (p. 117 ff.).

A continuous signal is sampled in a regular *sampling interval* Δt to receive a discrete and, therefore, digitally storable representation. Fundamental for this is the observation that the sampled data can be used to reconstruct the original signal by the means of *filtering*. Figure 2.1 shows a simplified 1D case to

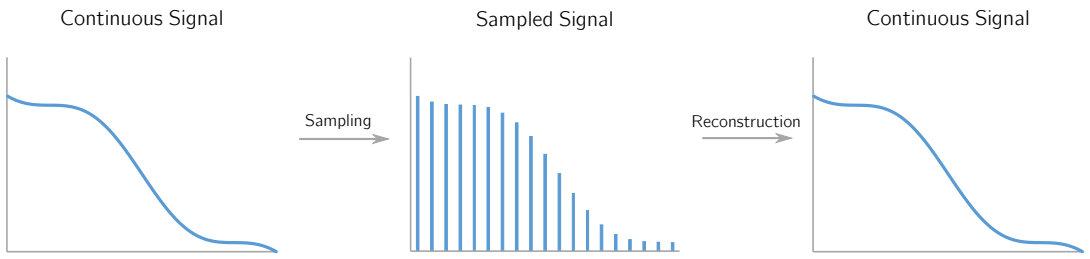


Figure 2.1: A continuous signal (left) is sampled (middle) and reconstructed (right).

illustrate the basic principles, which can be straightforward applied to higher dimensions. Reconstruction is performed by the application of a filter; for example, the *sinc* filter

$$sinc(x) = \frac{sin(\pi x)}{\pi x} \tag{2.1}$$

is an ideal low-pass filter that eliminates all frequencies above $\frac{1}{2}$ of the sampling rate. Assuming the signal is bandlimited, i.e. does not exhibit frequencies above a certain threshold, sampling in intervals of at least the so-called *Nyquist rate* of twice the maximum frequency allows the original signal to be perfectly reconstructed. In this case, sinc$(f_s x)$ yields this perfect reconstruction filter, with f_s being the sampling frequency. Sampling a signal at a rate above its Nyquist rate will, thus, not increase the amount of contained information and is referred to as *oversampling*. In contrast, sampling at a rate below this threshold discards information of the input signal and is known as *undersampling*.

Besides requiring the costly evaluation of trigonometric functions, the sinc filter has the property of unlimited support as depicted in Fig. 2.2 (top), making it impractical for applications where a fast and low-cost filtering must be performed. Instead, filters of limited support are used for faster reconstruction of a sampled signal. Two popular filters in real-time graphics are shown in Fig. 2.2 (middle and bottom): The *box* and *tent* filter, which implement nearest neighbor and linear interpolation. Several more advanced filters that provide high-quality reconstructions exist [Nov05, MN88]. Especially Gaussian filters of varying support are very prominent due to their exclusively positive filtering values.

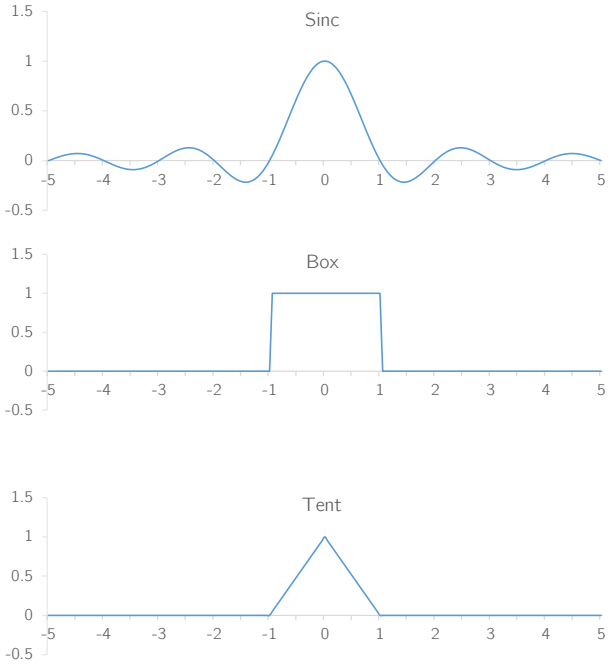


Figure 2.2: Sinc, box and tent-filters.

2.1.1 Resampling

The result of data visualization is typically a 2D image presented to the user on some form of graphical display. Current displays are unable to depict continuous signals, but also rely on regular sampling of the input to a discrete set of pixels. With the input signal already being sampled during data acquisition and processing, *resampling* is required to generate the final image. In this process, the signal is first reconstructed and then sampled again at a different interval $\alpha\Delta t$ under the presence of either magnification ($\alpha < 1$, upsampling) or minification ($\alpha > 1$, downsampling).

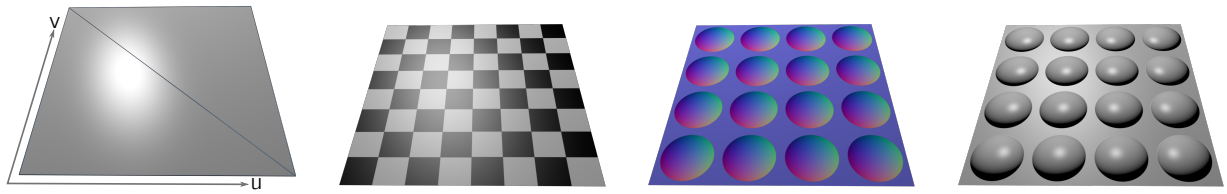


Figure 2.3: Texture-mapping of a simple object: No texture, diffuse color texture, normal map visualized, normal map shading (from left to right).

Magnification can easily be achieved by sampling the reconstructed signal at a higher rate. If the signal had been perfectly reconstructed, the smaller interval $\alpha\Delta t$ will be able to capture all present details and can once again serve as a basis for perfect reconstruction. Minification, on the other hand, may introduce undesirable *aliasing* artifacts. A perfect reconstruction would be obtained by applying $\text{sinc}(\frac{x}{\alpha})$, resulting in a low-pass filter of the original signal. When using approximate filters of limited support, it is therefore important to remove the higher frequencies beforehand to mimic the behavior of the sinc filter.

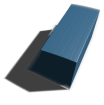
2.1.2 Texture Maps

A common example of a signal stored in a sample-based representation following these principles in computer graphics are *texture maps* [Cat74]. These regular 2D sampling grids store high-resolution information of material properties such as colors and reflectance [BN76] as well as small-scale surface information like small bumps and scratches [Bli78]. During rendering, this information is projected onto the geometry surface following a specified *mapping*—usually achieved by tangent space texture coordinates given at vertices and interpolated inside the respective triangle—and is considered in the evaluation of the bidirectional reflectance distribution function (BRDF) during shading (Fig. 2.3). Texture mapping is a widely spread approach to enhance a rendered surface with fine details without the need for a geometry of higher resolution, and we will review related work in this context in section 3.1.

2.1.3 Voxel Grids

3D *voxel grids* are conceptually similar to 2D textures, yet store surface information in a full volumetric Cartesian grid. In contrast to textures, this sampling grid does not necessarily rely on a coarse geometry representation and an according parameterization thereof. Instead, the surface can be stored along with all required properties—such as color, transparency, reflectance and various BRDF parameters depending on the lighting model—at a uniform resolution.

Conceptually, this thesis distinguishes between two different classes of voxel data (Fig. 2.4):



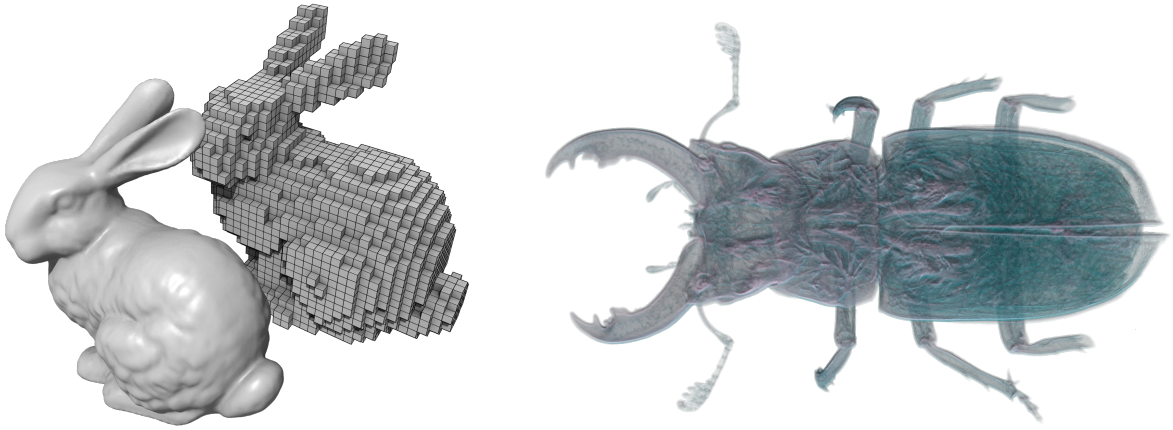


Figure 2.4: Binary voxelization of a triangle model (left) and a volumetric data set obtained from a medical CT scan (right).

- **Volumetric:** The voxel grid represents a dense, volumetric *distribution* of arbitrary quantities, e.g. density, color or a distance to a surface. This kind of data is often obtained from medical scans.
- **Binary:** Each voxel is an opaque element with a binary state indicating the presence or absence of a surface point. Thus, the voxel grid defines a solid surface.

Depending on the stored data, both classes can also be converted into each other. Given a function $f : \mathbb{R}^3 \to \mathbb{R}$, an *isosurface* to the *isovalue* c is defined as the set of points $\vec{x} \in \mathbb{R}^3 : f(\vec{x}) = c$. In a discretized volumetric domain, the isosurface is usually only present at interpolated values and, thus, can only be contained inside voxels that exhibit a sign change of $f(x) - c$ along at least one dimension if $f(x)$ is evaluated at the discrete neighbor points. The *cuberille* approach [HL79] extracts these voxels and retrieves a binary voxelization of this according level set. More sophisticated algorithms, like marching cubes [LC87], are able to extract a smooth triangle representation by fitting planes inside each voxel classified as part of the isosurface depending on the values at the neighboring voxels.

Vice versa, quantities given only along a surface as a set of binary voxels can be distributed into the surrounding area or downsampled to lower resolutions by 3D Gaussian or similar filters. The position of the surface itself can be converted into a distance field [BA02] which stores the distance to the surface at each point inside the voxel grid. From this grid, the 0-isosurface can then be visualized by direct volume ray-casting.

While both data types can be stored in the same 3D data structure—and can, in fact, be treated equally if a binary voxelization is interpreted as a volumetric density distribution of only 0 and 1 valued voxels—they are handled differently with respect to two aspects in our system:

1. **Rendering.** Volumetric data is visualized by marching a ray through the volume and, by sampling in fixed intervals along the ray, either accumulating $RGB\alpha$ values (as performed in direct volume

rendering) or searching for an isosurface. During sampling, the gathered values are interpolated between adjacent vertices of the voxel grid. The used color values may either be directly stored at the grid positions or, more commonly, derived from a *transfer function* (see section 2.4) that maps 1D scalar values to $RGB\alpha$ values. In the same spirit, normals required for shading are usually calculated as the gradient of the density function at each sampling point, e.g. by the use of central differences. For a function $f : \mathbb{R} \to \mathbb{R}$, the analytic gradient f' at a point $x \in \mathbb{R}$ can be approximated by

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2.2}$$

with a small value $h \in \mathbb{R} > 0$, typically of the size of one voxel if trilinear interpolation is employed. This approximation can be extended to functions of higher-dimensional input values by applying Eq. 2.2 in all dimensions.

Binary data sets do not contain opacity information, and each voxels' boundaries are checked for an intersection with a ray during rendering, resulting in a blocky appearance of the surface if the data set is oversampled—conceptually, the data is sampled with a box filter instead of a tent filter as commonly used for trilinear interpolation. As high quality gradients cannot be derived from binary data with reasonable computational effort, normals are explicitly stored at each voxel if required, along with additional shading information such as colors or reflectivity. This data is in general not interpolated. This difference is depicted in Fig. 2.5.

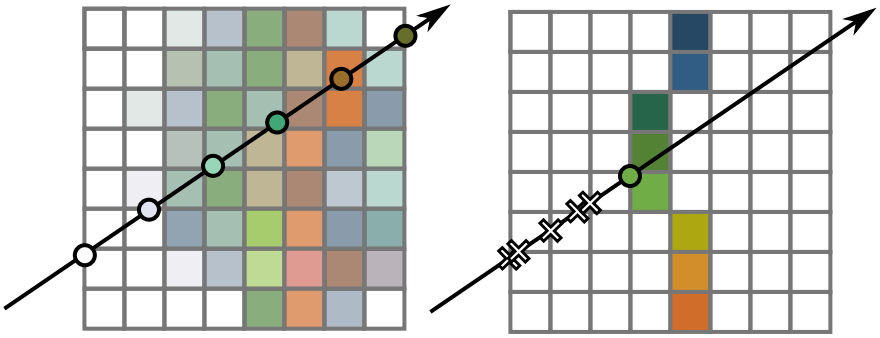
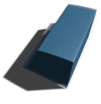


Figure 2.5: Fixed-step DVR ray-marching through volumetric data (left) vs. DDA ray-marching through a binary voxel grid (right). The left ray accumulates semi-transparent samples in fixed intervals, while the right ray leaps along the voxel boundaries until a non-empty cell is hit, where it returns the stored color and terminates.

2. **Data Sparsity.** Acceleration structures (section 2.3.1) exploit the fact that renderable data is *sparse*, i.e. only present in a subset of the complete 3D domain. Binary voxelizations are in general sparse data sets as they contain surface data, similar to triangle representations. Volumetric data sets, on the other hand, potentially contain information at each vertex of the sampling grid. The definition of "empty space" inside such a data set is dependent on the visualization method and



parameters. In isosurface ray-casting, all parts of the domain that do not contain the visualized isovalue can be classified as empty space; if direct volume rendering with transfer functions is performed, this generalizes to areas that lie either outside of the range of the transfer function, or contain only values which are mapped to colors of 0 opacity. If the isovalue or transfer function can change during visualization, the acceleration structure must either be rebuilt, or must contain additional information to adapt to the new parameters. For example, each node can contain a histogram of its data to drive the decision whether or not a sub-tree of the structure can be skipped.

It is important to note that both classes of voxel data are not completely orthogonal: The [Cra11] GigaVoxels system, for example, builds upon volumetric voxelizations, but targets data acquired from triangle meshes along with colors and material properties. Thus, large parts of the domain are pre-classified as empty space with data only present in a small shell around the initial surface.

2.1.4 Voxelization Fundamentals

If the source data is present in the form of a triangle surface, the conversion into binary voxel data is referred to as *voxelization*. This process has been intensely studied for years, and various algorithms have been proposed especially for the voxelization of triangle meshes. Depending on the application and source data, the demands to be met by the resulting 3D grid may vary.

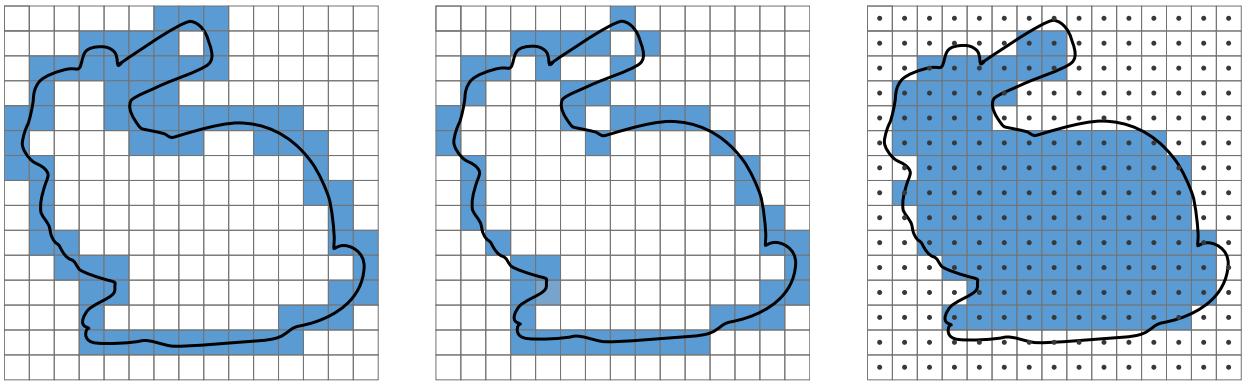


Figure 2.6: Binary voxelizations simplified to 2D: Conservative (left), 6-separating (middle), solid (right).

Binary voxelizations exhibit different properties defined on the basis of guaranteed connectivity of the generated voxels along a surface as exemplary depicted in Fig. 2.6. Two voxels are defined as *adjacent* if they share a face, an edge, or a vertex [COK95]. Each voxel has a maximum of 26 adjacent voxels in 3D, thus any two voxels that exhibit at least one of these three properties are termed *26-adjacent*. Subsequently, voxels sharing a vertex, edge, or face are *8-*, *12-*, and *6-adjacent*. A set of voxels is called *N-connected*, if for every pair of voxels a path exists in which all consecutive pairs of voxels are *N-adjacent*. Let X be a set of voxels and S^d a subset in X representing a voxelization of a continuous

surface. If $X - S^d$ is not N-connected, S^d is termed $N - separating$.
Depending on the application, a voxelization may be required to exhibit a certain guaranteed degree of separation, most commonly 6-separation. Early fast voxelization techniques [FC00, LFWK03] rely on rasterization and cannot easily guarantee any degree of separation. Later approaches exploit conservative rasterization and explicit voxel/triangle intersection tests in GPU compute mode [ZEP07, SS10, Pan11] to generate 6-separating or *conservative* voxelizations, where each voxel touched by a surface is included in the voxelization. This property is often required in simulation domains. In addition, binary voxelizations can be *solid* and, thus, classify the inside and outside of an object. Solid voxelizations in general require watertight input meshes that exhibit no holes and, consequently, have a well-defined inside region.

2.2 Level of Detail

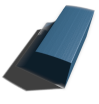
The final step in rendering consists of a resampling of the rendered data onto a regular 2D grid. As already introduced, minification during resampling introduces aliasing-artifacts due to the presence of frequencies above the Nyquist rate. Applying a low-pass filter to the original data beforehand allows the low-quality filter used during resampling to more closely resemble the optimal sinc filter and alleviates the aliasing problem.
This low-pass filtering is most commonly performed in a preprocess by generating several coarser representations of the source data and storing those as several LoDs. Naturally, the storage memory requirements for such low-pass filtered information are usually greatly reduced as only a considerably smaller amount of the input frequencies are required for later reconstruction.

2.2.1 Screen Space Error Metric

Most applications in computer graphics render using a *perspective projection* (compare Fig. 1.4) in order to model the behavior of the human visual system. In contrast to an orthogonal projection as used in engineering detail drawings, the projected size of a displayed object decreases with increasing distance from the center of projection, i.e. the position of the virtual camera. Given a final resampling resolution of h screen pixels along the y-axis and a virtual camera with a vertical field of view of θ, Cohen [COM98] derives a world space error tolerance at a point P with distance P_z to the center of projection as

$$\epsilon(P) = \frac{2}{h} \cdot \tan\frac{\theta}{2} \cdot \tau \cdot P_z. \tag{2.3}$$

For a screen-space error below a selected threshold τ, we can then select a corresponding level of detail l for a displayed object. The metric in Eq. 2.3 is strictly increasing in depth and, therefore, can be



evaluated for the point P' of the object with closest distance to the camera for a conservative estimation of the induced screen space error. Assuming each level l was generated such that a maximum error ϵ_l in world space is ensured, the object's representation at this level can be used in rendering iff

$$\epsilon_l \leq \epsilon(P') < \epsilon_{l-1} \tag{2.4}$$

is fulfilled, assuming level 0 denotes the topmost, coarsest level of the hierarchy. The desired screen-space error can be varied for a visualization system to balance between memory consumption, rendering speed and image quality. It is usually given in units of pixels and is referred to as *pixel error*.

2.2.2 Voxel LoD

Section 1.3 already derived the need for a LoD representation from technical limitations and highlighted the shortcomings of rasterization-based systems. When dealing with regularly sampled data such as textures or voxel grids, LoD creation and selection is greatly simplified. For a grid of samples with distance d from each other along all dimensions, a sample represents data inside its *voronoi neighborhood*, which is a cube of side length d in the interval of $[-\frac{d}{2}, \frac{d}{2})$ around the grid vertex location. For a level l, the sampling distance d_l then corresponds directly to the maximum world space error induced in creating this level. Creation of the next coarser level $l-1$ from a given level l is reduced to a regular resampling of the data with increased sampling distance.
In theory, the sampling resolutions of two successive levels can be completely unrelated as long as they are increasing along with the level index. However, resampling by using a perfect reconstruction filter is impractical for all but the smallest datasets, even if the LoD creation is performed in an offline preprocess as it is typically the case. In practice, the sampling distance d_{l-1} at level $l-1$ is usually chosen as $d_l \cdot n, n \in \mathbb{N}$ with $n > 1$ and less expensive filters of limited support are used.

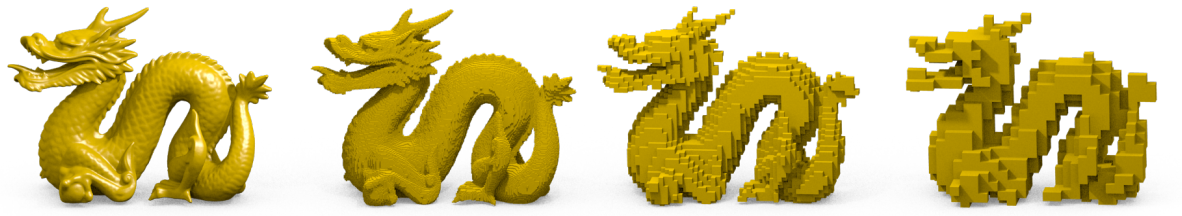


Figure 2.7: Voxelizations in different levels of detail: Original surface, 256^3, 64^3, 32^3 (from left to right).

For binary surface data, a *maximum filter* creates a conservative surface representation by assigning a voxel as part of the surface if at least one voxel in the filter's support contains the surface (Fig. 2.7). Other common filters are a simple box- or more expensive Gaussian filters for data such as colors or density distributions in volumetric voxelizations (Fig. 2.8). For normals, specialized filters have been

proposed based on a *normal distribution function* to preserve the appearance of a surface in the presence of a complex lighting interactions. A recent overview on this topic can be found in [BN12].

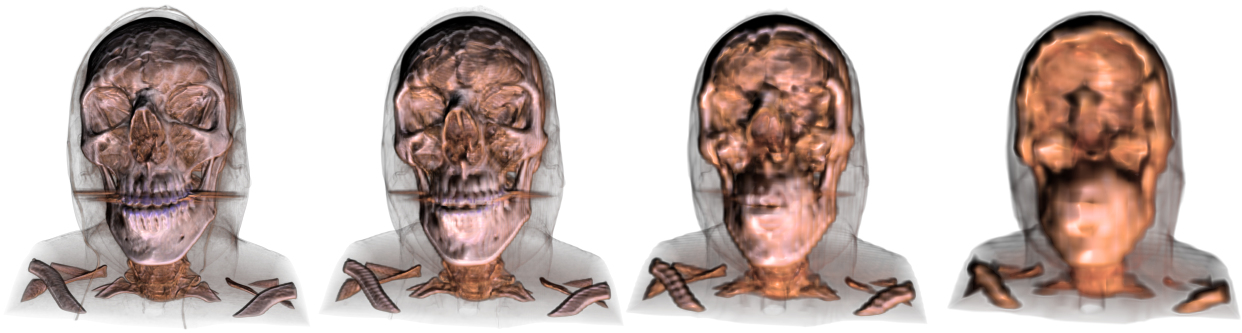


Figure 2.8: Different levels of detail of a volumetric voxelization: 256^3, 128^3, 64^3, 32^3.

One important core observation is that enforcing a pixel error of 1.0 or less eliminates oversampling in the final resampling step. Thus, the geometric error introduced by rendering discrete cubical elements from a binary voxelization is guaranteed to be at most one pixel on screen. This fact is illustrated in Fig. 2.9, where an isosurface is rendered once with isosurface ray-marching including trilinear interpolation, and once from a binary voxelization including precalculated surface normals for high-quality shading at little visual difference. Only in the rightmost image, where the pixel error was set to 2.0 pixels, blocky artifacts resulting from oversampling of the surface become slightly visible.

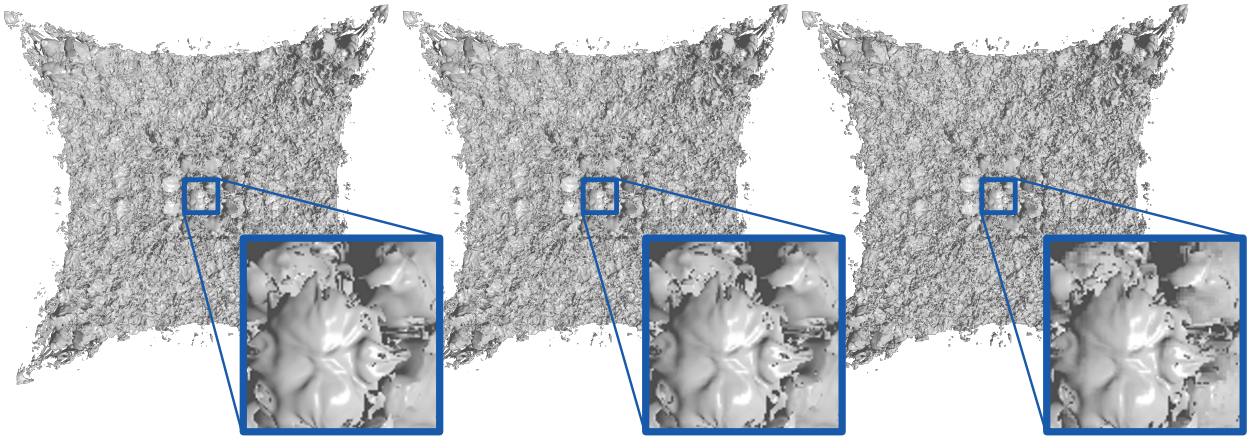
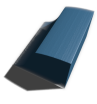


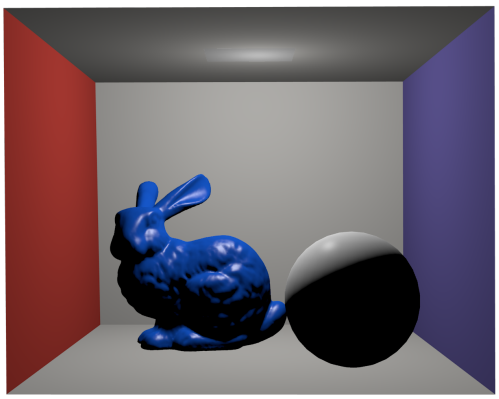
Figure 2.9: An isosurface visualized using direct volume ray-casting on the original resolution (left). In the middle image, only surface voxels where stored as a binary voxelization, which is then rendered as small cubes using DDA ray-marching. Right shows the next coarser LoD of the binary voxelization.

2.3 Ray-Tracing

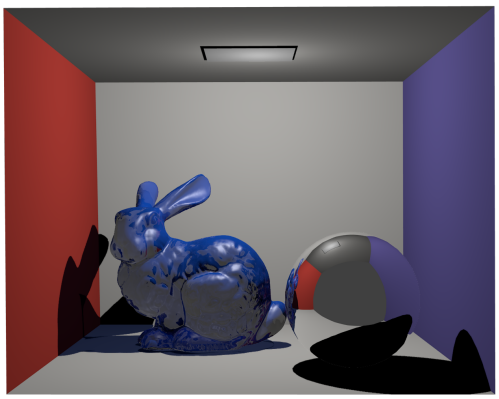
In contrast to Rasterization, *ray-tracing* [App68] is an *image-order* approach. Rasterization, as an *object-order* approach, produces an image by processing each primitive of the scene and determining



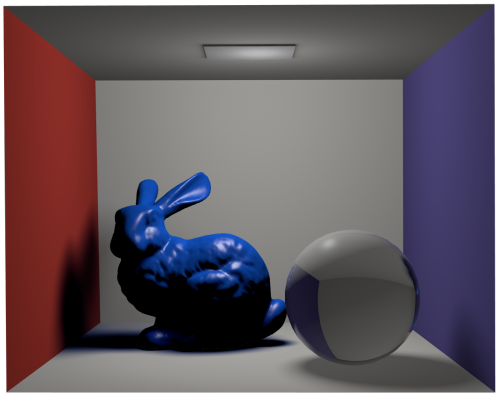
the covered pixels on the screen. Ray-tracing, on the other hand, processes each pixel of the screen and determines the triangles visible under the area of the pixel. We speak of *ray-casting* to emphasize the focus on primary rays cast from the camera into the scene [App68]. The more complex whitted ray-tracing [Whi80] allows multiple levels of recursion of each ray to trace reflections, refractions and hard shadows inside a scene. Distributed ray-tracing [CPC84] takes this concept a step further and spawns multiple rays at each intersection to account for e.g. diffuse shadows from an area light source. The most advanced algorithm, path-tracing [KK86], finally is able to compute full global illumination by tracing paths from diffuse reflections through the scene. On the down side, however, the latter two—and path tracing in particular—require a significant number of rays shot per pixel to achieve noise-free images, since the directions of reflected rays is determined randomly. Figure 2.10 presents the effects achieved with each of those methods.



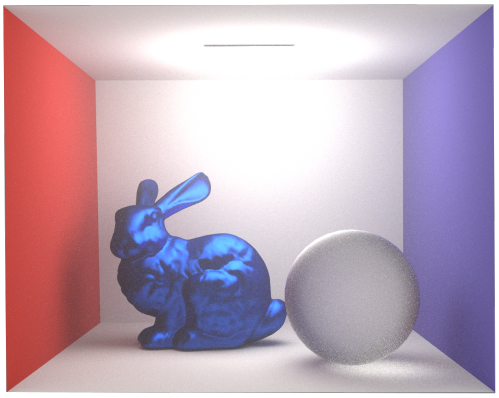
(a) Primary rays only (ray-casting).



(b) Whitted-style ray-tracing with hard shadows.



(c) Distributed ray-tracing with soft shadows.



(d) Path tracing with full global illumination.

Figure 2.10: Comparison of ray-tracing algorithms.

Ray-tracing is fundamentally a search algorithm that tries to answer the following question: Given a ray, where does the first intersection with a primitive occur? While this question can be answered in a brute-force approach by testing all primitives in the scene, sophisticated *acceleration structures* are

required to deliver the answer in a reasonable time for all but the most simple scenes.

In the last years, ray-tracing has been increasingly popular especially in offline rendering due to its straightforward support of physically correct secondary effects and the limitations of the current, rasterization-based pipeline (see also section 1.3). For massive scenes, ray-tracing offers superior support for fine-grained visibility queries [DHW*11] to greatly reduce bandwidth and memory requirements as well as rendering time. Concurrently, significant advances in ray-tracing systems and algorithms [PBD*10, DHW*11] denote the technique as a rival to rasterization also in interactive and GPU-driven applications.

As this thesis' main concern is to provide new, memory-efficient data structures, a complete coverage of the topic would go beyond its scope. For an excellent overview of triangle-based ray-tracing and an in-depth discussion of acceleration structures, we refer to [PH10, Sam06]. In the following, we will give a brief overview with a focus on GPU ray-tracing and technological choices derived for this thesis.

2.3.1 Acceleration Structures

Decades of research have been spent on creating, traversing, and optimizing acceleration structures. Thus, a number of various approaches exist, which can in general be divided into two classes: spatial subdivision techniques and object hierarchies [WMG*09].

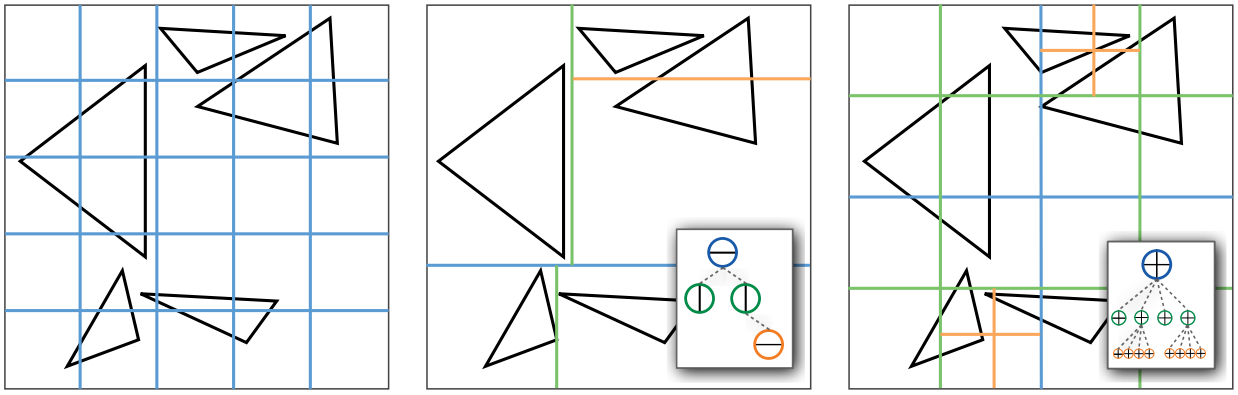
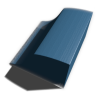


Figure 2.11: Examples of spatial subdivision structures, simplified to 2D: Regular grid (left), kd-tree (mid) and quadtree (right).

Spatial subdivision structures provide a unique mapping from spatial locations to subdivision cells, whereas an object may be contained in multiple cells. Subdivision cells can never overlap, making traversal in a front-to-back order simple since once a hit is found, no further cells need to be tested ("early exit"). However, the same object may be tested multiple times during traversal if it overlaps more than a single cell. Spatial subdivision structures provide tight fits around an object at the cost of a high level of subdivision, shifting a huge portion of the work from triangle intersection to a large



number of fast and simple acceleration structure traversal steps. They are in general memory-friendly as children can often be expressed by simple indices in relation to their parent. Popular examples are regular grids, kd-trees, quadtrees [Sam84] and octrees [Mea82] (2.11).
Spatial subdivision schemes may be adaptive to the scenes' geometry: for example, kd-trees use subdivision planes whose location is chosen heuristically based on the distribution of primitives in the subdivision space. Other structures, like octrees and quadtrees, follow a fixed subdivision scheme regardless of the contained geometry. This simplifies the build-process and allows for an easy parallelization, but results in an increased amount of empty space inside some subdivision cells, making the traversal less efficient in theory. Note that, even though a structure is considered to be non-adaptive regarding the placement of its subdivision planes, the desired finest subdivision level may very well differ across the scene depending on the contained geometry (compare Fig. 2.11, right).

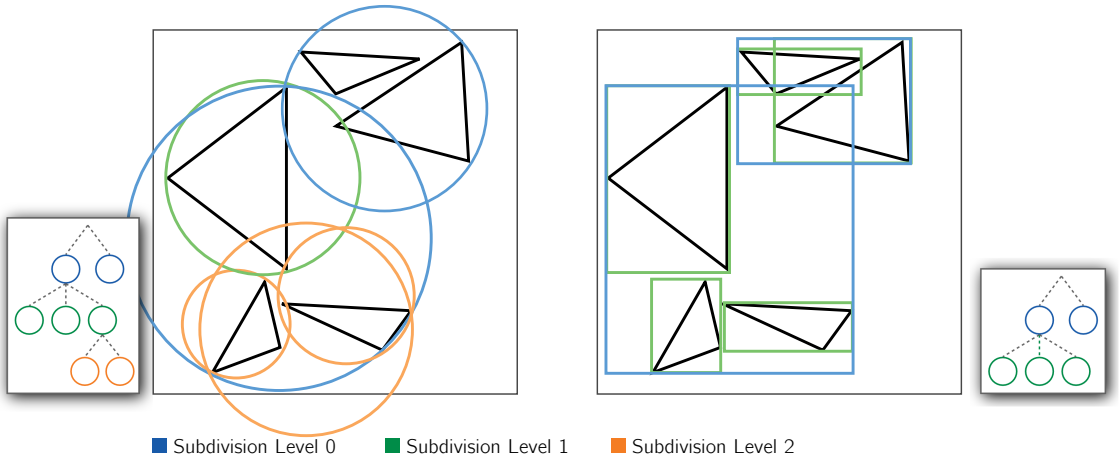


Figure 2.12: Bounding volume hierarchies using spheres (left) and axis-aligned boxes (right)

Object hierarchies provide a unique mapping from objects to subdivision cells, but multiple subdivision cells may overlap. This makes traversal more challenging and expensive: when a hit is found, the remaining overlapping cells still need to be tested as they may contain another hit closer to the camera. On the other hand, each object needs to be tested exactly once, and the number of subdivision cells—and, therefore, traversal steps—is often significantly reduced in comparison to spatial subdivision structures. Consequently, the memory consumption of an object hierarchy is not necessarily higher than a corresponding spatial subdivision, even though each individual cell comes at a higher storage cost. A typical and widely used object hierarchy are bounding volume hierarchies (Fig. 2.12).

In both cases, a core characteristic of the structure is the use of axis-aligned versus arbitrary split planes [WMG*09]. While the latter usually provide a much tighter geometric fit, axis-aligned planes greatly improve the simplicity, speed and robustness of the traversal and can usually be represented in a more compact way—a plane's orientation can be expressed in two bits instead of several floating point numbers. They are also simpler and faster to build and, therefore, widely used in current state of the art systems.

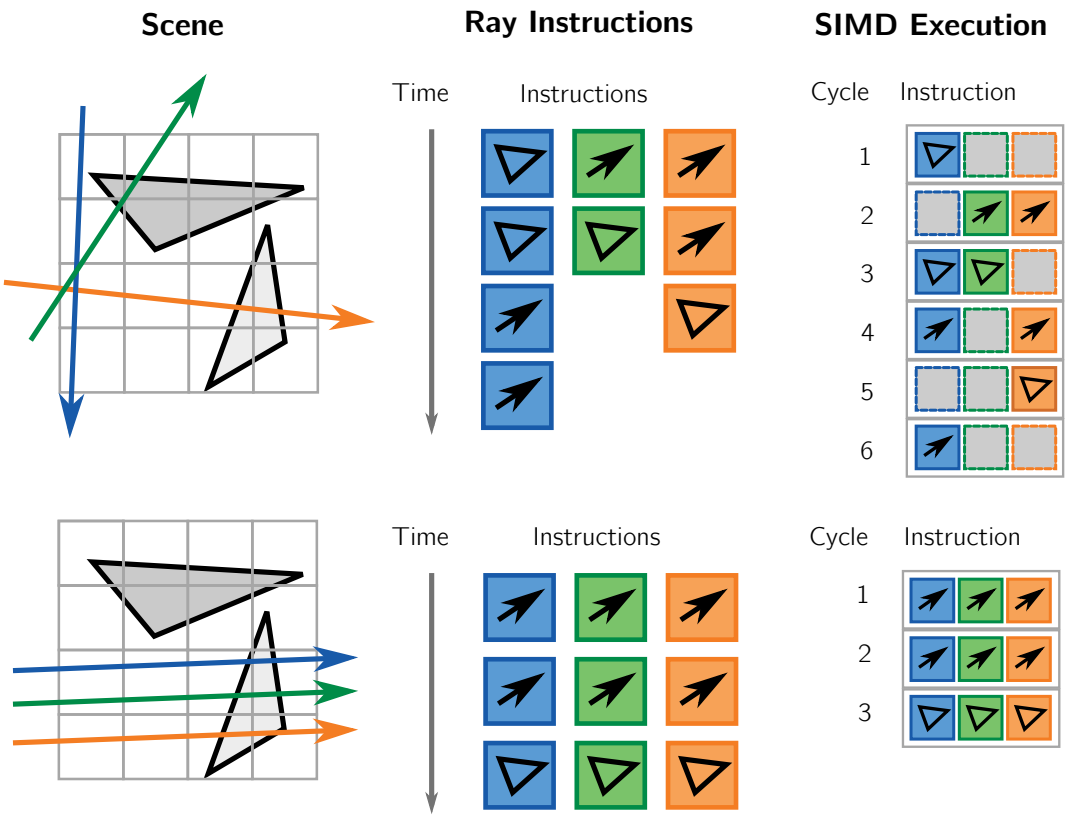


Figure 2.13: Examples of SIMD coherency: Two different sets of rays traverse the same scene with a regular grid acceleration structure. Given two instructions `advance` (symbolized by an arrow) and `intersect` (symbolized by a triangle), the above set of rays requires significantly more cycles to find the first hit for each ray when mapped to a single SIMD lane, even though both scenarios require a total of 9 operations.

2.3.2 SIMD considerations

Single instruction, multiple data (SIMD) processors such as GPUs (for more details, see 2.5) execute a single instruction concurrently on a vector of arguments (a *SIMD lane*). In the case of ray-tracing, this is generally mapped to a fixed number of rays traced in parallel with the same instructions performed on each ray. The performance is, thus, dependent on the number of rays inside each SIMD lane that require the same instruction in each execution cycle, called the *instruction coherency*. The impact of an SIMD model is schematically depicted in Fig. 2.13, where two sets of rays traverse a scene using a regular grid acceleration structure. We assume an architecture with a SIMD-width of 3 that offers the instructions `advance` to advance the ray to the next grid cell, and `intersect` to intersect all primitives inside a cell and then advance to the next cell. While both sets of rays require the same number of operations in total to find the first intersection or leave the scene, the top set requires twice the number of cycles when processed in a single SIMD lane in comparison to the bottom set.

Along with several other complex factors in place—such as *memory coherency* with respect to the data

required by each ray, and several levels of caching throughout the memory hierarchy of a processor—this fact makes estimation of ray-tracing performance a difficult and highly scene- and hardware-dependent task. Accordingly, the construction of an "optimal" acceleration structure for a scene is considered an NP-hard problem [Hav00, PGDS09]. Several works exist that examine this behavior for various scenes, hierarchies and split metrics [AL09, AKL13] along with a vast body of literature regarding the optimization of ray-traversal of various acceleration structures for SIMD-processors, e.g. [BEL*07, GPSS07, WIK*06, BAM14]. Nowadays, both kd-trees as well as bounding volume hierarchies are most widely used in triangle ray-casting and have both become competing gold standards.

2.3.3 Integration with Level of Detail

As all acceleration structures except the flat grid are organized in a hierarchy, a combination with multiple levels of detail inside the inner nodes is possible. Especially in the case of octrees this is a trivial and straightforward extension, which has made them the prime—and so far only—choice in voxel-based ray-tracing in the last years. For triangle data, however, several problems arise aside from the initial construction of coarser representations, which render true multilevel hierarchies rarely used in practice. Usually, LoDs are generated from higher-order surface patches which are either tessellated into triangles in each frame, or which are directly intersected with each ray. This degenerates the problem to a case of dynamic geometry with changes in between frames [WMG*09], but restricts the LoD selection to a per-view basis for all (including secondary) rays.

In the rare case of the *Razor* system [DHW*11], multiresolution surfaces are a core part of the system design and the desired resolution is selected on a per-ray rather than a per-frame basis to also account for secondary rays from multiple bounces. Intermixing different levels of detail for a single ray implicates a number of problems on the system, most prominent LoD popping, cracks between cells and tunneling (Fig. 2.14). Djeu et al. [DHW*11] solve this by geomorphing between discrete LoDs, resulting in a smooth interpolation of different levels.

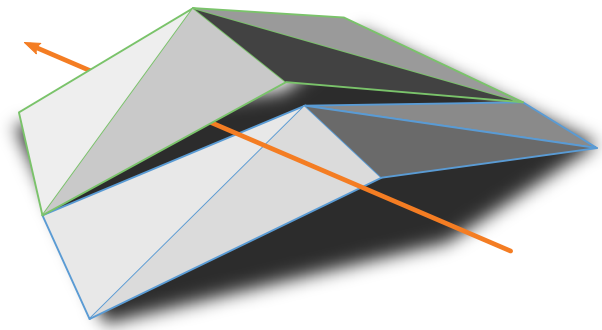


Figure 2.14: A change of LoD (color-coded as blue and green triangles) along a ray may lead to completely missing the surface (tunneling).

2.3.4 Acceleration Structures in Voxel Ray-Tracing

Changing the underlying primitive type from triangles to voxels simplifies the layout of the various acceleration structures. The data is now given in cubical nodes of fixed extent and discrete positions. Thus, the chosen acceleration structure will naturally inherit these properties by using axis-aligned split

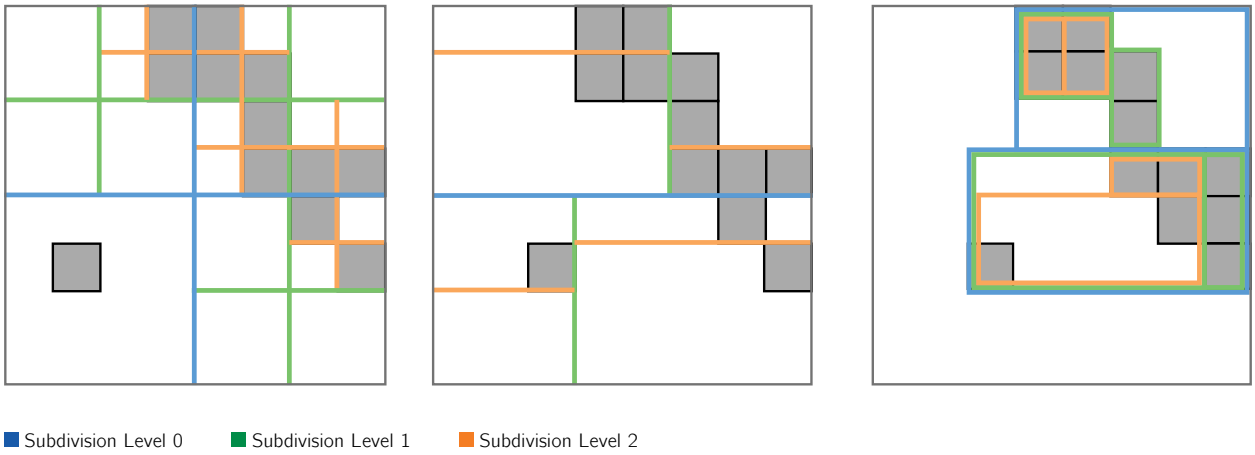


Figure 2.15: Acceleration structures in a binary voxelization, simplified to 2D: Quadtree (left), kd-tree (middle), bounding volume hierarchy of axis-aligned boxes (right).

planes, whose placement is constrained to voxel borders. This results in a one-to-one mapping of spatial locations to subdivision cells and degenerates object partitioning schemes into space partitioning schemes (see Fig. 2.15). Given a fine enough, but still non-infinite level, each acceleration structure at some point reaches a tight fit which exactly represents the scene's geometry, even though the resulting tree depths may vary.

Voxel ray-casting is a perfect fit for GPUs and systems with relatively wide SIMD units in general. Aila et al. [AKL13] observe that, in the context of triangle ray-tracing, the overlaps between nodes in a bounding volume hierarchy has major impact on the traversal performance. They derive a metric to measure the end-point overlap average *EPO* and use its minimization as an additional split heuristic criterion in a BVH builder, along with a split-area heuristic *SAH*. Their builder incorporates both metrics with a modifiable parameter α as

$$(1-\alpha)SAH + \alpha EPO; \qquad 0 \leq \alpha \leq 1. \tag{2.5}$$

For complex scenes, they find α to be as high as 0.98 with an average of 0.59, meaning the performance is largely—and in some scenes nearly exclusively—limited by the amount of end-point overlaps. Voxel data, on the other hand, exhibits an EPO of 0 in all cases. Thus, the ray-caster has to perform less work in the leaf nodes as no resolution of overlapping primitives is required at all. Overall, the rendering performance is mainly bounded by acceleration structure traversal and is expected to perform better on systems with wide SIMD-units like GPUs. [CW14] provide an exhaustive comparison of various acceleration structures for voxel ray-tracing in multiple scenarios and conclude that kd-trees, octrees as well as BVHs are competitive and on average exhibit performance differences between 15% and 30%. If larger bricks of potentially more complex and compressed data structures are used in the leaves, as targeted by our framework, these differences are expected to be smaller as more work will be performed

during traversing these structures in the leaves.

Finally, the combination of binary voxel LoDs and ray-tracing acceleration structures is less painful than it is the case with triangles. Cracks and tunneling artifacts are completely avoided if LoDs are built conservatively bottom-up, and can be easily detected in all other possible cases.

2.4 Direct Volume Rendering

Many basic concepts of this thesis originate from direct volume rendering (DVR) research and applications. The presented framework lends itself well to standard volume rendering, as we will demonstrate in chapter 5. This section introduces the basic principles and theories of volume rendering for readers not familiar with the concept. For an excellent in-depth coverage of advanced effects, we refer to the tutorial by Engel et al. [Eng06], which also served as a basis for the following overview.

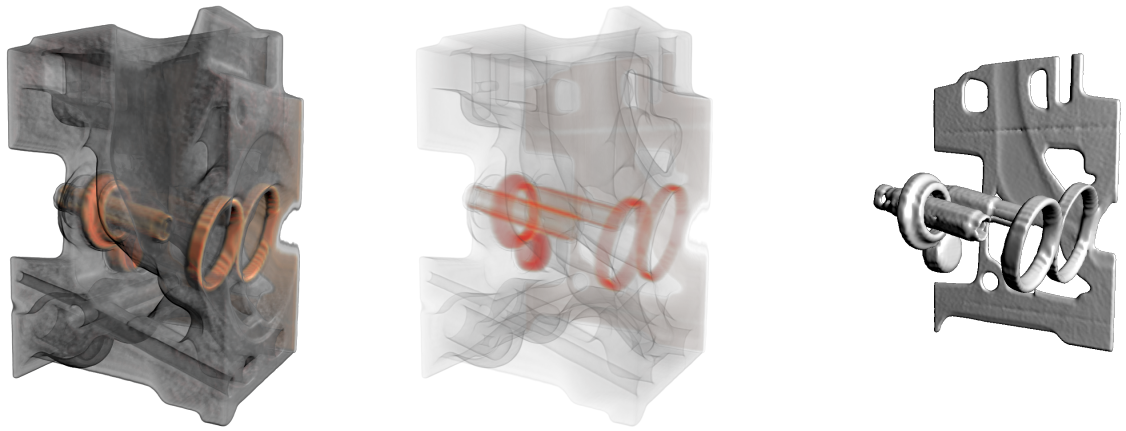


Figure 2.16: Examples of direct volume rendering: the same data set is visualized with different transfer functions (left, middle) and direct isosurface ray-casting (right).

2.4.1 The Volume Rendering Integral

Conceptually, volume rendering techniques visualize data given as a continuous three-dimensional signal $f(\vec{x}) : \mathbb{R}^3 \to \mathbb{R}$. In practice, this data is stored at the vertices of a three-dimensional voxel grid following the introduced principles of sampling, with values between the sampling points obtained by interpolation. *Indirect* rendering algorithms work by extracting a surface from the data [LC87] which is then rendered e.g. using standard rasterization.

In contrast, *direct* methods evaluate the light transport inside the 3D volume based on an optical model to determine the resulting color visible through a screen pixel. Optical models consider the various physical effects present in a non-homogeneous medium: absorption, emission, and scattering. For the

underlying theory of light transport and details on optical models, we refer to [CZ67, Max95].
Mapping from the stored scalar values to $RGB\alpha$ colors is performed by *transfer functions*. These functions are generally user-defined and try to associate a human-understandable meaning with the abstract values. They are, thus, highly dependent on the type of data and the desired information to be extracted from it. This process is referred to as *classification* (Fig. 2.16).

In the following, we will assume a widely used emission-absorption model, which accounts for light absorption as well as self-emission inside the volume, but neglects the more complex effects resulting from light scattering (such as indirect illumination and shadowing). For an illustration, see Fig. 2.17.

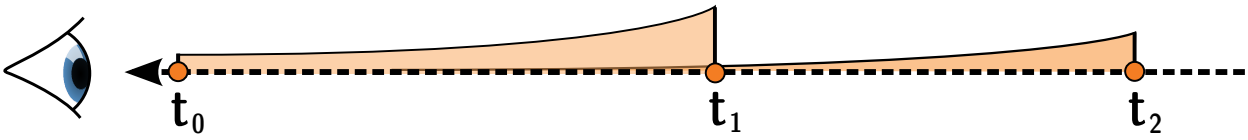


Figure 2.17: Partial absorption of radiance emitted along a ray at ray distances t_1 and t_2.

To determine the final color visible along a ray cast into the 3D volume (denoted with $\vec{r}(t)$, where the parameter $t \in \mathbb{R}$ is the distance from the eye), the *volume rendering integral* is evaluated along the ray. At each point $\vec{x} \in \mathbb{R}$ inside the volume, an *absorption coefficient* $\kappa(f(\vec{x}))$ as well as the *emissive color* $c(f(\vec{x}))$ is derived from the stored scalar value. Given the ray origin and direction, the position $\vec{x}$ corresponding to a point along the ray $\vec{r}(t)$ can be calculated. We introduce a function $s : \mathbb{R} \to \mathbb{R}$ that maps the ray parameter t to the scalar value resulting from the evaluation of f at the position corresponding to point along the ray at distance t. Following [Eng06], we denote both parts of the used optical model as functions of the ray distance for simplification:

$$c(t) := c(s(\vec{r}(t))) \tag{2.6}$$

$$\kappa(t) := \kappa(s(\vec{r}(t))) \tag{2.7}$$

The total absorption $\hat{\kappa}$ in an interval $[t_1, t_2]$ along a ray is determined by integrating over the varying absorption coefficients inside the interval:

$$\chi(t_1, t_2) = e^{-\tau(t_1, t_2)} \tag{2.8}$$

with

$$\tau(t_1, t_2) = \int_{t_1}^{t_2} \kappa(t) dt \tag{2.9}$$

called the *optical depth*.

The final color is then computed as the integral over the emission terms along the ray, taking into account the amount of absorption that is applied to the emitted light at each position depending on the optical depth:

$$C = \int_{t_0}^{t_e} c(t) \cdot e^{-\tau(t_0,t)} dt \tag{2.10}$$

The integral starts at the position where the ray first enters the volume at parameter t_0 until its exit at t_e.

2.4.2 Discretization

Since an analytical solution of Eq. 2.10 is not possible in practice, the solution is approximated by discretization of the sampling points along the ray and replacement of the integrals with Riemann sums [Eng06]. The corresponding optical properties for each scalar value are retrieved from a lookup table determined by the current transfer function, yielding a single $RGB\alpha$ value.

Assume we take n samples along a ray, derived from a fixed ray sampling distance of Δt as $n = \left\lfloor \frac{(d_e - d_0)}{\Delta t} \right\rfloor$. In the i-th interval, the emissive color is approximated by weighting the color retrieved by the transfer function at the start of the interval with the interval length:

$$C_i = c(i \cdot \Delta t)\Delta t \tag{2.11}$$

.

Accordingly, the amount of absorption inside the i-th interval is given by

$$\tilde{\chi}(i) = e^{-\tilde{\tau}(i)} \tag{2.12}$$

with a discretized total absorption along a ray up to the i-th interval of

$$\tilde{\tau}(i) = \sum_{j=0}^{i-1} \kappa(j \cdot \Delta t)\Delta t \tag{2.13}$$

.

Replacing the summation in the exponent by a multiplication of exponents yields

$$\tilde{\chi}(i) = \prod_{j=0}^{i-1} e^{-\kappa(j \cdot \Delta t)\Delta t} \tag{2.14}$$

.

In addition, the discretized absorption coefficient is usually expressed through an opacity value $\alpha_i = 1 - e^{-\kappa(j \cdot \Delta t)\Delta t}$. The resulting discretized volume rendering integral 2.10 then computes as

$$\tilde{C} = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - \alpha_i) \tag{2.15}$$

.

During ray-casting, the equation is usually computed in front-to-back order by using alpha blending. The accumulated color C'_i and opacity α'_i in step i then computes as

$$C'_i = C'_{i-1} + (1 - \alpha'_{i-1} C_i) \tag{2.16}$$
$$\alpha'_i = \alpha'_{i-1} + (1 - \alpha'_{i-1})\alpha_i \tag{2.17}$$

.

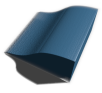
2.4.3 Rendering Techniques

Based on these derived equations, visualization of volume data sets can nowadays be performed using one of three popular techniques.

Ray-Casting:
Ray-casting can be employed to iteratively evaluate Eqs. 2.16 and 2.17. Nowadays, it is the predominant algorithm for DVR on programmable graphics hardware due to its superior quality and simple implementation using 3D textures [KW03]. In a GPU shader with one thread per pixel, the ray is marched through the volume and fetches the scalar values from a 3D texture, utilizing hardware-supported trilinear interpolation between the discretized samples (compare Fig. 2.5 and the introduction to ray-casting in section 2.3). The quality and speed of the algorithm is dependent on the selected interval size: coarser intervals lead to approximation artifacts for varying emission- and absorption terms along the ray, whereas finer intervals result in a higher number of evaluations, pressuring both the GPUs compute units as well as the memory bus. Figure 2.18 shows a comparison between different sampling rates.

Splatting:
An approximation for early volume rendering is splatting the individual voxels onto the screen using a Gaussian reconstruction kernel [Wes90]. While this approach is simple and fast, it cannot compete with other methods in terms of quality of the reconstructed volume.

Texture Slicing:
In contrast to ray-casting, texture slicing is an object-order approach. Multiple slices through the volume are rendered by means of a proxy geometry and then blended together to form the final image



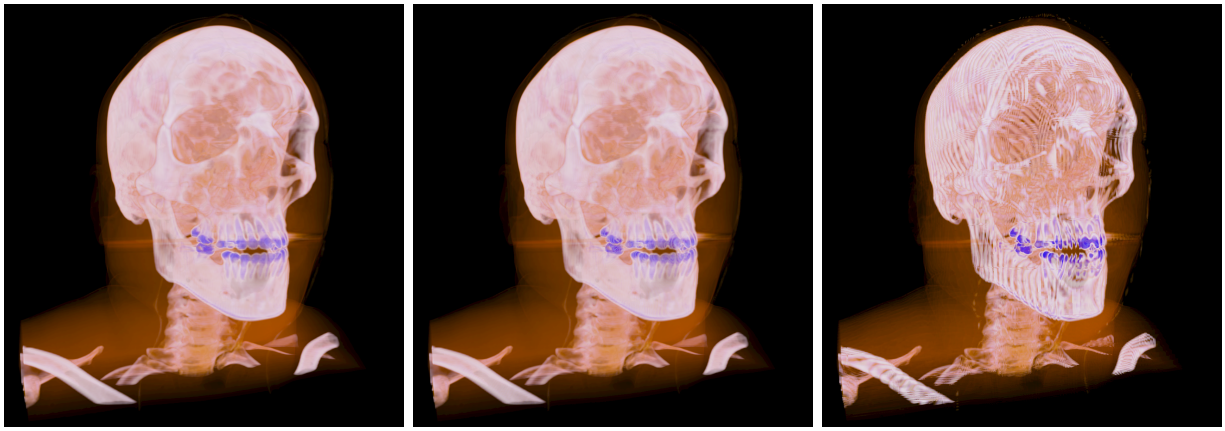


Figure 2.18: DVR quality comparison for sampling interval sizes of 0.1, 1.0 and 5.0 voxels.

[CN94, CCF95, LL94]. Most commonly, view-aligned slices (Fig. 2.19) are rendered. The scalar values are stored in a 3D texture and mapped according to the texture coordinates of the rendered slices. On non-programmable hardware, the look-up texture contains the colors resulting from a mapping of the current transfer function instead.

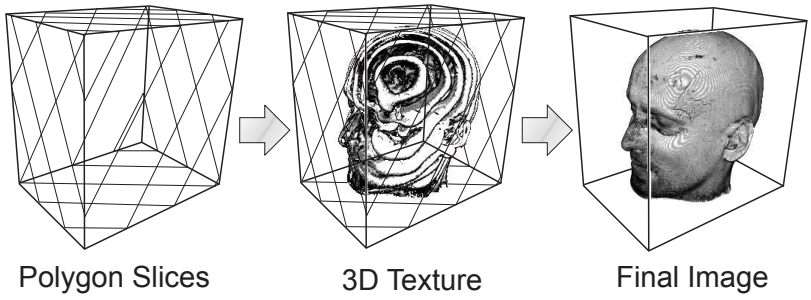


Figure 2.19: View-aligned slicing. Source: [Eng06].

On hardware without 3D texture support, axis-aligned slices provide an alternative (Fig. 2.20, top) using only a stack of 2D textures without interpolation between the slices. To support arbitrary viewing angles, three orthogonal stacks are stored in memory; during rendering, only the stack most parallel to the image plane is used. This leads to visible popping artifacts as the location of the samples changes abruptly when switching from one stack to another (Fig. 2.20, bottom).

Much work has been done on extending the speed, quality and capabilities of texture slicing on commodity PC graphics hardware [WE98, RSEB*00]. Even in recent years, slice-based volume rendering is a common choice, especially on mobile platforms with limited hardware support [Krü10]. The main drawback of the technique, however, remains: As it cannot benefit from empty-space skipping and occlusion culling as efficiently as ray-casting, its application for high-resolution data sets is limited by crucially high fill rate and bandwidth requirements.

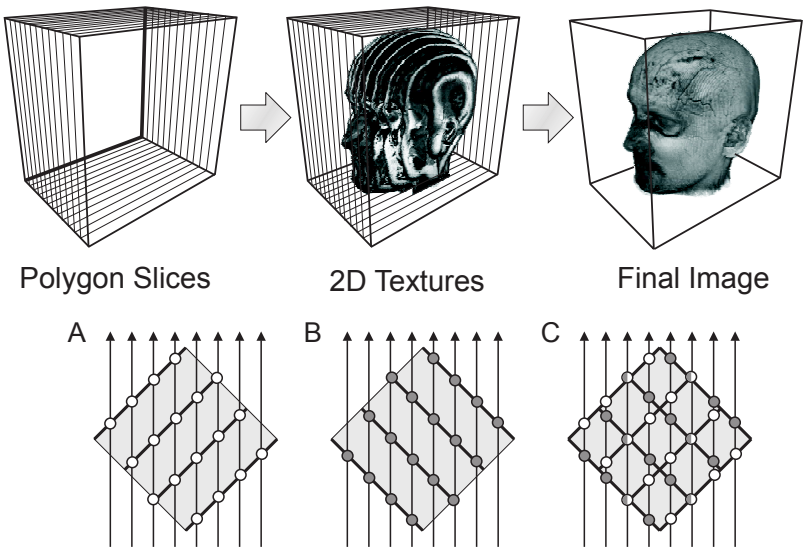


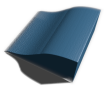
Figure 2.20: Top: Object-aligned slices in DVR. Bottom: Changing the rendered stack due to camera movement causes an abrupt change in the sampling pattern. Source: [Eng06]

2.5 GPU Architecture

Over the last decade, GPUs have evolved rapidly from pure rasterization devices with vertex transformation capabilities to fully programmable, general-purpose compute units. Nowadays, they are found in every consumer-level hardware from mobile devices to high-end workstations, and are successfully used in supercomputers to speed up the processing of problems which can be formulated in a data-parallel way. As this thesis heavily utilizes general-purpose GPU compute (*GPGPU*) algorithms, an overview over the current architecture, its capabilities, and limitations is given in the following. As the development of new hardware rapidly progresses, all details can only be a snapshot of the current state. These observations are based on NVIDIA's Kepler (GK110) architecture [NVI12], which is a direct advancement from NVIDIA's previous architecture (Fermi). For a more high-level overview on the general graphics and rendering pipeline, we refer the reader to [AMHH08].

2.5.1 GPU SIMD Model

GPUs execute all instructions in a SIMD manner: Each operation is executed in parallel on a vector of operands of fixed size. In GK110 and the previous NVIDIA architectures, the width of this SIMD vector (called the *warp* size) is 32, whereas it is 64 for AMD GPUs. GPUs are highly optimized to work on single-precision floating point data, but also contain ALUs for integer and double precision operations. GK110 consists of up to 15 processors (SMX), each with its own set of registers and a private L1 cache (Fig. 2.21, left). Each SMX in term contains 192 cores with their own ALUs working



on single floating point or integer operands and, thus, is capable of executing 6 warps in parallel (Fig. 2.21, right). Each SMX also contains its own set of schedulers to select the execution order of pending warps. A program—called *kernel*—is executed in multiple *threads*, which are in term grouped in *thread groups* of a user-defined size and internally scheduled to warps of 32 threads.

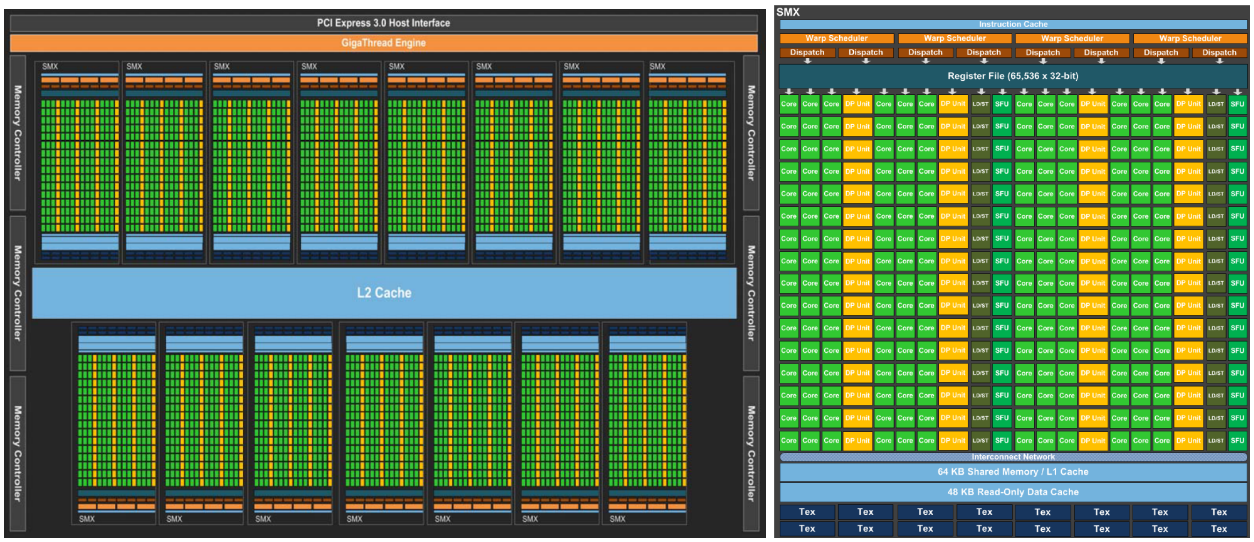


Figure 2.21: Left: SMX of GK110. Right: Cores inside a single SMX. Source: [NVI12]

2.5.2 Memory Regions

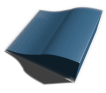
Like a CPU, a GPU contains a variable amount of random access memory, usually between 1 and 6 GB, which is referred to as *global memory* and can be accessed at high bandwidths of up to 288 GB / s (NVIDIA Geforce GTX Titan). This is a multiple of the bandwidth available on CPUs, even though recent generations have also significantly increased in this regard. However, memory access comes with serious latency if the requested data is not already residing in the L1 or the L2 cache shared between all processors.

Each SMX contains a small amount of *shared memory* which can be accessed by all threads of a thread group and allows for data sharing between threads. It is shared with the L1 cache and accordingly faster accessible than global memory. In addition, a high number of registers is available for single-cycle data access for all cores.

2.5.3 Asynchronous Execution

GPUs execute all kernels asynchronously: Once a kernel is dispatched by the CPU, the GPU is responsible for scheduling and executing it along with any other kernels previously dispatched that have not yet finished executing, leaving the CPU free to perform different tasks in the meantime. HyperQ [NVI12]

even allows the GPU to schedule threads dispatched from multiple CPUs in parallel.
This asynchronous execution is heavily influenced by data-dependencies between kernels. Especially if computed data is required by the CPU to dispatch additional kernels, a synchronization point is necessary where the CPU is forced to wait until the required results are available. This is often the case if the number of threads in a kernel is dependent on the result of a previous kernel call, as kernel execution is always initiated by the CPU with a known number of threads. For small workloads and multiple synchronization points, this quickly leads to underutilization of the GPU and unnecessarily long stalls of the CPU in turn. Besides minimizing CPU read-backs as far as possible, *indirect dispatching* can be employed to determine the number of dispatched threads directly from a GPU buffer written in a previous kernel execution.



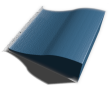
3 Related Work

Building upon the given layout-out fundamentals of sampling, related algorithms and techniques, we will now review previous works in this context. Since a vast body of work has been published in a variety of related fields, we will focus on publications that paved the way for the usage of SBRs in massive model rendering and large-scale volume visualization systems.

3.1 Usage of Sample-Based Representations in Computer Graphics

Since their first introduction in [Cat74], 2D textures have been the standard way to enhance polygonal geometry by sub-polygonal details and are nowadays supported by any hardware that offers 3D rendering capabilities. This technique, however, comes with two major limitations: Firstly, a 2D parametrization of the underlying 3D surface is required, which is generally non-trivial to obtain and often generated or at least corrected by artists in visual effects and computer games. Secondly, the resulting changes in surface appearance are restricted to the underlying geometry and cannot modify a low-resolution object's silhouettes. While textures were initially introduced to influence diffuse material colors, they were quickly extended to modulate the surface's specular and reflective properties [BN76] as well as to model small bumps [Bli78] which indirectly influence the surface's normals and therefore specular reflection directions. Since then, a number of extensions have been proposed to improve the perception of surface details on

Figure 3.1: Relief mapping of a stone texture applied to a low-polygon object. Source: [POC05].



otherwise flat polygons especially at grazing angles: Relief texture mapping [OBM00] and its applications to arbitrary surfaces [POC05] (Fig. 3.1) as well as parallax mapping [KTI*01] and the more advanced parallax occlusion mapping [BT05] enhance polygonal surfaces by texture-, height- and normal-maps and are able to simulate correct parallax effects and even self-shadowing. Policarpo et al. [PdON06] also demonstrated an approach for non height field structures. Shell maps [PBFJ05, JMW07] aim at adding surface details at silhouettes by extruding the geometry along the surface normal and mapping detail information e.g. from 3D textures into the resulting volumetric space.
For further reading and a comprehensive overview over the state of the art of GPU displacement mapping, we refer to [SKU08].

3.1.1 Unparameterized Mapping

A number of approaches have also been developed to avoid the need for an explicit parametrization of the surface, or even the need for an underlying surface at all. Early works of Peachey [Pea85] and Perlin et al. [PH89] demonstrate the mapping and generation of texture data in 3D space. With the resulting 3D textures, a surface parametrization is trivially given by the 3D identity function of a sample's position. To reduce the memory requirements of the sparsely populated volumetric texture, Lefebvre et al.[LH06] add one indirection to the texture lookup using hash functions evaluated in GPU shaders.

An alternative to spatial hashing was developed earlier in parallel by Benson et al. and DeBry et al. [BD02, DGPR02]: Octree textures use a regular space subdivision in form of an octree to minimize the storage overhead induced by empty space. The basic idea extends 2D texture mipmapping [Wil83] into 3D space, storing multiple lower resolution versions of the texture generated by downsampling in 3D. To deal with the resulting increase in memory, only a narrow shell around the surface is stored. During rendering, the correct level is selected and filtered to prevent aliasing and visible level transitions. A similar approach is followed by Lefebvre et al.'s textured sprites [LHN05b] and tile trees [LD07]: a single object is decomposed hierarchically until a local parametrization can be easily obtained in contrast to a global one.

A GPU implementation of octree textures has been presented by Lefebvre et al. [LHN05a]: They used a single 3D texture (*indirection pool*) to encode a pointer-based octree structure as well as the leaf color data. Each texel in the indirection pool contains a reference index to a child node in the same texture, which may be marked as empty if no children exist. The leaf nodes contain color information along with a one voxel wide halo to allow for hardware interpolation of the data. The concept and used data structure is illustrated in Fig. 3.2. Besides its usage for parameter-free texturing of arbitrary meshes, Lefebvre et al. demonstrate the application of their data structure to simulations such as liquid flow on a surface.
A similar, hierarchical scheme was used in the context of global illumination [CB04]: Brickmaps subdivide the scene in 3D space, but contain bricks of fixed voxel extent at the leaves. This solution provides a

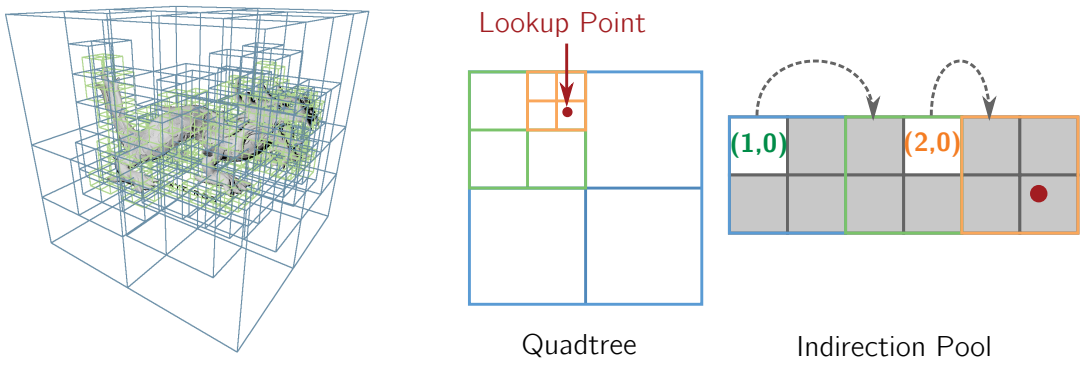


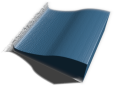
Figure 3.2: Octree textures [LHN05a]. Left: An adaptive octree encloses a mesh. Blue denote the inner nodes with children, green the leaf nodes that contain texture data. Right: Example of storing a sparse octree in an indirection pool, simplified to a 2D quadtree example. Each node contains the (x,y) index of its first child inside the indirection pool, if children exist. A lookup point in 2D can be located inside the texture by hierarchically traversing the quadtree.

good trade-off between memory efficiency and complexity. Also the memory overhead required for the halo data around each sample is greatly reduced: For a single voxel, the halo increases the memory usage by 26 times over the original data. Using 32^3 voxel bricks, this overhead shrinks to 50% of the source data. Naturally, the amount of empty space inside each brick is dependent on the complexity of the surface therein and greatly influences the net memory reduction or increase of a bricked structure in comparison to single voxel leafs.

3.1.2 Rendering from Sample-Based Representations

Closely related to our work is the rendering of objects or whole scenes from sample-based representations without underlying geometry. In early works, Schaufler et al. [SS96] use pre-rendered 2D representations (*impostors*) for distant objects instead of an actual geometric representation. To make the approach feasible for arbitrary views of non-rotationally symmetric objects, billboard clouds [DDSD03] represent complex geometry by multiple impostors. Neyret [Ney98] used 3D textures as a level-of-detail representation for complex meshes in CPU ray-tracing. He proposed accurate filtering methods for colors and normals to maintain the reflective properties of the input data when resampled into a texture at different resolutions. His representation could also be used inside a shell above a surface, allowing for the mapping of vegetation onto terrain. This work has inspired later GPU-solutions, such as [DN04], which presented a slice-based solution for the rendering of large, static forests, and was also generalized to arbitrary meshes as volumetric billboards [DN09] (Fig. 3.3). In the spirit of octree textures, Pfister et al. [PZvBG00] obtain a voxelization by a depth-peeling from orthogonal directions [LR98] and insert the non-empty voxels into a sparse octree, from which they are rendered as oriented point-splats termed *surfels*.

Aside from terrain rendering systems as e.g. [DKW09], height fields and similar structures have seen



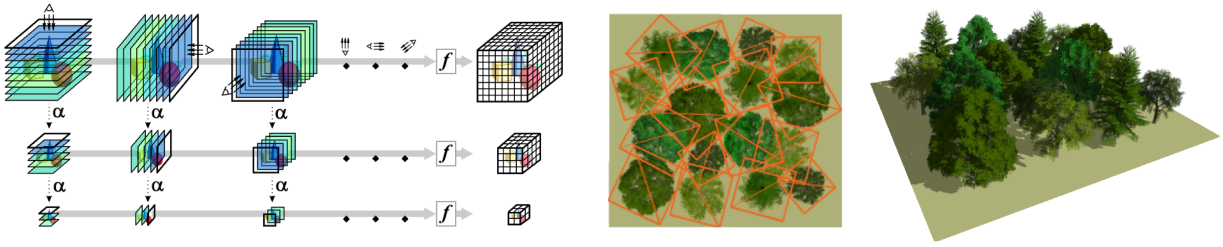


Figure 3.3: Volumetric billboards [DN09]. Arbitrary polygonal meshes are voxelized from six orthogonal directions, combined by α-blending along the view direction as well as a user-defined function f, and downsampled to multiple levels of resolution. Left: Construction process, right: results achieved with this method for a small forest. Source: [DN09].

many applications for reconstructing and rendering more complex models. [BD06] used relief textures from six orthogonal projections to reconstruct a 3D image. Building upon the idea of volume surface trees [BHGS06] and geometry textures [dTWL08], Novák et al. [ND12] split up objects using a bounding volume hierarchy until a leaf can be represented by a local height field under a predefined maximum error. The resulting representation is then rendered with a GPU ray-casting system. A similar idea was also proposed in [CHCH06], where a 2D geometry image representation of a 3D mesh is used as a level-of-detail solution.

With the availability of GPGPU compute capabilities along with NVIDIA's CUDA platform, much more flexible memory addressing became available along with direct write access into arbitrary buffers. Developers were no longer confined by fixed texture formats for the storage of arbitrary data, which quickly resulted in the realization of more complex data structures on the GPU. Sparse voxel octrees [LK10] were presented as an extension of GPU octree textures. The authors demonstrated the implementation of a sparse octree texture with single voxel leaves and out-of-core capabilities. In contrast to previous works, the octree is not used to texture existing geometry, but is directly rendered using GPU ray-casting. Due to the lack of halo data, interpolation in object space is too costly for interactive rendering and was thus performed as a post process in image space, exploiting the knowledge of the underlying levels of details and thus the required filtering radii. Extending on this work, [KSA13] presented a more compact data structure by identifying recurring regions inside the tree, but sacrifice the ability to store additional data like colors and normals along with the surface geometry.

3.2 Level of Detail Techniques in Direct Volume Rendering

Due to the amount of data naturally present in full, i.e. non-sparsely populated, 3D volumes from computer tomography scans and similar applications, data reduction by level of detail is an intensively researched topic in this area. Laur et al. [LH91] presented a hierarchical approach for interactive volume rendering as early as 1991. They generated a 3D mipmap of the volume and stored an octree based

on estimated error terms along with the data. During rendering, the cells are drawn with splats that match the extent of the cell at the according level. By progressively refining the image when moving, interactive frame rates of 5 fps could be achieved, with the image slowly converging to the final quality if the camera stopped moving.

A number of different LoD selection techniques have been proposed over the years. Levoy et al. [LW90] used volume ray-casting [Lev90] in combination with an eye-tracking device. Instead of requiring multiple resolutions of the volume, they rendered the image into multiple levels of a render target. Only areas currently in focus of the user's eyes are rendered in the highest resolution, and coarser levels are used elsewhere. This results in a sparsely populated 2D mipmap from which the final image can then be composed.

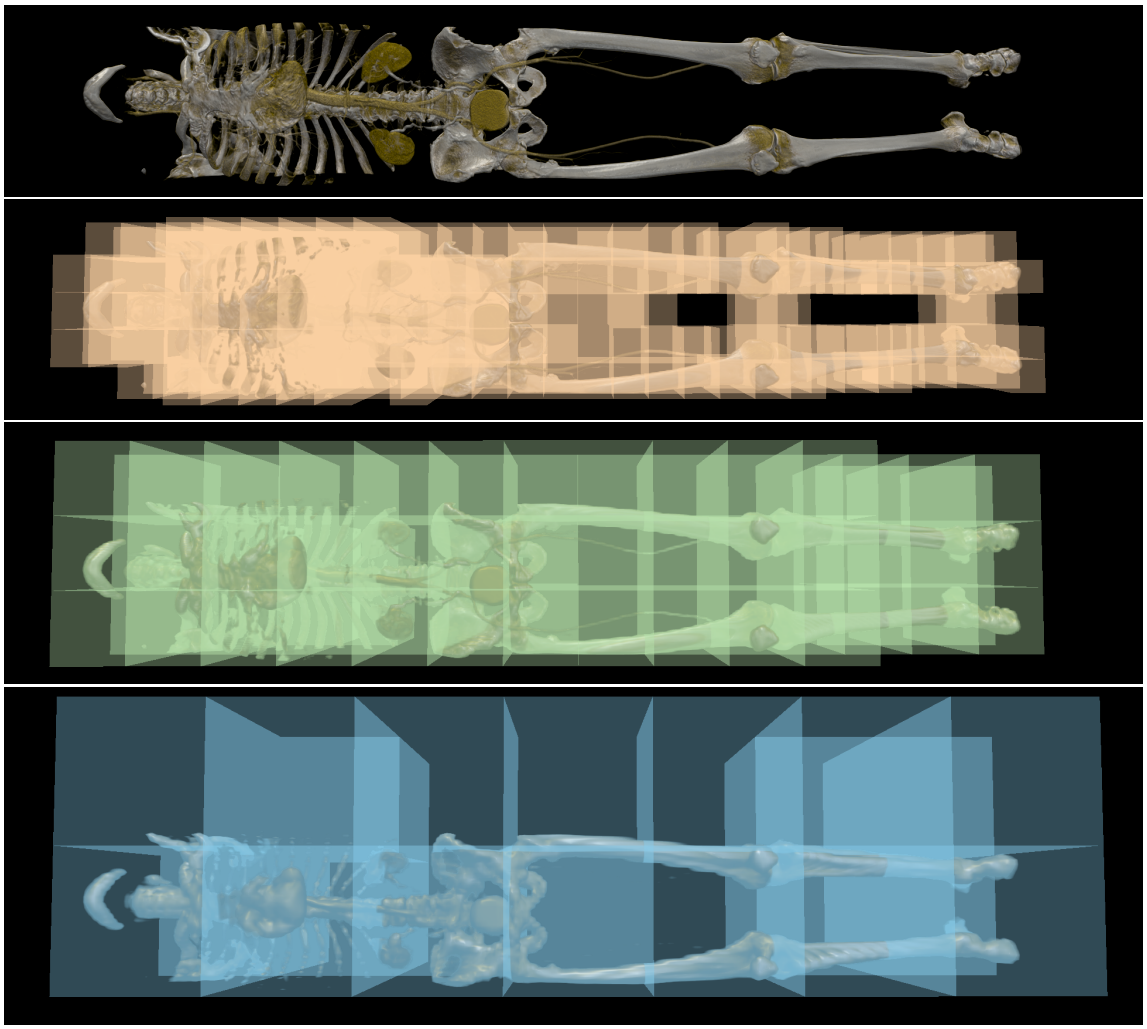
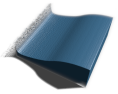


Figure 3.4: A volume is downsampled (coarser LoD from top to bottom) and disjointed into evenly-sized bricks on each level.

LaMar et al. [LHJ99] were one of the first to select an appropriate level of detail from an octree for each



node based on the position of the camera in the scene in hardware-assisted volume rendering. They subdivided the volume into bricks of fixed voxel extent and created downsampled versions to represent coarser LoDs using fewer bricks—$\frac{1}{8}$ of the respectively finer level—with the same amount of voxels per brick (Fig. 3.4). The resulting octree is traversed top-to-bottom on the CPU, with distant regions represented by coarser levels while regions close to the viewer require higher resolutions. The node data is then rendered using spherical shells as a proxy geometry whose extent depended on the selected level.

Due to the usage of cubic bricks of equal voxel resolution, the volume data can be stored in a single 3D texture. Figure 3.5 schematically depicts a three-level hierarchy of bricks and the corresponding storage of the volume data. Boada et al. [BNS01] also took data characteristics such as homogenity and a user-defined importance into account to locally restrict the generated octree levels. This was later extended further [LLYM04] to also consider the currently applied transfer function. In a similar approach, Li et al. [LMK03] clustered similar voxels inside the volume to more efficiently skip empty space by performing larger ray-casting steps in homogeneous regions.

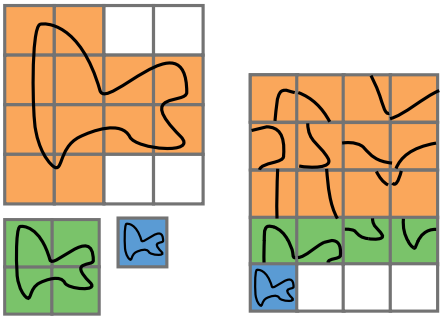


Figure 3.5: Storage of bricks from different LoDs in a single texture.

In contrast to the widely used hierarchical (octree) bricking schemes, Beyer et al. [BHMF08] introduced a flat bricking scheme (see Fig. 3.6) that represents downsampled levels with the same amount of bricks of decreased resolution—in analogy to an octree, the resolution is reduced to $\frac{1}{2}$ per dimension, resulting in a total of $\frac{1}{8}$ of the resolution of the finer level.

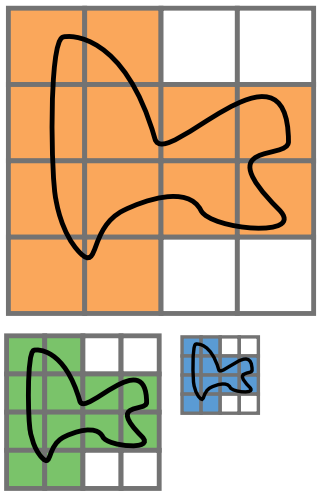


Figure 3.6: Flat bricking

This allows a more flexible LoD selection during rendering: Nearly homogeneous regions can be represented with coarser resolutions, while high-frequency regions receive a large number of voxels. While this approach can efficiently reduce the memory usage by an improved local adaptation of the LoD to the volume and viewing properties, it requires a more complex layout to store bricks of mixed resolutions in a single 3D texture, which introduces additional memory overhead due to empty space and complicates memory management, as the bricks can not always be tightly packed. Moreover, a specialized and thus manual interpolation of values between brick boundaries is required to hide the level differences of adjacent bricks.

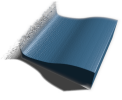
Prior to their work, a number of approaches have been proposed to deal with artifacts resulting from the mixture of different LoDs in a single image. Weiler et al. [WWH*00] demonstrated a hierarchy construction that guarantees consistent level transitions by interpolating values at correct levels in the boundary regions. For GPU ray-casting,

[LLY06] presented an interpolation scheme between multiple levels without the need for precomputed voxel overlaps, which in term cannot rely on hardware-supported trilinear interpolation and comes at a performance penalty.

3.3 Out-Of-Core Volume Ray-Casting

With more and more high-resolution datasets available from simulations or scans, the demand for rendering systems being able to deal with big data in an interactive, out-of-core fashion grew. The term *out-of-core* is generally used to describe the handling of data that is too large to fit into the system's main memory and is dynamically loaded and unloaded in small chunks from and into storage memory, such as a hard drive. GPU-based systems can also be considered out-of-core if the available GPU memory is insufficient to store the complete data set, as the same basic techniques are applied to deal with this problem. It is important to note, however, that streaming data from CPU- to GPU memory is at least one order of magnitude faster than reading from a storage memory and thus the impact of stalls resulting from "blocking" reads is less severe. Nevertheless, streaming all required data in every frame is impractical for highly interactive applications: Even at the theoretical bandwidth maximum of the latest revision of the PCIe 3.0 Bus at nearly 16 GB / s, filling a 6 GB GPU memory requires about 0.375 seconds.
In data visualization systems, the streaming process is usually view-driven and dependent on the current position and viewing direction inside the explored data. These systems are thus closely coupled to the solutions already presented in the previous section. As the threshold for data sizes being regarded to as "out-of-core"—as opposed to in-core—is platform-dependent and evolves with hardware development, we only consider more recent work in this section that is potentially capable of scaling with the available data sets and system memory. As we are interested in single-node solutions, we do not consider systems designed for multi-node, distributed memory clusters such as [CDM06, FCS*10, BHS*11]. The same applies to non-interactive applications such as the open source C++ library OpenVDB [Mus13], which gains more and more attention especially in visual effects production.

Optimized CPU ray-casting systems have been able to demonstrate impressive, interactive results, but can usually keep the rendered data sets in the large CPU main memory and are thus regarded to as in-core. Examples include [KWPH06], which used an adaptive octree structure to quickly skip empty and homogeneous regions, but was restricted to isosurface rendering, for which a majority of the rays can quickly be terminated as they hit the opaque surface. Direct volume rendering, on the other hand, comes at a severely higher cost in terms of ray marching steps and thus bandwidth requirements. Knoll et al. [KTW*11] could later achieve rates of 5 fps and more on high-end CPU hardware by employing a bounding volume hierarchy without LoD and carefully hand-optimizing the use of SIMD CPU features like SSE. It should be noted, however, that their demonstrated advantage in comparison to GPU solutions



is solely due to the limited GPU memory that requires a large portion of the data to be uploaded every frame.

Most closely related to our work are GPU ray-casting systems. Earlier works such as the *Tuvok* engine [FK10] relied on the GPU solely for rendering, while data management, brick selection and visibility determination were performed on the CPU. This makes the system flexible and usable on a wide range of devices—the free volume rendering application ImageVis3D, which is built upon Tuvok, is for example also available on mobile devices, where the ray-casting rendering algorithm is replaced by volume slicing. On the downside, drawing of each brick has to be issued by the CPU, which comes with a performance penalty for high numbers of draw calls and requires brick sorting for correct blending. In addition, this forces the renderer to either perform no occlusion culling at all—thus uploading all bricks in the correct LoD inside the view frustum—or very conservatively, as it would require continuously reading back the occlusion information from the GPU to drive the selection of bricks. Aside from requiring bandwidth, this enforces a synchronization point between the CPU and the GPU which drastically reduces performance and, thus, is impractical.

3.3.1 Ray-Guided Systems

To overcome these problems, Crassin et al. introduced the GigaVoxels system [CNLE09] and along with it the notion of *ray-guided* ray-casting. Concurrently, Gobbetti et al. presented their own, similar solution [GMG08]. In previous systems that employ the GPU for rendering, data management was CPU-driven and *preemptive*: The CPU determines the required data, uploads everything to the GPU, and starts the rendering process. The GigaVoxels system shifts this paradigm to a GPU-driven, *request-based* procedure. On top of applying well known out-of-core streaming and caching techniques, two main differences to existing systems are to note:

1. **Single-pass rendering:**
   All rendering is performed in a single pass on the GPU. One draw call is issued to rasterize the bounding box of the volume, and a pixel shader is responsible for traversing the volume along the ray from the camera to the bounding box exit. As in previous systems, the volume data is preprocessed and stored in a multi-resolution hierarchy in the form of an octree—or N^3 tree as a more generalized structure, which is the extension of an octree to higher branching factors. The data structures are inspired by the already presented octree textures implementation [LHN05a]: A *node pool* stores node descriptions and pointers to children, if existing, along with pointers into a *data pool*, which contains the volume data of the node. This hierarchy is traversed along the ray, and for each node, the GPU selects the required cut-off level of detail that is required to maintain a user-defined screen space error.

2. **GPU Feedback:**
   Initially, only the root node is present in GPU memory. During rendering, the GPU gathers

information about the data along each ray. Depending on the data and transfer function, the node pool contains information on whether a node contains visible data, is empty or can be represented by a homogeneous region, and the data pointer indicates if the data is currently present in GPU memory. If missing children or brick data is encountered, the GPU stores this information and either terminates the ray or falls back to rendering coarser LoDs; rays are also terminated if a maximum saturation is reached, yielding in optimal occlusion culling on a per-pixel basis.

At the end of each frame, the information about missing data is read back to the CPU, where it is processed and the required uploads are performed. In the next frame, this procedure is repeated.

Figure 3.7 illustrates this process for a single frame.

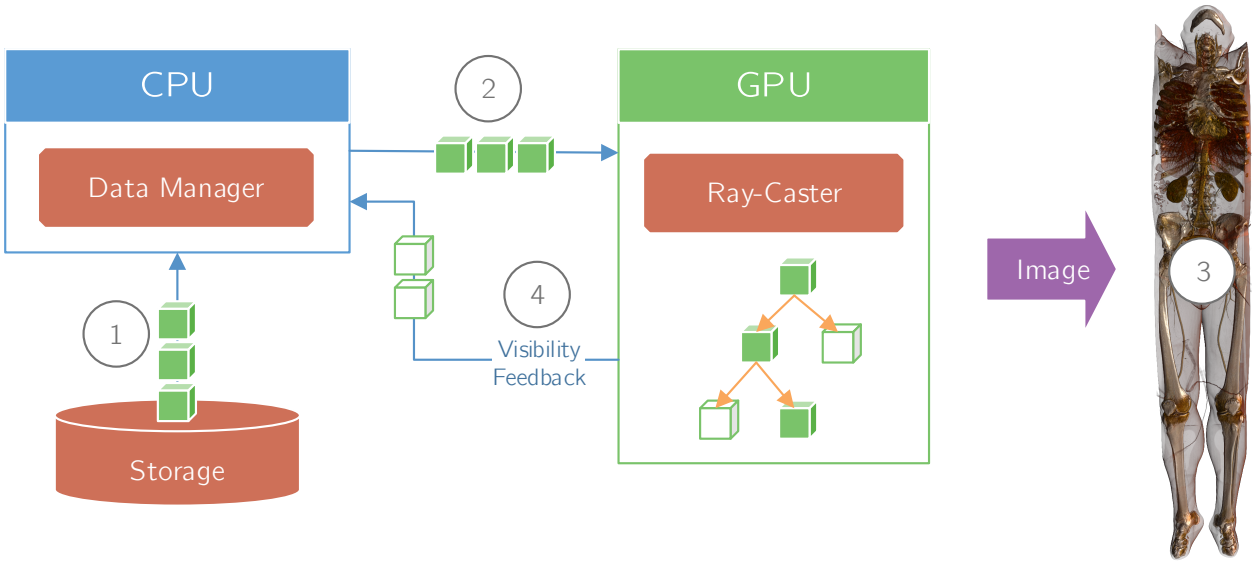
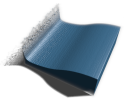


Figure 3.7: Ray-guided volume ray-casting: A multiresolution octree is created in a preprocess and stored. The CPU reads required bricks from disk (1) and uploads the volume data along with the octree node metadata to the GPU (2). The GPU traverses the sparse octree down to the required LoD or until missing nodes are encountered and ray-casts the non-empty leaf bricks (3). During traversal, it stores information about missing visible bricks (white). After rendering has completed, this information is transferred back to the CPU (4), where the missing bricks are loaded from storage and the process restarts.

3.3.2 Improvements and Similar Applications

A number of papers have since presented similar, improved systems based on these ideas. One major change was introduced only shortly after the original publications with the availability of general-purpose computing on the GPU, which allowed for scattered data writes from a pixel shader to arbitrary locations in global GPU memory. This made the complex implementation of GPU feedback, which packed the data into multiple render targets and assorted the information to the existing bricks, obsolete. Guitián



et al. [GGM10] updated their framework to read back a small linear buffer that contains visibility information of all bricks in the data set. Crassin [Cra11] also presented a sophisticated system that heavily utilizes CUDA to perform cache management and data requests directly on the GPU as an update to his GigaVoxels system. His Ph. D. thesis also provides valuable insight into a generalized paging and caching scheme which served as a basis for our framework.

Engel [Eng11] presented CERA-TVR, a similar system that was built for general volume rendering applications and can handle anisotropic domains as well as distributing the rendering over multiple frames instead of falling back to coarser LoDs until the requested data is available. In a later update, the ImageVis3D engine also was equipped with a ray-guided solution [FSK13] and was employed in an exhaustive analysis of the influence of different brick sizes, loading- and caching schemes.

A slightly different approach was taken by Hadwiger et al. [HBJP12] for the visualization of microscopy data obtained by multiple 2D image slices. With the premise of dealing with extremely dense data that cannot efficiently benefit from empty space skipping, they use multi-level page tables instead of an octree to select the required bricks in the correct LoD during fixed-step ray-marching. This allows to quickly map from virtual (i.e. a brick level and position inside this level) to physical addresses and eliminate the overhead of octree traversal, since all nodes nearly exclusively contain non-empty data. In addition, data can be acquired and added into their acceleration structure on the fly during visualization.

3.4 Sample-Based Representations in Massive Model Rendering

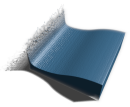
While the GigaVoxels system is capable of rendering massive data sets, its original design intends the usage for medium-sized scenes in applications such as video games, where fast rendering and high quality are of primary concern. It is heavily optimized for bricks of sizes as small as 4^3 to represent sparsely populated, hand-crafted scenes and for the combination with high-quality rendering effects such as global illumination through voxel cone tracing [CNS*11]. For densely sampled environments, the choice of small voxel bricks introduces a severe overhead due to the overlaps between neighboring bricks. Larger bricks, on the other hand, result in an increased amount of "wasted" memory inside each brick due to empty cells, making the optimal brick size for the system very application- and even scene-dependent.

Especially in the context of rendering massive models, more compact SBRs have been used as an alternative or extension to triangle data. Rusinkiewicz and Levoy presented the first true out-of-core point-based renderer, QSplat [RL00] in 2000. Their approach can render highly detailed models at interactive rates, but is restricted to point-sampled data and relies on heavy preprocessing to reduce the amount of rendered points. Guthe et al. [GBBK04] also presented a point-based renderer in combination with triangle-based rasterization to achieve a fast, appearance-preserving LoD rendering. Triangle-based input data is converted into a combination of triangles and points over various LoDs and stored in an octree structure. They demonstrated interactive framerates for high-quality renderings including soft

shadows. Their algorithm is based on point splatting and rasterization of individual octree cells and is thus restricted in scalability by the number of generated cells and points due to the already introduces limits of the GPU rasterization pipeline (section 1.3). They do not provide preprocessing times, but also greatly reduce the amount of primitives to be displayed.

Wald et al. [WDS05] approximated clusters of primitives by volumetric proxies, but did so only during loading times and always rendered the complete scene at the finest level for the final image. A similar approach is taken by the Far Voxels system of Gobbetti and Marton [GM05], where clusters of primitives are approximated by view-dependent cubic voxels which are then intermixed with rasterized triangle-based geometry. They supported only simple shading and were limited in quality by the used voxel splatting approach, but could demonstrate stable frame rates for a large variety of meshes composed of hundreds of millions and even billions of triangles. In the same spirit, [CW14] presented a rasterization-based solution that employs hybrid triangle / voxel rendering with a very low memory footprint, which can fully benefit from anti-aliasing via multisampling and can be easily integrated into existing rendering pipelines.

In the context of ray-tracing, related solutions have been presented to handle massive amounts of data: Yoon et al. [YLM06] replaced coarser nodes in a kd-tree with simple planes which are then rendered instead of the actual underlying geometry. While they were able to significantly speed up the rendering of large scenes, they relied on blocking I/O during rendering and can not reach truly interactive performance. For production-quality offline image generation, PantaRay [PFHA10] also exploits a voxel-based level of detail for coarser BVH nodes in the distance. Developed in parallel with our system, Áfra [Áfr12] demonstrated a hybrid voxel / triangle solution for interactive ray-tracing of large models. His system is purely CPU-based and achieves interactivity by allowing for a visible error of more than one pixel, but in turn supports fast ray-traced effects such as indirect illumination.



4 System Design

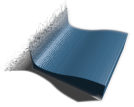
4.1 Design Goals

Section 1.5 stated three design goals of our system: scalability, interactivity, and flexibility. In review of the problems presented in section 1.2 and 1.3, and the observations in section 2.3.4, we arrive at the following conclusions:

1. Voxels, and sample-based representations as a generalization, can be very efficiently combined with ray-casting to achieve these goals.
2. While the choice of acceleration structure may impact performance, the differences are not crucial especially if larger bricks and complex data structures are used in the leaf nodes.

From this, we derive our framework design, which is largely based on GigaVoxels [Cra11] with a ray-guided GPU ray-caster at its core. The scene is organized in a sparse octree with bricks of a fixed spatial extent—usually one to two orders of magnitude larger than the extent of a single sample—inside the bricks. The increase in memory resulting from a bricked structure is counteracted by compact, data-specific data representations in the leaves instead of a full voxel grid. We chose this design as it complements the stated goals in the following way:

- **Scalability:** The octree offers a fast build process that can be performed out-of-core, in parallel and, if necessary, at runtime in a top-down fashion. It allows for a straightforward integration of LoD into the creation and rendering. In combination with ray-casting, visibility determination and out-of-core streaming can be efficiently and elegantly combined without negative performance implications.
- **Interactivity:** Sample-based ray-casting has been shown to be a perfect fit for the massive parallel processing power of current GPUs. The usage of bricks of larger spatial extent in the leaves results



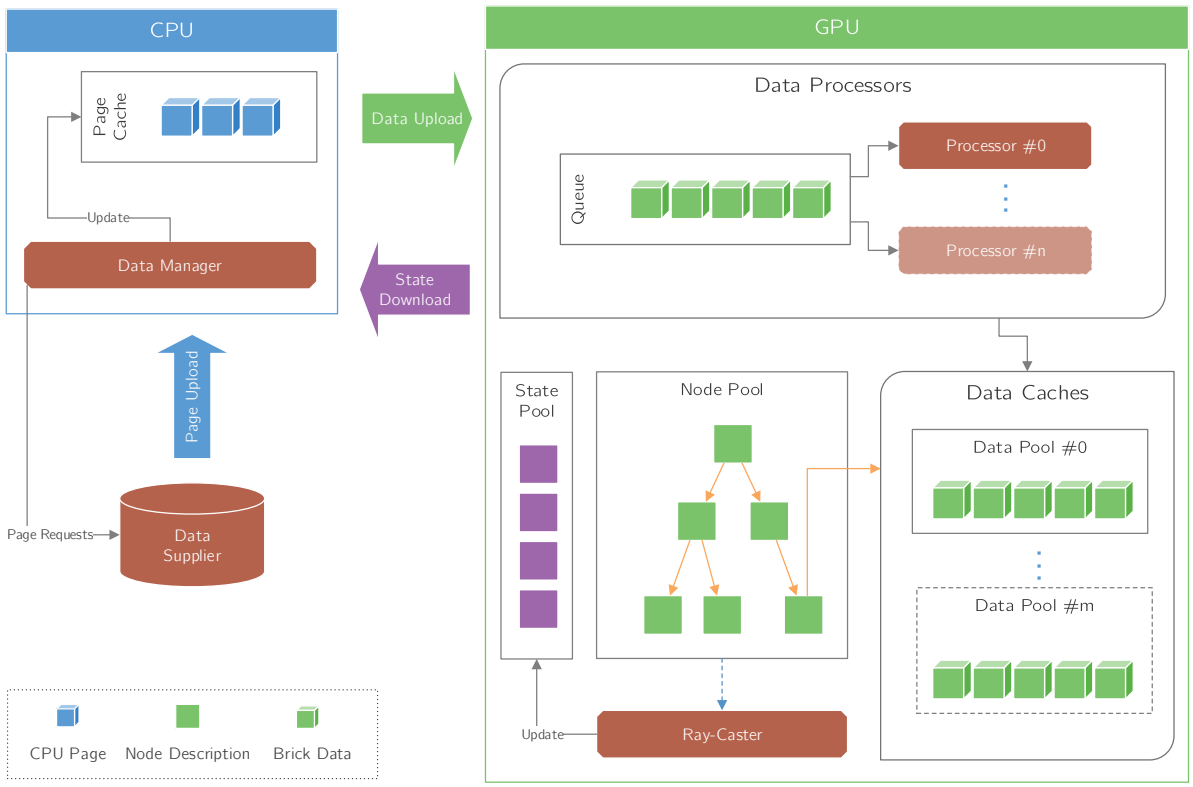


Figure 4.1: Basic design of our rendering system with the four core components shown in red.

in a rather shallow tree that can be efficiently traversed and managed. Furthermore, an integration of arbitrary data structures inside the leaves is trivial.

- **Flexibility:** The adaptive acceleration structure performs well for most kinds of data and, thus, our system can handle sparsely sampled surfaces as well as dense volumetric data sets. We also support the inclusion of boundary regions to rely on hardware-accelerated trilinear interpolation.

Figure 4.1 depicts a high-level system overview. In the following, we introduce the basic components as well as some general-purpose implementations of the modules. Application-specific data structures and additions to the system are then described in the according application chapters later on. On a high level, our system contains four core components:

Data Supplier:
The Back-end of the system is a central *data supplier*. All data is delivered in atomic entities of pages whose size and content is dependent on the type of the supplier. Suppliers can be very application-specific and customizable; in general, they are able to load or generate pages based on the application's needs such as bricks assigned to a specific LoD, position in the scene, or time step in time-dependent scenarios.

Data Manager:
The *data manager* coordinates all requests issued by the GPU and determines which data needs to be paged in and out of CPU and GPU caches. It requests newly required pages from the supplier and handles brick upload into the GPU data pools. In case of memory exhaustion, it also determines the data to be evicted from CPU caches and all GPU pools.

Brick Processors:
After brick data has been uploaded to the GPU, it may be processed by one or multiple *brick processors* on the GPU in parallel or sequentially. Brick processors are once again application specific and perform tasks such as decompression, acceleration structure creation, or conversion into a renderable representation. Usually, a fixed number of bricks are added to a processing queue and processed in a single batch to utilize GPU parallelism.

Ray-Caster:
The ray-caster traverses the octree stored in the node pool and reports missing children and data to the CPU. For now, rendering of the bricks stored inside the octree nodes is regarded to as a "black box". It will be customized to work with varying SBRs dependent on the application and, thus, will be detailed in later chapters.

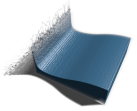
4.2 GPU Data Pools

Renderable data on the GPU is organized in a number of pools. The *node pool* contains the octree structure along with possible brick metadata information. The minimum information carried by each node is the following:

```
struct Node
{
    Node*       childPointer;
    void*       dataPointers[N];
    Metadata    meta;
}
```

Here, `childPointer` is a pointer to the first of the node's 8 children. It is thus required that either none or all of a node's children are present in the pool at any time. However, it is important to note that the presence of any renderable data for a node is independent of its presence in the node pool. The node pool is organized as a 1D buffer in GPU memory, with `childPointers` being simple offsets inside the buffer. Renderable data is referenced by one or multiple `dataPointers` into predefined *data pools*. Depending on the stored data, these pools are of one of the following types:

- **3D Texture Data:**
  A large 3D texture that contains a number of 3D voxel bricks of predefined size. The pointer itself is then a 3D index referencing the foremost top left corner.



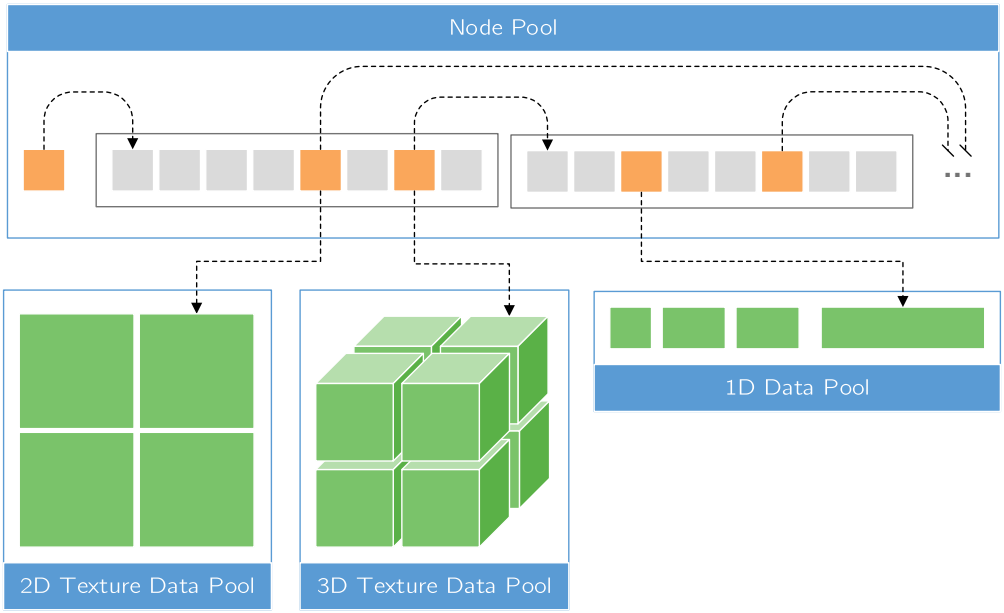


Figure 4.2: Data Pools. Each node in the node pool points to the first of its 8 children and to one of three exemplary data pools that store 1D, 2D and 3D data.

- **2D Texture Data:**
  A large 2D texture containing a number of 2D textures of predefined size. The pointer itself is a 2D index referencing the top left corner.
- **1D Data:**
  A generic, 1D buffer. This pool may contain data chunks of varying size. The pointer itself is a 1D offset from the start of the buffer.

Figure 4.2 schematically shows the layout of the node pool and the various data pools. All pointers may reference data bricks that are not present in GPU memory or do not exist at all; both cases are handled by setting the according pointers to `NULL`. Multi-dimensional pointers are stored as a single integer using Morton codes [Mor66]. Finally, the `meta` component of a node contains application-defined metadata information, e.g. a flag indicating the presence of children and SBR data for the brick. This allows the GPU to request missing nodes or SBRs from the CPU. Other examples include the presence of acceleration structures inside the node, or minimum / maximum density values contained in a volumetric brick.

4.2.1 Data Pool Memory Management

All memory pools are preallocated with a fixed size by the CPU. 2D and 3D texture pools are partitioned into slots of a fixed size, thus memory allocation and deallocation inside each pool can be easily managed on a per-slot basis. 1D data pools, on the other hand, often require allocation of varying sizes. A number

of general-purpose parallel allocators have been proposed to manage memory directly in a GPU kernel [VH14, WWWG13, SKKS12]. In our system, memory allocation is mainly performed by the CPU using one of the following allocators depending on the application's needs.

Buddy Allocator:
A buddy allocator [Kno65] manages memory by treating the available space as a multi-level hierarchy: On the first level, a single block of memory is available. Upon allocation, a free slot may be split in half to provide two slots of half the size on a lower level until a predefined finest level. A list of free slots is maintained for each level. If memory of size s_m should be allocated, the finest available slot of size $\geq s_m$ is determined. If the finest level is not yet reached, the slot is recursively split until the child slots cannot provide enough space. Upon deallocation, the according slot is once again added to the list of free slots; if all slots of a parent are deallocated, they are merged into a free slot at the next coarser level. Figure 4.3 illustrates the process.
This allocator comes with no additional computational overhead once data is allocated, and allocations as well as deallocations are cheap. However, it wastes up to 50% of the available memory as all requested sizes need to be "rounded up" to the predefined chunk sizes. We deem a buddy allocator to be most useful if optimal memory utilization is not of primary concern, but a large number of allocations and deallocations of varying size is performed during runtime.

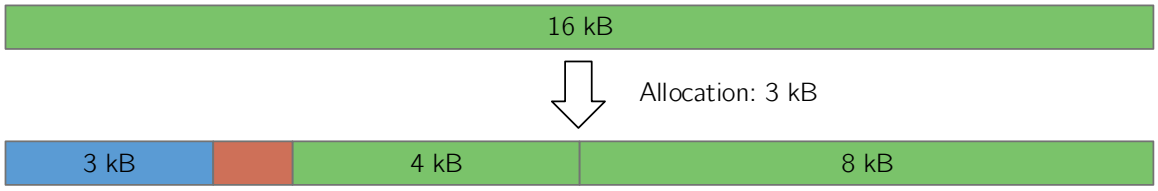
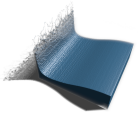


Figure 4.3: Buddy allocation. An allocation is performed by subsequently splitting a free block into two smaller blocks until the finest possible granularity that can serve the request is reached. Green and blue indicate free and allocated blocks, orange the wasted memory after the allocation.

Virtual Memory Allocator:
The virtual memory allocator manages data on the granularity of pages of application-defined fixed size. Each pointer into a buffer managed by a virtual memory allocator no longer represents a physical offset, but a virtual address consisting of a page-index and an offset inside this page. The 1D data pool provides space for a fixed number of pages. We keep a one-level *page table* in GPU memory to map from virtual page index to the currently occupied slot in the data pool, with a fixed value reserved to indicate that this page is not residing in GPU memory. For example, our fluid rendering application relies on 40 bit addresses, consisting of a 20 bit page index and a 20 bit offset to address half-words inside the data buffer. It thus requires a table of 4 MiB and is able to manage up to 2 TiB of virtual memory.



Whenever the data is accessed on the GPU, address translation is performed on the fly by splitting the virtual address and replacing the page index by the current offset read from the page table. The table itself can be re-uploaded whenever entries have changed, which does not impact performance due to its compactness. Figure 4.4 shows a simple example of virtual addressing on a per-voxel basis.

In comparison to a buddy allocator, the data is tightly packed inside GPU memory without empty space. While an additional lookup is required each time the data is accessed, we found this to have no noticeable impact on performance as the page table is very compact and many requests can be served from GPU caches. On the downside, the usage of virtual addresses increases the memory overhead for the pointers itself. We minimize this impact by storing a virtual base address for larger chunks—usually per brick—which allows us to resort to compact 32 bit or 16 bit offsets inside the chunk. This once again introduces a small amount of empty space if a page is not completely filled, yet does not provide sufficient space for the next chunk. This effect can be minimized by carefully choosing the page and chunk size. In addition, the page replacement strategy is dependent on a good distribution of data to pages which maximizes the spatial coherency inside a page.

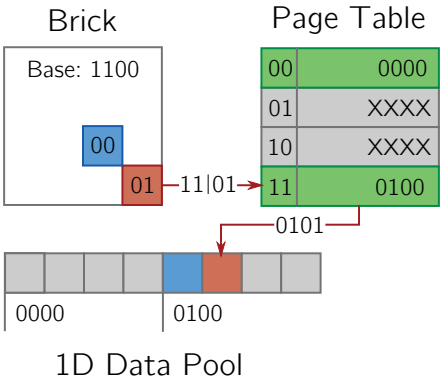


Figure 4.4: Address translation with 2 bit page addresses and 2 bit offsets. The depicted brick contains two voxels. Its base offset is combined with a per-voxel offset to generate a virtual address, which is translated by a 4-slot page table.

Heap Allocator:

The heap allocator assigns memory from a large area of predefined size and returns an arbitrary address inside this heap upon allocation. Internally, the memory manager keeps track of the allocated chunks as well as the free space available after each chunk (follow-up memory). Allocated chunks are organized in a binary search tree which is kept sorted by increasing follow-up memory at each time. Upon allocation, the first chunk that provides enough follow-up memory is selected, and a new chunk of appropriate size is created after the selected one. In turn, if deallocation is performed, the newly available memory is added to the follow-up memory of the previous chunk.

Since multiple deallocations cause the memory to be fragmented gradually, defragmentation is performed in regular intervals: In each frame, a fixed maximum budget of memory may be moved to close gaps between adjacent chunks. This requires an update of all pointers into the affected data pools as well as multiple memory copy operations inside GPU memory. Newer GPUs support copying data from overlapping regions without the need for an intermediate buffer to improve performance for these cases. Nevertheless, these operations may become a bottleneck if the budget is chosen too generously or defragmentations are forced by the memory manager due to insufficient available memory. The heap allocator is preferred if a moderate number of memory allocations and deallocations of greatly varying size is expected or the use of virtual addresses is not feasible due to the overhead from increased pointer sizes.

4.3 CPU / GPU Interaction

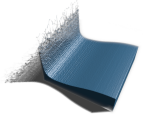
For image generation, the GPU ray-caster traces a ray for each pixel from the current camera position until the end of the scene's bounding box. The octree stored in the Node Pool is traversed until a node containing renderable data at an appropriate level is reached. Under the presence of LoD, the level is determined by a user-defined maximum screen space error as introduced in section 2.2.1; if no LoD is present, the tree is traversed down to its leaf nodes. In either case, various types of cache misses may occur during the traversal:

1. A node's children need to be traversed, but the `childPointer` is `NULL`.
2. A node's data needs to be rendered, but the according data pointer is `NULL`.

These two misses generate two different kinds of requests: A *subdivision request* and a *data request*. Requests are stored on the GPU using a separate *state pool*. As suggested by [FSK13], the state pool is organized as a simple hash table of fixed size independent of the node pool size. It contains an identifier of the node currently occupying the slot along with a single integer value (the request flag vector). Whenever any kind of request needs to be issued, the state pool slot corresponding to the node is determined by `Hash(node) % STATE_POOL_SIZE`. The `Hash` function simply generates a linear index from the 4D brick coordinates (x, y, z, level). If the according slot is empty or occupied by the same node, the request flag is added to the slot. If another node is already present in the slot the system performs a few iterations of linear probing in the next slots and, if no suitable slot can be found, discards the request. It is important to note that hash collisions do not indicate failure: the ray-caster will try to issue the request in subsequent frames as long as the cache miss has not been resolved, with the miss eventually being reported. In the meantime, the system may choose to render coarser LoDs, present the "incomplete" image or force multiple rendering passes until the rendering can be finalized.

After each render pass, the request flag vector is downloaded to the CPU which is then responsible for handling the requests. We keep a duplicate of the node pool in CPU memory as well as a hash table for fast mapping of node pointers to state pool slots. Once all requests have been assigned to the according nodes, the missing data is either directly uploaded to the GPU from the page cache, or it is requested from the data supplier. If the supplier cannot instantly deliver the data—e.g. if it requires loading the page from disk—the completion of the request is delayed until the page is available.

The CPU page cache as well as the GPU caches are administrated by the CPU and updated using a *least recently requested* replacement policy based on the requests contained in the last state pool. Crassin [Cra11] presents an advanced system to manage caches directly on the GPU without the need for mirrored data structures. As large portions of our system need to be adapted to specific applications, we pursued this simpler, yet more flexible approach. However, our applications rely on larger data bricks than the 4^3 voxels suggested by Crassin due to the compactness of our data structures—ranging from 16^3 up to 128^3. This significantly reduces the management overhead for caching and data updates as



well as the memory required for a reasonably sized state pool and the overall node pool and also allows for more brick metadata to be included. Usually a state pool size of very few megabytes is sufficient to handle requests with minimal collisions, which can be transferred and processed in a few milliseconds.

4.4 The Streaming Supplier

The simplest and most common example of a data supplier is a *streaming supplier* that loads pages of pre-generated data from storage memory. All bricks of the octree are grouped to pages of fixed size in a breadth-first manner (see Fig. 4.5) and stored on disk along with the sparse octree structure. When a brick's data is requested, the corresponding page is located and—if not already present—loaded into the page cache through the use of the asynchronous I/O features present in all operating systems. The GPU will then keep requesting the data over the next frames until the I/O operation has finished and the brick's data can be uploaded to GPU memory.

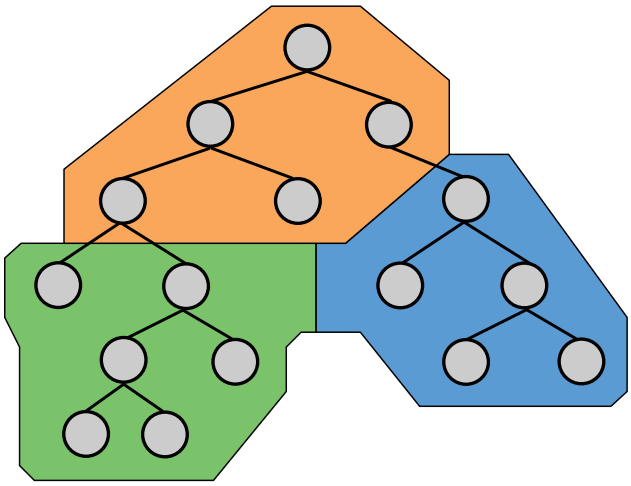


Figure 4.5: GPU bricks (grey circles) are grouped to pages (colored orange, blue, and green) in a breadth-first manner for storage.

The size of a single page has to be chosen carefully: While modern hard drives can deliver up to 200 MiB/s, they require a significant amount of seek time (typically in the order of 5 to 10 ms) before the data can be accessed. For small, scattered pages, this seek time quickly dominates the reading process. Larger pages, on the other hand, result in an increased number of bricks being loaded and stored in the CPU cache without ever being requested by the GPU and, thus, waste bandwidth and cache slots. This effect is minimized by the breadth-first organization of bricks into pages, as requested bricks also bring their neighbors along with a few finer LoDs into CPU memory to maximize the spatial coherency. Fig. 4.6 shows an exemplary analysis of disk throughput vs page sizes for a standard hard drive (Seagate Barracuda 7200.14, available since 2011). Based on these observations, we chose a page size of 4 MiB as a good trade-off for our default streaming supplier.

4.5 Octree Traversal

The octree is traversed using a 3D variant of DDA ray-marching similar to [RUL00]. We trace each ray from its first intersection with the domain's bounding box until it leaves the scene. Starting from

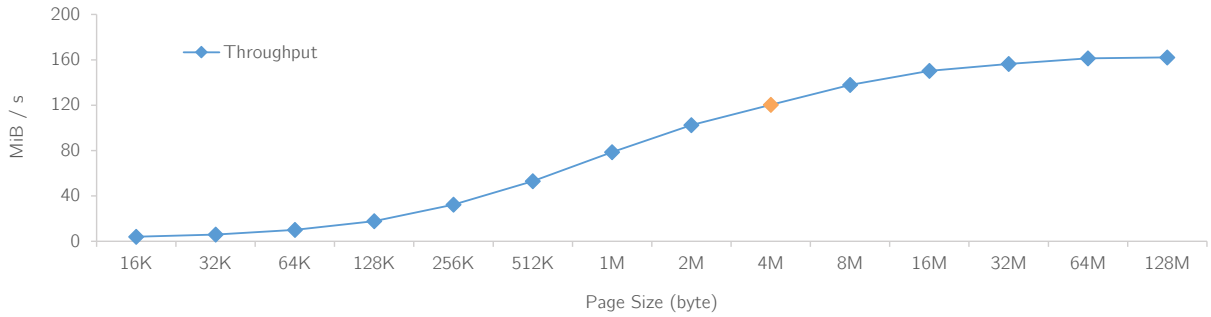


Figure 4.6: Disk bandwidth vs. page size. We chose a page size of 4 MiB as a middle ground between throughput and waste of cache memory.

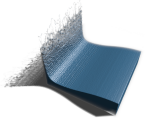
the root node, we descend down the octree until we reach a leaf or an internal node that satisfies our LoD criterion and contains renderable data. Rendering of the brick data in each node is then performed by a custom *brick traverser*, which is specific to the used SBR and will be described in detail in later chapters.
Traversal calculations are carried out on a ray given in its parametric form starting from an origin $\vec{o}$ in direction $\vec{v}$ as

$$\vec{o} + t \cdot \vec{v} \tag{4.1}$$

with a ray parameter $t \in \mathbb{R}$. To simplify the implementation of the brick traverser, the ray's position is converted into *node-local* space, where the current node extends in $[0, 1]$ in all dimensions. During traversal, three operations are carried out:

Descent:
If a ray descends down the tree, we calculate the according 3D child index from the brick entry position of the ray $\in [0, 1]$ as `childIndex = floor(rayPosition * 2.0) % 2`. The node pointer is then determined by adding the linearized child index to the current node's child pointer. A ray always descends into the child that is hit first and, from there, advances into the remaining children if multiple intersections occur.

Advance:
If a node contains no renderable data or the brick traverser did not terminate the ray, we advance to the next node. Given a ray with positive direction $\vec{v}$ that enters a node at position $\vec{p}_{in}$ in node-local space, we calculate a new ray parameter t_{min} such that the ray exits at position $\vec{p}_{in} + t_{min} \cdot \vec{v}$ as



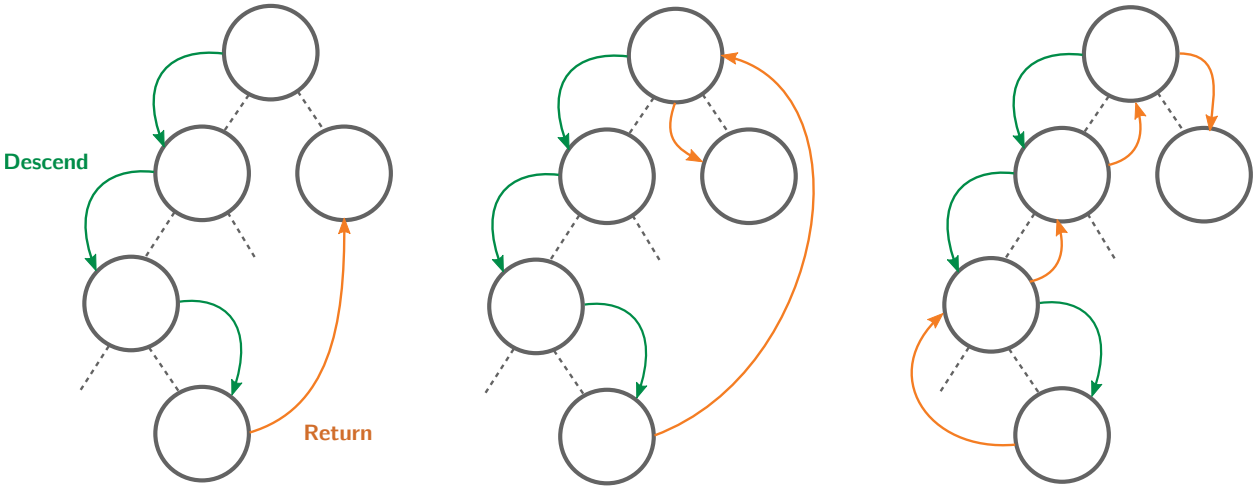


Figure 4.7: Examples of kd-tree traversal algorithms: Stack-based traversal commonly used in CPU ray-casting, and two stackless approaches.

$$\vec{t} = \frac{(1 - \vec{p}_{in})}{\vec{v}} \tag{4.2}$$

$$t_{min} = min(t.x, t.y, t.z) \tag{4.3}$$

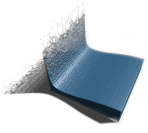
where the fraction is evaluated for each component. To avoid visiting the same node twice due to a lack of arithmetic precision, we also explicitly store the current node's index and add 1 to the dimension of lowest t value whenever the ray is advanced. To also handle negative directions, we mirror all negative coordinates during traversal and perform all calculations with a positive ray direction.

Ascent:

We avoid storage of neighbor pointers due to the increased management and memory overhead. Thus, pointers to the visited nodes are only available as long as they share a common parent. In case of a ray entering a node of a different parent, the next node to traverse needs to be determined. Several algorithms have been developed for kd-trees which also apply to octrees (for examples, see Fig. 4.7). In the context of CPU ray-tracing, a *stack-based algorithm* is most commonly employed: During descent into the first child, all remaining children which are also intersected by the ray are pushed onto a stack. When traversal of a node is completed, the next node to be tested is then retrieved using a stack pop operation. This algorithm requires only a minimal amount of node intersection tests and no additional node data needs to be stored, but it is not a good fit for GPU ray-casting as the potentially deep stack needs to be stored in comparably slow global memory. Thus a number of *stackless* approaches have been presented. *Kd-restart* [FS05] restarts the traversal from the root node once a child of a different parent is encountered during an advance operation. While this algorithm requires no additional data of

any kind, it may significantly increase the number of tests performed as already visited nodes at the top of the tree need to be retested. *Kd-backtrack* [FS05] reduced the number of tests by ascending up into the tree until the next node to be visited shares a common parent with the current one, eliminating the additional tests. On the downside, this requires explicit storage of parent pointers for each node and thus increases the size of the octree structure.

Our implementation relies on a combination of stack-based traversal and kd-restart known as a *short stack* [HSHH07]. Only a small, fixed number of return nodes is stored in a fast stack in registers. If the stack underflows during traversal, a restart is performed. In our experiments, a short stack of 4 nodes performed well on average and complements the comparably shallow trees we employ.



5

Direct Volume Rendering of Big SPH Simulations

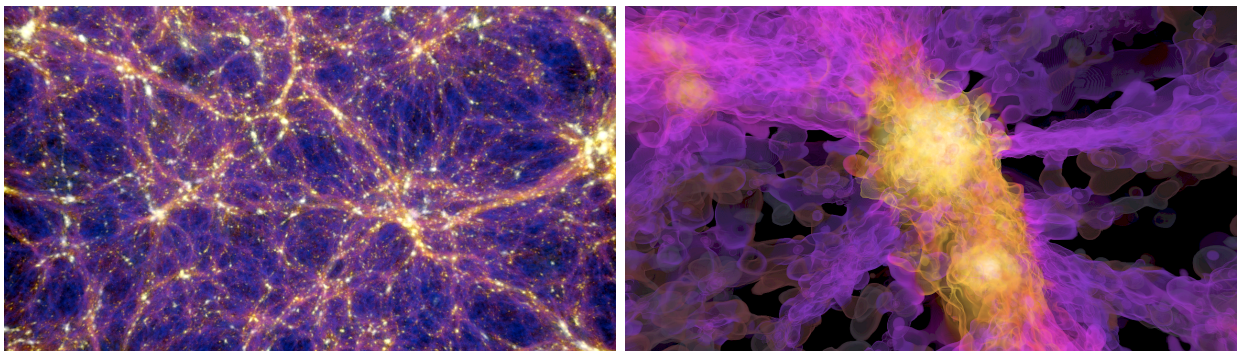
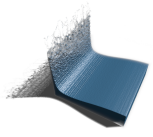


Figure 5.1: Cosmological structures in the Millennium Run at different resolutions. Rendering to a 1200×800 viewport took 213 ms and 79 ms per frame, respectively, using multiresolution volume ray-casting. The working sets in GPU memory were 2120 MB and 109 MB, respectively.

5.1 Introduction

In particle-based simulation techniques such as smoothed particle hydrodynamics (SPH), a volume covering a continuous field is reconstructed from a discrete set of particles carrying physical quantities. Since the particles are scattered irregularly over the 3D spatial domain, the reconstruction process requires determining at every spatial domain point the set of particles having influence on this point, and weighting their contributions to form the resulting value. Rendering SPH data can be done via volume ray-casting, by determining at every sampling point along the rays the influencing particle set and accumulating the respective values. This, however, requires huge numbers of search queries per ray to determine these sets locally.

A different approach is to resample the particle data into a grid in a pre-process and use cell-wise interpolation for reconstructing the values along the rays during rendering. This approach results in a



view-independent volume representation and does not require any overlap queries at run-time. If the data is resampled to a uniform grid, 3D texture-based volume rendering on the GPU can be used.

This resampling grid introduces a significant memory overhead, especially when the particle density dictates a high resolution to capture all simulation details. For instance, in the gas dynamics simulation addressed in this chapter—the Millennium Run—more than 10 billion particles were used to trace the evolution of the matter distribution in a cubic region of the universe. The spatial resolution of this simulation corresponds to an effective uniform grid size of about $100{,}000^3$.

5.1.1 Application Goals

In this chapter, we analyze the use of compression techniques and adaptive subdivision for the visualization of this particular data set. While the general SPH resampling technique is well-known and can in theory be applied to data sets of arbitrary sizes, it has since been unclear how to overcome the tremendous bandwidth- and storage limitations resulting from a uniform resampling if such extreme resolutions are enforced by the source data. The resulting data set then demonstrates the capabilities of our framework to deal with dense and spatially adaptive data in the context of direct volume rendering. To bring down the memory requirements, we embed a per-brick compression layer into the framework as a brick processor. By the use of a wavelet-based compression scheme, fast GPU-based decoding allows the exclusive caching of compressed bricks in CPU main memory to reduce the amount of necessary disc access. We build upon the `cudaCompress` library published separately by Treib [Tre14], which has already been used in several applications such as terrain editing [TRAW12] and vector field volume rendering [TBR*12]. `CudaCompress` employs Huffman-coding of a wavelet coefficient stream to achieve lossy compression of arbitrary data.

5.2 The Millennium Run: SPH and Related Work

The centerpiece of our evaluations is a cosmological N-body/SPH simulation known as the Millennium Run, presented in detail in the work of Springel et al. [Spr05, SWJ*05]. The Millennium Run stands as a representative for the class of gas dynamics simulations where gas expansion and contraction is simulated via SPH. Application areas of such simulations range from chemically reacting flows to molecular dynamics and cosmological structure formation. The Millennium data set consists of 10^{10} particles with varying radii of influence, reaching a spatial dynamic range of 10^5 per dimension in a 3D simulation domain.

5.2.1 Simulation

SPH has been introduced by Gingold and Monaghan [GM77, Mon92] and Lucy [Luc77]. Since then, a vast body of literature concerning simulation and visualization has been published, and a comprehensive review would go beyond the scope ot this thesis. We will thus only briefly touch on the underlying simulation of general fluids to provide the necessary basics for the resulting rendering techniques. Springel et al. [Spr05] provide a detailed insight into the fundamentals of their gas dynamics simulation, while Ihmsen et al. [IOS*14] discuss the basic principles of SPH for incompressible fluids with a focus on their use in computer graphics, and they provide a solid overview on a wide range of topics concerning SPH simulation.

The basic principle of SPH is the Lagrangian discretization of a continuous fluid with a set of small, moving elements. Each of those particles i consists of a position r_i at each discrete time t which is the center of its mass m_i, and of a density ρ_i. This formulation is used to numerically solve a differential equation describing the fluids' dispersion in space over time. Given a pressure gradient ∇P_i at a position r_i, [GM77] formulate the equation of motion of a particle at this position as

$$\frac{d^2 r_i}{dt^2} = -\frac{1}{\rho_i}\nabla P_i + F_i \tag{5.1}$$

with a total body force F_i acting on the particle. Over time, the positions of the particles are advected in the fluid using the local velocity

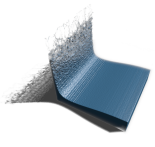
$$\frac{dr_i}{dt} = v_i. \tag{5.2}$$

The core component of SPH is a *kernel function* $W(r)$ which describes the computation of a physical quantity A at an arbitrary position r in 3D space by interpolation of the quantities A_i from a set of N neighboring particles at positions r_i:

$$A(r) = \sum_{i=1}^{N} \frac{m_i}{\rho_i} A_i W\left(\frac{||r - r_i||}{h_i}\right). \tag{5.3}$$

The radius of support h is called the *smoothing length*. Various kernel functions exist, for example the cubic spline kernel [Mon92]:

$$W(r,h) = \frac{3}{2\pi h^3} \begin{cases} \frac{2}{3} - \left(\frac{r}{h}\right)^2 + \frac{1}{2}\left(\frac{r}{h}\right)^3, & 0 \le \frac{r}{h} < 1, \\ \frac{1}{6}\left(2 - \frac{r}{h}\right)^3, & 1 \le \frac{r}{h} \le 2, \\ 0, & \frac{r}{h} > 2. \end{cases} \tag{5.4}$$



Using the same kernel, the current density ρ_j of a particle j can be estimated from a set of surrounding N neighbor particles as

$$\rho_j = \sum_{i=1}^{N} m_i W\left(\frac{||r_j - r_i||}{h_i}\right). \tag{5.5}$$

SPH simulation is then performed in discrete time steps where in each step, for all particles, pressure is computed from the particle densities and particle positions are updated using a numerical integration scheme such as semi-explicit Euler.

5.2.2 Visualization

Visualization of the resulting data sets is challenging, as the visualized quantities need to be derived from a large number of surrounding particles during evaluation of the kernel functions. Numerous approaches for visualization of particle-based simulations exist and offer a wide range of combinations of memory requirements, preprocessing times, visualization quality, and rendering speed. The applicability of a rendering method is additionally dependent on the type of simulated fluid and the information contained in the data. In the context of liquid simulations, visualization is mainly focused on the material boundaries, i.e. the surface of the liquid, and does not need to represent the interior, homogeneous structure. In the simulation addressed in this chapter, however, the internal structures and matter distributions are of grave importance to understanding the simulated phenomena and thus need to be considered in the visualization.

From an algorithmic point of view, visualization techniques can be roughly classified into three approaches:

Resampling to Scalar Fields. The most prominent visualization method applies a preprocessing step to resample the particle quantities to a Cartesian grid [CSI09, NJB07]. From the resulting data, surfaces can be extracted using marching cubes [LC87] or direct volume rendering. While this technique is able to produce high-quality images of both surfaces and fluid bodies, preprocessing requires significant amounts of time and memory. As the cell size should be in the range of half the particle spacing to capture fine details [AIAT12], the memory requirements of the resulting grid as well as the extracted triangle surfaces can quickly exceed the available memory for complex scenes.

Adaptive data structures can be used to exploit the advantages of uniform grids, but restrict the workload to a narrow band around the fluid surface, e.g. [YHK09, AIAT12]. Akinci et al. [AAIT12] demonstrate an efficient, parallel processing pipeline to generate adaptive triangle meshes from liquid simulations. The method of Bridson [Bri03] uses sparse grids, which store adaptively refined bricks in this band. Dynamic tubular grids and RLE encoded level-sets [NM06, HNB*06] do not resort to any brick representation and encode the voxels in the dynamically evolving narrow band. An out-of-core variant of dynamic tubular

grids was presented by Nielsen et al. [NNSM07]. The work on dynamic sparse grids in the context of numerical simulations has grown into the open source C++ library OpenVDB [Mus13]. It supports hierarchical adaptive data structures along with a number of processing operators.

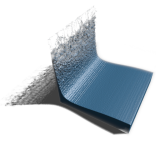
Fraedrich et al. [FAW10] use a perspective grid, which discretizes the view frustum to restrict resampling to the visible space at the cost of recurring resampling for every new view. Their work relies on a particle LoD structure to effectively reduce the number of resampled particles with decreasing sampling frequency [FSW09]. The particle set is represented at ever coarser resolutions with fewer and fewer particles, which are computed in a preprocess based on specific particle merging rules. The perspective grid is very memory-efficient, but it requires resampling large sets of particles in every frame; this process vastly dominates the rendering times and limits the performance significantly, restricting the solution from performing at interactive rates for simulations as large as the Millennium Run.

The effect of the resampling scheme on the quality of a surface has been studied intensively [MCG03, ZB05, SSP07, APKG07, AIAT12, OHB*13]. In the most recent approach by Yu and Turk [YT13] the resampling kernels are aligned locally with the distribution of nearby particles via principal component analysis.

Direct Rendering. To avoid the memory and processing time requirements of particle resampling, SPH data can be rendered directly by evaluating the SPH kernels at sampling points along the view rays by using metaballs for isosurface rendering [ZSP08, KSN08] or sphere intersections on the GPU [GSSP10]. Gribble et al. [GIK*07] perform direct ray-sphere intersections on the CPU using a grid-based acceleration structure. An isosurface extraction technique that works directly on the particle set was introduced in [RB08]. In a recent work, Yu et al. [YWTY12] generate a single surface mesh, whose vertices are then advected along with the nearby particles during simulation.

Screen-Space Techniques. Especially in computer games, screen-space approaches for rendering isosurfaces in SPH data have gained attention due to their real-time capability. These methods first generate a depth imprint of the fluid surface by e.g. rasterizing each particle as a small sphere, and reconstruct the fluid surface from the resulting depth map. Müller et al. [MSD07] reconstruct a triangle mesh in this way, and further improve the surface via mesh smoothing. Alternatively, the depth buffer is directly rendered [CS09] after a smoothing filter has been applied to reduce noise and visible sphere artifacts. Commonly, bilateral filtering [TM98, PM90] or a less computationally expensive, but lower-quality separable version [PV05] are used for depth smoothing, but more sophisticated filters that specifically address fluid surfaces have also been proposed [vdLGS09].

To also render the fluid body, order-independent splatting of transparent particle sprites has been used [ALD06, LFH06, HE03, HLE04, FSW09]. This approach is fast because it does not require any time-consuming evaluation of the SPH kernels, yet it can only produce an approximate visual representation because it blends together particle footprints in screen space and does not reconstruct a spatially continuous 3D field in object space. Volume attenuation can only be simulated when the particle set is



sorted in every frame, but even in this case the technique is not able to faithfully represent surface-like structures as shown in [FAW10].

5.3 System Components

Our proposed visualization system for very large SPH data takes as input a set of particles, each given by a position in the 3D simulation domain, a smoothing length specifying the particle's support, and additional scalar attributes like density. In a preprocess, the simulation domain is partitioned adaptively into blocks of different size using octree subdivision, and each block is discretized into a 16^3 grid. As a criterion for steering the subdivision process we use the particles' smoothing lengths: It indicates the size of the structures that are resolved by a single particle, which is up to 100,000$\times$ smaller than the domain extent in the Millennium Run. We subdivide the domain such that the cell size of a grid is equal to the smallest smoothing length of all particles influencing the grid's values.

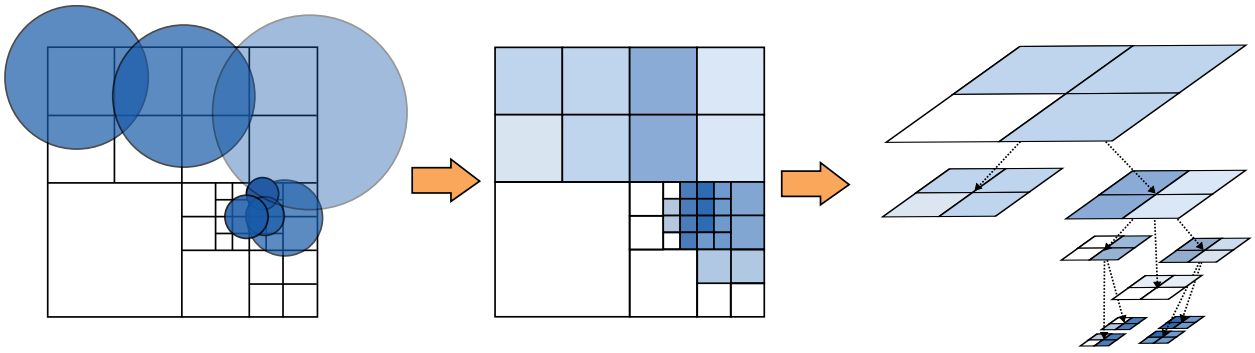


Figure 5.2: Schematic overview of the preprocessing pipeline: The particle domain is adaptively subdivided based on the smoothing length and distribution of the particles binned to it, until a predefined brick size is reached. Bricks are arranged in an octree data structure, where the data at inner nodes is created by averaging the respective child node data.

The preprocess generates an adaptive octree structure, where the leaf nodes represent the blocks at different resolution levels. The particle attributes are then resampled into the grids at these nodes, and the grids at the inner nodes are generated bottom-up via averaging the data values at the grids of the respective child nodes. Upon construction of its parent grid, the grid values are compressed and written to disk. The preprocess is illustrated in Fig. 5.2.

Our framework (see chapter 4 for a general overview) is configured as follows:

- The data generated from the preprocess is accessed by a standard streaming supplier as detailed in section 4.4.
- A single brick processor is used to perform GPU based decompression
- A single 3D data pool caches the uncompressed bricks on the GPU

- The brick traverser performs direct volume rendering as introduced in section 2.4

Figure 5.3 depicts our framework configuration along with the mentioned changes from the standard configuration. In the following, we will elaborate in detail on our custom preprocess and describe the decompression brick processor and DVR brick traverser.

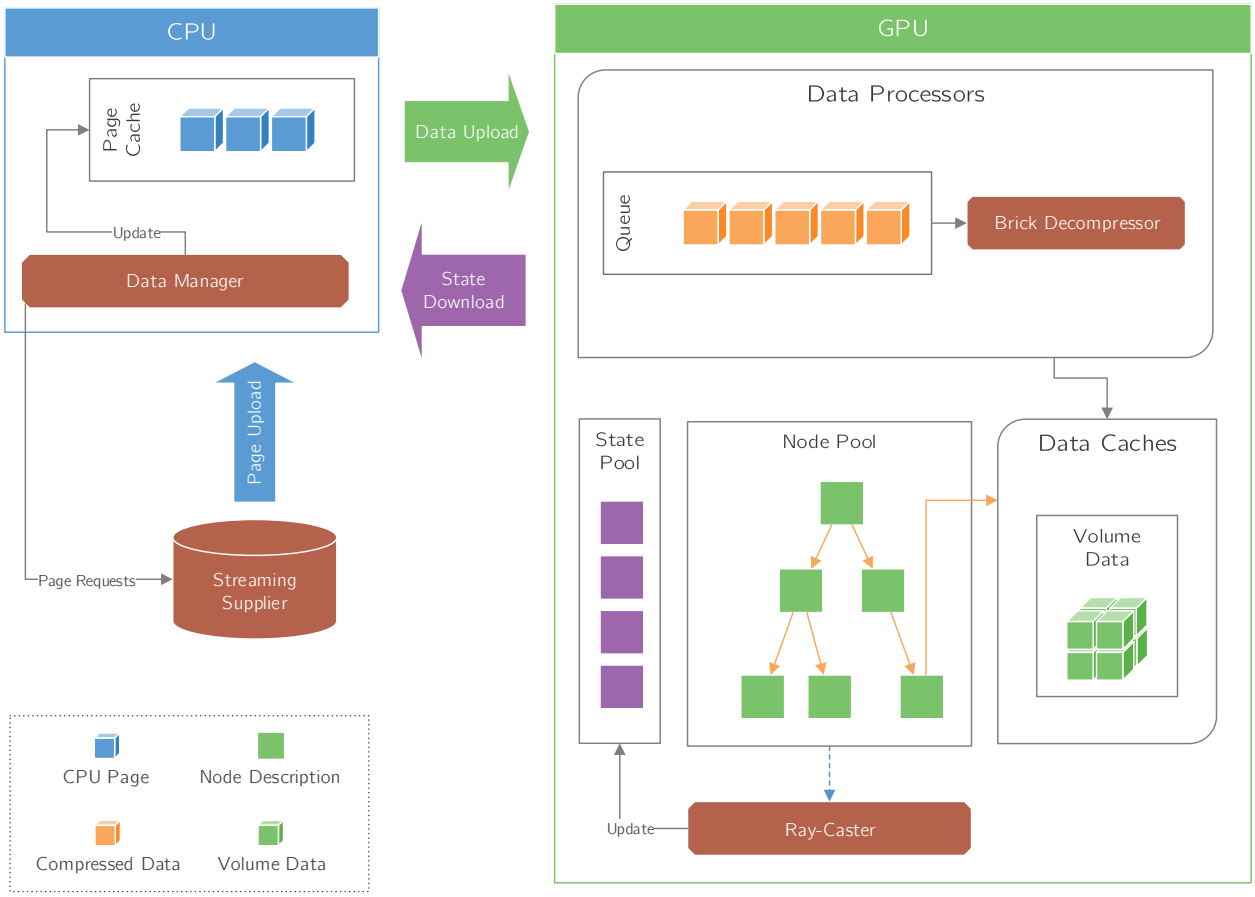
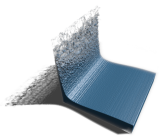


Figure 5.3: Customized components of our rendering framework: A streaming supplier loads pages of compressed bricks, which are decompressed in a brick processor and stored in a single 3D texture data pool. The ray-caster performs direct volume rendering of the density data.

5.4 Preprocess

The preprocess required to convert the particle data into the adaptive octree hierarchy is specifically tailored to exploit the highly parallel nature of modern CPUs and CPU clusters as well as GPUs. It is divided into two major steps: In a first step, the simulation domain is subdivided into cubic sub-domains of fixed spatial extent, with a side length of $\frac{1}{64}$ of the domain in each dimension. We will refer to these sub-domains as *superbricks*. All particles are binned on disk into their respective superbricks depending on their position and smoothing length. Particles at superbrick borders are duplicated in this process.



In a second step, for each of the superbricks we create an octree structure—a *subtree*—as explained in detail in section 5.4.1. The subtrees store an adaptive multiresolution representation of the 3D scalar fields encoded in the particle attributes. Each subtree is then inserted into the final octree at the respective position given by the superbrick's coordinates. The level of the root node of each subtree is determined by the extent of each superbrick—level 6 in our case, with 0 being the coarsest level. All bricks are compressed using a fast wavelet-based GPU compression scheme, which will be detailed in section 5.5.

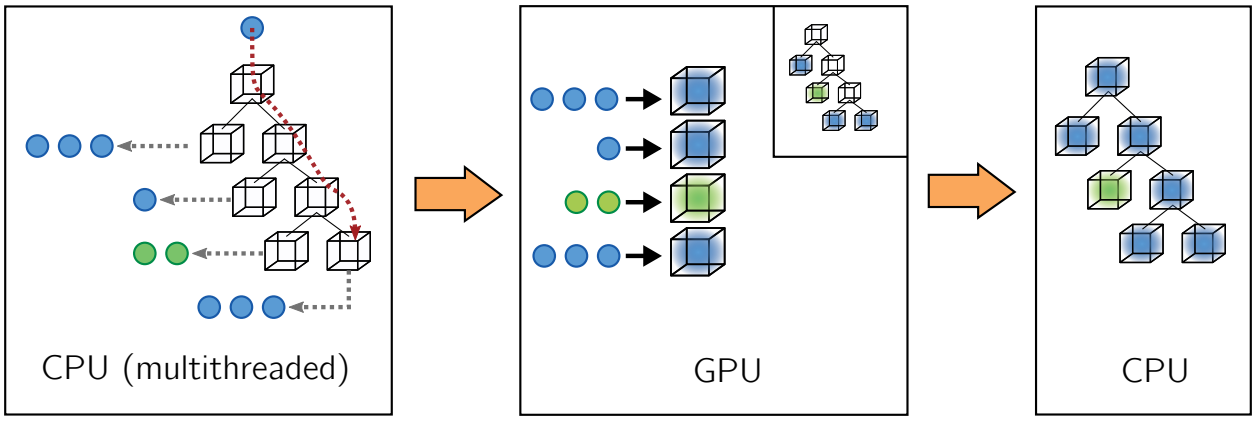


Figure 5.4: Subtree creation: A particle's position and smoothing length indicate the leaf node at which the particle is stored. The tree is traversed top-down until this leaf node is reached and the particle is stored; missing nodes along the path are generated. The particles of each leaf node are then sampled into the discrete grid, and coarser levels of detail are created bottom-up. Green nodes were created via subdivision but do not contain any particles of correct resolution, and they are deleted finally.

After each subtree has been built and inserted into the final tree, the coarsest levels are created from the finer levels by recursively merging 8^3 subtree root bricks into a single brick and averaging the corresponding voxel values.

5.4.1 Subtree Creation

With the domain divided into reasonably-sized superbricks, each subtree can be created individually, either on a single PC or distributed on a larger cluster. In combination with the LoD metric proposed in this section, the required particle data as well as the output volume data is usually small enough to be processed completely in-core. However, it is worth noting that this process is built upon the same out-of-core octree data structure as our rendering system described in section 5.6, and thus scales well to systems with limited memory or data sets with very large superbricks.

The subtree creation is based on the observation that the particle distribution and smoothing lengths exhibit very high variance over the entire domain. This allows us to adapt the effective sampling resolution on a per-brick basis: The distance between two grid points is chosen to be less or equal to

the smallest smoothing length h, assuring that no particle contributions get "lost". Given the desired grid size of a brick b and the domain extent d, the required octree level l can then be determined as

$$l = \left\lceil \log_2 \left(\frac{d}{h \cdot b} \right) \right\rceil$$

with $l = 0$ being the coarsest level with a grid size of b.

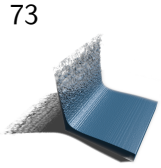
For each particle, the subtree is traversed top-down until the according level is reached. All intermediate nodes are created during traversal. To create the hierarchy, each non-leaf node is always subdivided into eight children. Child nodes which do not contain any particles at the required level are marked as *oversampled* and will not be inserted into the final octree. With each octree leaf node, a list of particles overlapping this node is stored. This whole process is parallelized over the particle data to maximize performance. An illustration is provided in Fig. 5.4.

Once the nodes have been created, an empty buffer is allocated for each leaf node to store the grid values at this node, and the particle list for the node is transferred to the GPU. Particle resampling is then performed in a CUDA kernel with one thread associated to each grid vertex—due to the small average number of particles per node, this gathering approach exhibited superior performance to particle scattering as it can be implemented without any write-conflicts. Each thread block first loads a portion of the particle list into shared memory to speed up the process. For each grid vertex, the contribution of each particle is then determined by evaluating a cubic spline kernel based on the particle's smoothing length.

Once all leaves of an octree have been created, coarser levels are generated by recursively merging each node's children, where the value of each voxel is averaged from the respective child voxels. All non-oversampled bricks are then compressed and stored within the final octree on disk.

5.5 GPU Data Compression

To reduce the total data size and, consequently, the disk bandwidth required during rendering, all bricks are compressed using a discrete wavelet transform (DWT) followed by entropy coding. We employ the GPU for both compression and decompression, as it has been shown to achieve superior throughput compared to CPUs [TRAW12]. Our GPU compression scheme for volumetric data is similar in spirit to the one proposed by Treib et al. [TBR*12]: First, the input floating point values are quantized with a user-defined quantization step. A 2-level 3D reversible integer DWT using the CDF 5/3 wavelet is then performed on each brick by applying the lifting scheme [Swe98]. More than 2 levels do not significantly improve the compression rate. Finally, the wavelet coefficients are compressed using Huffman coding. The decompression performs the inverse operations in reverse order.



In order to efficiently use the parallel processing power of the GPU, our compression layer processes multiple bricks in parallel (up to 256 in the current implementation). The brick-wise 3D DWT is implemented in a CUDA kernel. For each brick, we launch one thread block consisting of 256 threads which performs the following operations:

- Cooperatively load one brick consisting of 16^3 16-bit integer values (corresponding to 8 KB of memory) from global into shared memory.
- Perform the first-level 16^3 DWT. In this step, the threads are grouped into a 16×16 grid, i.e. each thread processes one row of data along each dimension.
- Shuffle the low-pass and high-pass coefficients from interleaved order (due to the lifting scheme) into subband order: Each thread loads 16 values from shared memory into registers and, after a block sync, stores them to their target positions. In this way, no additional "staging buffer" in shared memory is required.
- Perform the second-level 8^3 DWT, with the threads grouped into an $8\times8\times4$ grid.
- Shuffle the 8^3 coefficients into subband order.
- Store the final result to global memory.

This approach is very efficient because each element is read from and written to global memory exactly once. The decoder writes directly into the volume ray-caster's 3D texture using 3D surface writes to avoid an extra `memcpy`.

When using larger bricks (e.g. 32^3), a brick is too large to fit into shared memory at once. In this case, we fall back to a three-pass algorithm, i.e. we run three CUDA kernels sequentially:

1. Each thread block loads a $32\times32\times1$ slice from global to shared memory, does a DWT in the X and Y dimension, and stores the result back to global memory.
2. Each thread block loads a $32\times1\times32$ slice, does a DWT in the Z dimension, and stores the result back to global memory.
3. Each thread block loads a $16\times16\times16$ block of low-pass coefficients, does a DWT in all 3 dimensions, and stores the result back to global memory.

For the final entropy coding stage, we employ a Huffman coder as in [TRAW12]. However, we skip the preceding run-length encoding step, as it does not improve the compression rate in our case.

To justify the use of lossy compression, Fig. 5.5 shows images created from compressed and uncompressed data, where the particle quantities are stored as 32-bit floating point values. This coincides with the precision of the input data. As can be seen, the compression introduces no visual artifacts. For bricks of size 16^3, decompression is performed at an average output rate of 2.4 GB/s, yielding a decompression time of 3.18 μs per brick.

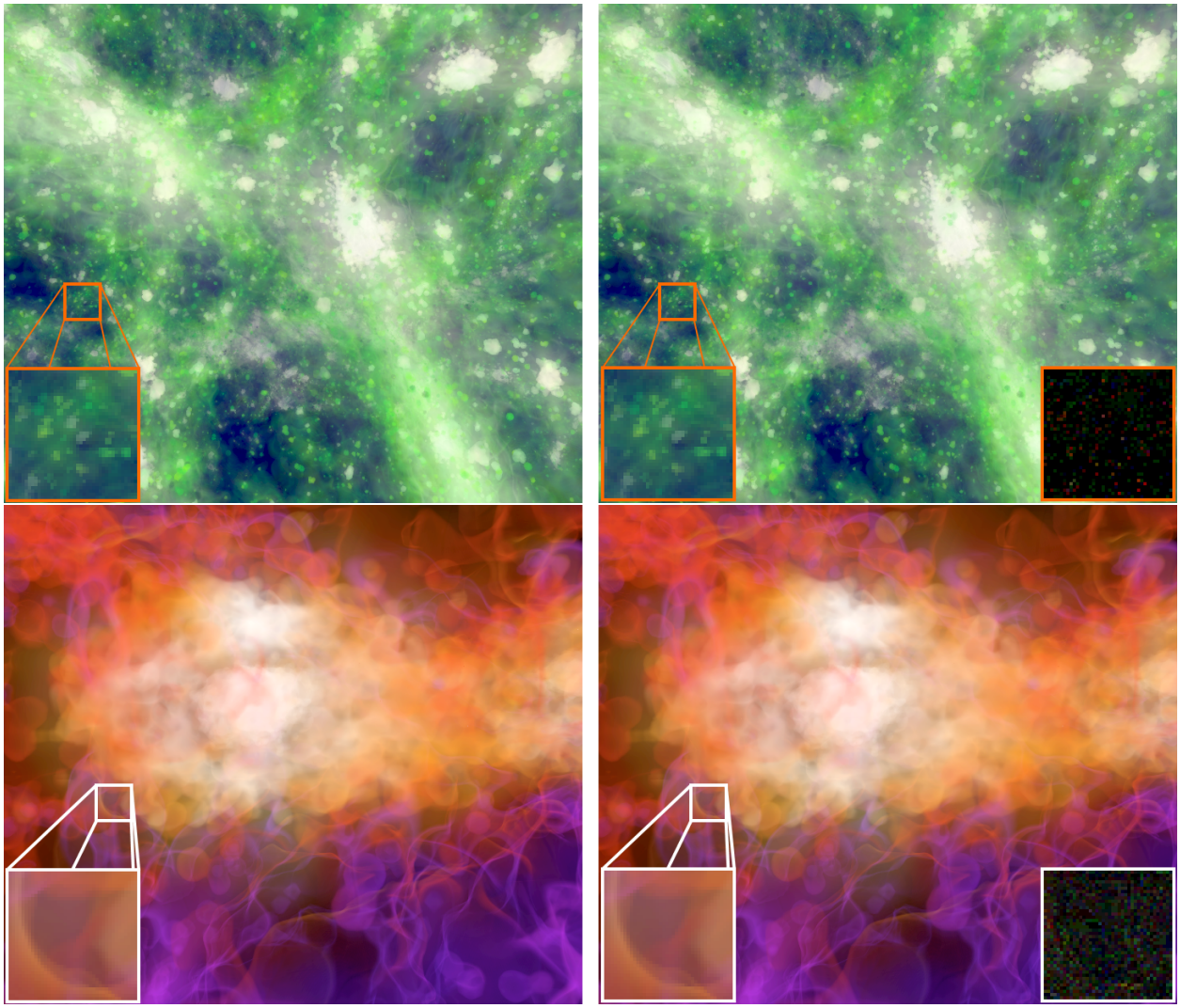
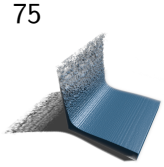


Figure 5.5: Quality comparison between the uncompressed (left) and compressed (right) data. The difference image is scaled by a factor of 20.

5.6 Volume Rendering

The brick traverser visualizes each brick using transfer function preintegration [EKE01]. In addition, we allow for quadrilinear filtering using data from the brick's parent to switch more smoothly between different octree levels. A characteristic of our top-down style preprocess is the refinement of only those octants of a brick which require a higher resolution due to the determined per-particle LoD. Always creating all 8 children at once would increase the total number of bricks by more than a factor of 3. Whenever a child node is not available during rendering, we fall back to rendering the corresponding octant of its parent brick. To make this possible, the data manager ensures that a brick's volume data is always uploaded to the GPU before descending further.



The decisions to descend or render are driven by a user-defined desired maximum pixel error in screen space. In addition—as it is common when rendering this type of astro-physical data—a focus distance along with a region of interest can be defined to reduce the amount of displayed information. We also employ a variant of β-acceleration [DH92]: Depending on the currently accumulated opacity α, a ray may decide to not further descend down the hierarchy if the screen space error induced by rendering the current brick is less then $e_s \cdot (1 + \alpha \cdot k)$, where e_s is the desired screen space error and k is a user-defined factor ≥ 0. In our experiments, choosing $k = 1.0$ increased the rendering performance by about 20%, with minimal effect on the output quality.

Aside from these adaptations, we use the standard traversal and GPU feedback algorithm introduced in section 4.5 and 4.3. To further reduce the latency introduced by low disk bandwidth, whenever the data loader is not busy waiting for data, the streaming supplier performs prefetching of data in a spherical region [NNT*05].

5.7 Results

We have evaluated the performance and memory consumption of our Direct3D 11 rendering system and the preprocessing pipeline on a single desktop PC equipped with a dual Intel Xeon X5560 processor (quad-core, 2.80 GHz), 24 GB of main memory and an NVIDIA Geforce GTX Titan with 6 GB of video memory. Timings and working set sizes always refer to a viewport of 1200×800 pixels with a desired screen-space pixel error of 1.0, though some images have been cropped for layout reasons. Two data channels were sampled from the particle data: Dark matter density and density-weighted velocity dispersion, both quantized using a quantization step of 0.01 after logarithmic rescaling, thus each channel can be stored using 2 bytes per voxel. Our variant of β-acceleration was enabled with $k = 1.0$. All I/O was performed on a RAID-0 configuration of two hard drives with a maximum throughput of 190 MB/s.

Table 5.1: Preprocessing times for one time step of the Millennium Run.

Action	Time (h:m)
Superbrick creation	00:34
Octree creation	04:41
Particle resampling	19:40
Subtree merging	00:44
Compression	07:28
Disk I/O	02:13
Total	35:20

5.7.1 Preprocessing

Preprocessing times for one time step of the Millennium Run simulation are given in table 5.1. The size of each brick was chosen to be 16^3 voxels, including an overlap of one voxel at the borders to ensure

correct interpolation between bricks. The timings include all data transfers between main memory and GPU memory.

Table 5.2: Influence of different brick sizes on file size and rendering speed. For comparison, the size of the original particle data is 225 GB (for reference, [FSW09] required 198 GB). *Frame* gives the rendering times for Fig. 5.1, left.

	File size			
Brick size	Raw (GB)	Comp (GB)	Ratio	Frame (ms)
8^3	1,655.56	540.47	0.33	276
16^3	2,916.85	512.61	0.18	213
32^3	7,967.81	998.34	0.13	168

The selected brick resolution has proven to be a good trade-off between rendering speed and memory requirements: Small bricks result in a better adaption to the local feature sizes. They can thus approximate larger regions with coarser resolutions, greatly reducing the overall memory requirement. On the other hand, they increase the overhead of overlaps—for bricks of resolution 16^3 with one voxel of overlap, only 67% of the voxels contain non-redundant information—and reduce the effectiveness of per-brick compression. They also diminish rendering performance due to the octree traversal overhead and the more incoherent layout of the volume data in GPU memory. Table 5.2 compares compressed and uncompressed file sizes as well as frame rendering times for different brick sizes. Rendering performance is given for the screenshot in Fig. 5.1 (left). As can be seen, a brick resolution of less than 16^3 voxels even increases the total file size as the effectiveness of the compression decreases drastically.

To show the effectiveness of adaptive subdivision, table 5.3 shows statistics for the different octree levels, starting from the "superbrick level" 6 up to the finest level.

5.7.2 Rendering

We compare the rendering quality and performance to particle splatting [FSW09] and perspective grid ray-casting [FAW10]. In all comparisons, we will refer to the presented system as multi-resolution direct volume rendering (MRDVR). For MRDVR, we present the rendering times as well as the required GPU memory for the visible uncompressed bricks, which we denote as the *working set size*.

While the existing approaches have been evaluated on an older Geforce GTX 280, experiments have shown a speedup of both techniques of a factor of about 3 on the target GPU (Geforce GTX Titan). While the compute and rasterization performance has experienced a massive increase over the last years, the limiting factors have shown to be the texture fill rate and memory bandwidth due to the high amount of overdraw and blending, which have only increased by a factor of 4 and 2, respectively. Thus, the particle splatting approach performs with an average of about 30 ms per frame, while the perspective

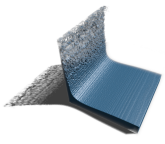


Table 5.3: Per-level statistics. Average *Children* are given per internal (non-leaf) node. *Oversampled* bricks are not contained in the final octree. # *Particles* is the number of particles whose smoothing radii dictate sampling on the respective level.

	# Bricks			
Level	Stored	Oversampled	Children	# Particles
6	262,000	0	7.9	2,855,365,100
7	2,074,646	21,290	5.5	2,797,593,157
8	10,487,783	4,789,657	3.2	2,227,115,066
9	22,747,376	33,640,354	2.3	2,282,715,327
10	27,027,644	65,031,897	2.0	2,105,993,312
11	20,636,515	60,094,086	1.9	1,335,442,527
12	9,015,712	28,574,277	1.9	474,714,085
13	1,208,903	3,953,823	–	45,567,944

grid ray-caster can only achieve hardly interactive rendering times of over 3 seconds per frame for this data set.

Figure 5.6 shows the differences in quality and rendering time between the unordered splatting approach and MRDVR. The dominant structures are visible in both approaches, yet MRDVR allows for much more complex transfer functions.

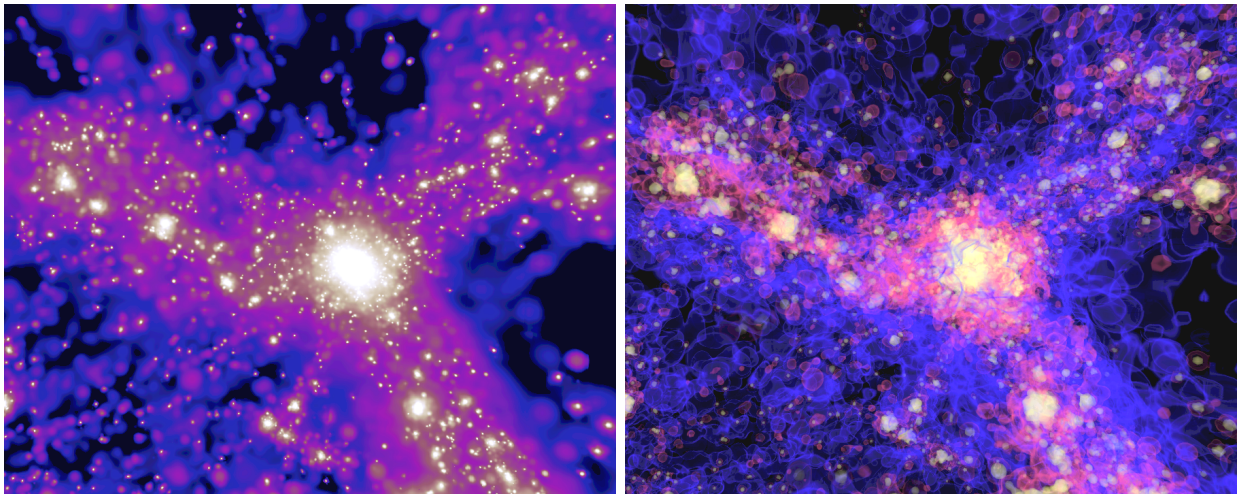


Figure 5.6: Comparison between unordered splatting (left, 18 ms) and MRDVR (right, 122 ms). Unordered splatting can not reconstruct fine, surface-like structures, and the lack of occlusion hinders depth perception.

Figure 5.7 displays the results with and without β-acceleration. Small features in the background are well preserved while the rendering time exhibits a speed-up of about 20%. The working set size is

reduced by 30%.

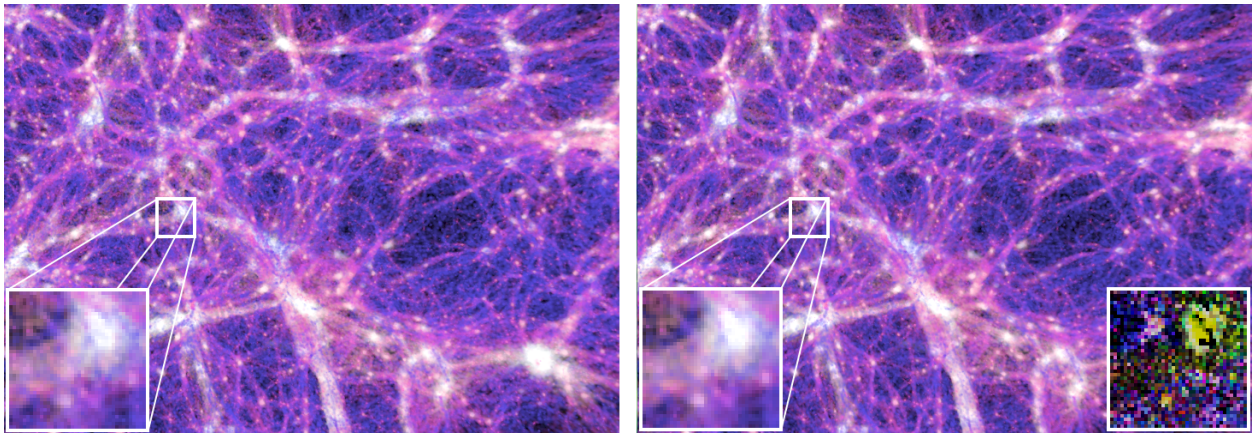


Figure 5.7: Rendering results without (left, 268 ms) and with (right, 220 ms) β-acceleration. The working set is reduced from 2.44 GB to 1.74 GB. The bottom right shows a difference image, scaled by a factor of 20.

Finally, we performed a trip through the data set and recorded rendering times, working set sizes and memory transfer sizes. The flight features a wide range of speeds as well as close-up and distant views all across the domain. Results can be seen in Fig. 5.8. The dominant drops in the working set size are due to rapid viewport changes, causing a large number of GPU cache misses. As coarser levels of detail are being rendered in these cases, the performance improves temporarily at the cost of image quality. However, the missing data is delivered within a few frames.

5.8 Conclusion

In this chapter we analyzed the use of our framework for the interactive visualization of large SPH data sets. The key conclusion is twofold: Firstly, by resampling the particle attributes to a regular, yet adaptive hierarchical octree grid and compressing the resampled data using a wavelet-based scheme, only a moderate increase in memory is introduced. Compared to the compressed particle data including a particle LoD hierarchy, a factor of 2 could be demonstrated. Secondly, since any resampling of particle attributes at runtime can be avoided, rendering performance increases significantly. At the same quality, our tests have shown a performance increase of about 10-15$\times$ compared to the perspective grid approach. Compared to order-independent particle splatting, our system is about 5-10$\times$ slower, yet it comes with significantly higher quality and more flexible visualization options like isosurface rendering or gradient shading.

In the future we will investigate the integration of our desktop system into a remote, client/server-based visualization infrastructure supporting astrophysicists in their explorations. Since our method avoids any time-consuming processing of particles at runtime, it is especially suited for multi-user scenarios, where

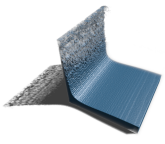


Figure 5.8: System performance during a flight through the data set.

at the same time images from different perspectives are requested. Since it is not possible in general to keep the working set of more than one user on the GPU, the working sets of all users have to be swapped in and out of the GPU periodically. The memory and, thus, bandwidth-aware design of our system accommodates an efficient realization of these operations.

6

Binary Grids for Giga-Particle Fluid Rendering

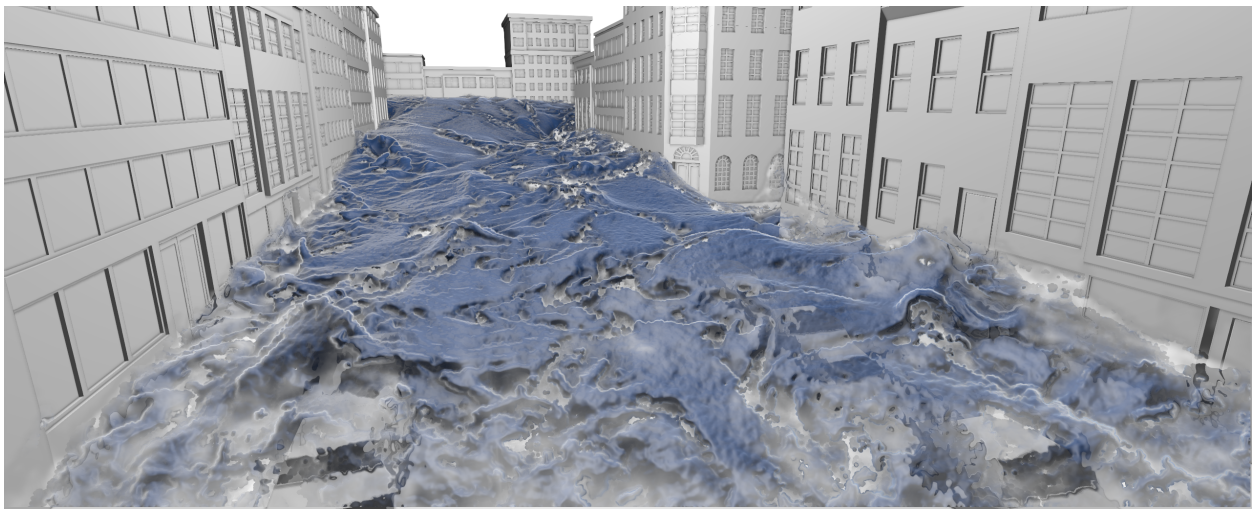
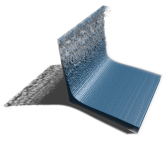


Figure 6.1: Rendering a 500 million particle simulation, preprocessed in 12 seconds, at 72 ms per frame on a 1280×720 viewport using ray-sphere intersections, image-based surface smoothing, volumetric foam and approximative refractions on a single PC.

6.1 Introduction

The preceding chapter demonstrated the massive scalability of our system for particle data comprising tens of billions of particles of largely varying radii, distributed over a comparably large domain. The demonstrated solution, however, relies on exhaustive preprocessing in the order of several hours to reduce the memory requirements arising from a densely sampled scene with volumetric properties. In the following, we derive a specialized compact data structure for the visualization of particle-based fluid simulations that significantly reduces the memory requirements while also greatly decreasing preprocessing time.



Over the last years, particle-based methods have become increasingly popular for simulating high-resolution violent liquid effects with possible fragmentation, splashing, and large deformations. Especially the Fluid-Implicit-Particle (FLIP) [BR86] method has become a standard in visual effects. In contrast to the already presented Smoothed Particle Hydrodynamics (SPH) [Mon92], where the state of a particle state depends on its surrounding particles, FLIP performs the projection step to enforce incompressibility on an Eulerian background grid. The simulated particles are collisionless and do not require knowledge about each other. Thus, compared to SPH, FLIP methods are computationally lightweight, in particular because they do not need to update and frequently query a spatial search structure to resolve particle neighborhoods. This way, high-resolution FLIP simulations can easily use hundreds of millions of particles.

In computer animation FLIP was introduced by Zhu and Bridson [ZB05], and further advanced to accurately handle obstacles [BBB07] and simulate two-phase flows [BB12]. Ando and co-workers [ATW13] use an adaptive unstructured background grid in FLIP and adjust the density of particles accordingly. Recently, Cornelius et al. [CIPT14] combine an SPH solver for pressure projection and boundary handling with a FLIP simulator to avoid having to use a grid at all. Since the particle advection step in FLIP does not require any particle-particle interactions, it is significantly less involved computationally than SPH. The resulting representation also relies on particles of certain radius and, thus, the general principles and challenges regarding rendering are very similar to SPH and covered in section 5.2.

In contrast to the particles used in SPH gas dynamics simulations, fluid particles are of significantly smaller and often fixed radius. In addition, the resulting visualization is only interested in the implicit surface defined by the particles and can safely assume a homogeneous mass distribution inside the fluid body. While in general the same rendering algorithm—splatting of all particle densities in combination with direct volume rendering or, more suitable in this case, isosurface ray-casting—could be applied, the question arises if storage of density distributions is a necessity.

The problem of efficient and interactive rendering of the gigantic particle sets resulting from large-scale particle simulations has not gained much attention up to now. In particular previewing tools for giga-particle-scale simulations are not existing, yet they are important to analyze and control the visual effects created by such simulations. For instance, to detect failures early, to investigate specific fluid details, or to look at a scene from different viewpoints to find the best angle. Previewing tools do not need to show the fluid effects in final-film-quality, but they should show a plausible rendering that reveals the fluid surface and splashes, and support interactive scrubbing through the animation to analyze the dynamics over time.

Some visual effects companies have started to engage in the issue of rendering very large particle simulations. See, for instance, the 2013 release of Houdini by Side Effects. However, the classical approach, which first computes a 3D scalar field by summing up the particle smoothing kernels and then reconstructs a polygonal isosurface from this field, is not suited for previewing. For instance, to capture all fluid details in the FLIP simulation shown in Fig. 6.1, a regular grid of size $4K^3$ is required

to discretize the scalar field. This results in several gigabytes of data per time step and many minutes of preprocessing until the fluid surface is available.

Current real-time rendering approaches for particle-based fluid simulations can handle a few million particles [MSD07, vdLGS09, GEM*13], but they do not scale well with the number of particles and time steps. This is due to bandwidth and memory constraints which limit the amount of particles that can be streamed and processed at reasonable rates. Another constraint is often caused by the use of a rasterization-based rendering pipeline. Especially when many isolated splashes occur and occlusion culling cannot be performed effectively, the required rasterization capabilities go beyond what is available today. Real-time approaches which can classify and render volumetrically splashes out of the box at runtime have not yet been proposed.

6.2 Application Goals

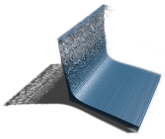
We introduce a novel binary voxel representation for particle positions that can be efficiently integrated into our ray-casting framework to enable interactive rendering of giga-particle fluid simulations. Our representation relaxes bandwidth limitations throughout the entire previewing pipeline and enables parallel GPU processing for high performance rendering. It further reduces the memory requirements significantly and shows no qualitative difference to the initial data representation. The regular voxel structure permits highly efficient data access operations, which we exploit to classify foam particles and perform ray-surface intersection testing and volumetric compositing with early ray-termination.

All data processing tasks are embedded into our ray-guided ray-casting framework. Based on spatial particle binning, which is the only required preprocess, all other processes are carried out on request during rendering. In this way we can restrict the workload, including memory transfer, to the fluid parts which are seen.

Our rendering system displays the isosurface by rendering each particle as a sphere and post-processing the resulting surface. Therefore, we have adapted and extended the total-variation-based image denoising model of Rudin, Osher, and Fatemi [ROF92] to work on oddly-shaped image domains and allow depth-dependent smoothing strengths. We demonstrate superior quality of this model over other available models, at similar speeds.

For the 500 million particle simulation shown in Fig. 6.1, our system shows the following performance features when rendering onto a 1280×720 viewport:

- A very low preprocessing time of less than 12 seconds.
- A time to first frame of less than 1 second.
- Rendering at 50 milliseconds per frame.
- Interactive scrubbing through the animated fluid.



Note, in particular, that the very low processing time distinguishes our approach from existing point based rendering approaches, for instance [KSW05, GEM*13]. Such approaches typically build upon sophisticated compression/acceleration structures and require a pre-process of the order of many minutes for the data sets we address.

To make fluid effects look more realistic, dedicated simulation and rendering techniques for foam (*white water*) have been presented, e.g. [IAAT12]. It is one concern of our system to demonstrate how foam particles can be classified and rendered efficiently for very large particle sets.

6.3 System Components

Our approach begins with an unordered set of particle positions. Additional attributes can be associated to each particle, such as a color index, a velocity index, a size indicator, or any other classifier resulting from the application that generates the particles. While we target time-dependent data, we assume all time steps to be independent from each other and, thus, all elaborations on rendering and preprocessing only refer to a single time step. For contiguous sequences, each set of particles is read, processed and cached separately.

Our framework (see chapter 4 for a general overview and Fig. 6.2 for an overview of the adaptations presented in this chapter) is configured as follows:

- The particle data is read and processed by a *particle supplier* to generate our novel binary voxel representation.
- Two brick processors are linked in sequential order to perform *foam classification* and *mipmap generation*.
- Two 3D data caches store the classified surface- and foam particles.
- An arbitrary number of 1D data caches store particle attributes.
- A novel brick traverser performs sphere ray-casting on our compact particle representation.

In the following, we will introduce all novel components step by step and detail on their integration into the rendering framework.

6.4 Particle Supplier and Core Data Structure

Upon selecting a time step and an associated set of unordered particles from external storage, the particle supplier starts by loading all particles into main memory and performs a fast, lightweight preprocess as illustrated in Fig. 6.3. The entire process including disk reads is performed in separate threads

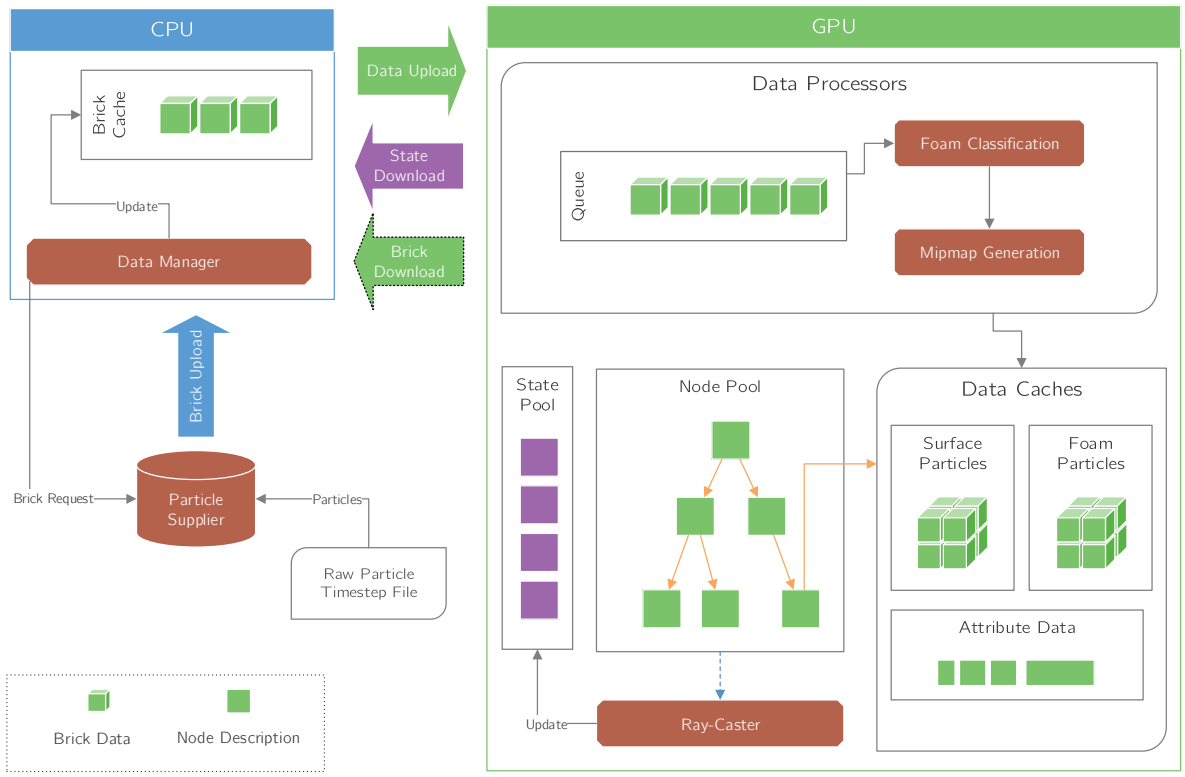


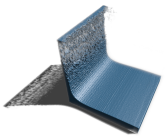
Figure 6.2: Customized components of our rendering framework: A particle supplier generates a binary voxel representation from a raw particle file during runtime. Two brick processors classify foam particles and generate an in-brick acceleration structure.

completely on the CPU, which allows the user to explore previously loaded and processed time steps in the meantime.

The domain is first partitioned into cubical bricks whose size depends on a user-selected resolution, and all particles are binned into these bricks during the loading process. On completion, the adaptive octree used in our rendering front-end is created on top of the so defined non-empty leaf bricks and is uploaded to the GPU node pool if the loaded time step is immediately visualized. Note that the created octree is time step dependent and only a single time steps' octree structure is present in the node pool at all times. If a different point in time is selected to be visualized by the user, the entire node pool is replaced; since the node descriptions are only very few megabytes in size, this operation has no noticeable impact on performance.

6.4.1 Binary Voxel Representation

After the node pool is filled with the current time steps' octree structure, rendering can start instantly without further delay. Note that at this point, no data besides the structure of the octree itself has



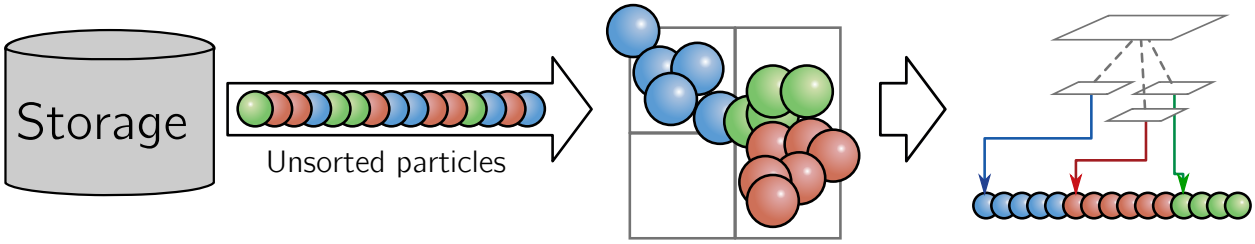


Figure 6.3: The preprocess: Particles are binned into bricks and an octree is built to encode empty space hierarchically.

been uploaded to the GPU if the time step is rendered for the first time. During rendering, the GPU detects the visible leaf bricks and signals the CPU to create our renderable *binary voxel representation (BVR)* as introduced below from the particle list associated with the bricks. All bricks requested for conversion are queued in a processing queue in the particle supplier, which is constantly processed by multiple background threads. In addition to the GPU-driven conversion, the CPU traverses all non-empty leaf bricks without generated BVR in front-to-back order in each frame and queues them in a separate conversion list. This second list is processed whenever no GPU-requested bricks are present to perform pre-conversion of not yet visible bricks. Once conversion of a time step is complete the initial particle data is discarded, leaving only our compact BVR in CPU caches.

Our BVR uses a 3D Cartesian grid to discretize the fluid domain, i.e., every brick is discretized by a grid comprising $n_x \times n_y \times n_z$ cubical cells. The size of a cell in the domain space is set by the user and directly allows to balance the system between rendering speed and particle position accuracy. In our test scenario shown in Fig. 6.1, the domain was discretized using a 4096^3 grid and partitioned into bricks of size 64^3.

When a brick is selected for conversion, the CPU generates a 3D array that stores one bit (initially set to 0) for every cell of this brick. Then the CPU loops over the set of particles which was associated to the brick in the preprocess. For every particle, it is computed into which cell this particle's center falls, and the bit of this cell is set. Finally, the bits of contiguous *sub-bricks* of size $4 \times 4 \times 8$ cells are stored in 4 32-bit integer values. In this way, every sub-brick can be stored in one single element of a 4-channel integer texture on the GPU. For instance, when bricks of size 64^3 are used, a 3D texture of size $16 \times 16 \times 8$ stores all bits of one brick, and one texture lookup operation retrieves all bits of one sub-brick. It should be noted that, internally, the conversion process works solely on 3D arrays of integers, using bit operations and address arithmetic.

6.4.2 Preprocessing Performance Details

During the preprocess, binning is performed interleaved with data loading: We fetch chunks of fixed data size using asynchronous I/O from disk and process the particles of each chunk in parallel. In this stage, particle positions are already quantized to grid cell centers to reduce the required memory as

8 bit per dimension are sufficient for our target brick sizes. Binning and quantization are performed by multiple threads (one per available CPU core) and parallelized over all particles of one chunk. We allocate a single array with enough space to store all binned particles along with the respective brick index, which can then be filled with quantized particles. Since each particle is processed by only a single thread, this operation can be carried out without any locks. During quantization, an atomic counter is maintained for each non-empty leaf brick to keep track of the number of quantized particles generated for this brick.

As the brick traverser requires a certain amount of overlapping voxels (see section 6.5), all particles inside these regions are duplicated and stored in separate arrays for each thread to avoid the need for atomic resizing of particle storage memory. After quantization of all particles is complete, the final array is resized once and all particles of the overlap regions are copied to the newly allocated memory.

After all chunks of a time step have been processed, the packed particles are sorted by their Morton-ordered brick index, and each brick finally stores a pointer to its first entry inside the quantized particle list. This pointer is easily calculated from the sum of particle counts of brick with lower index.

All the datasets we use in this work contain particle positions as 32-bit floating point values per component initially. Reading the particles on the CPU runs at disk I/O speed, delivering nearly 500 MB/s using a single solid state disk. Binning, including particle quantization and octree construction, achieves a throughput of roughly 60 million particle positions per second.

6.4.3 Attribute Storage

To handle additional particle attributes as well, we store these compactly and provide a highly efficient access operation to this encoding. The attributes of *non-empty* cells in a sub-brick are written sequentially to a 1D attribute buffer, where the order is determined by the linearized cell indices. A 16-bit pointer to the first entry in this buffer is stored for each sub-brick. The attribute buffer is managed by a virtual memory allocator as described in section 4.2.1.

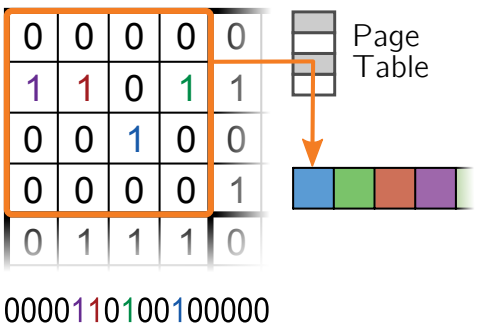
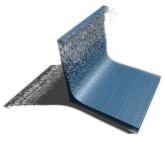


Figure 6.4: Attribute encoding: Bit-counts are combined with a single offset per sub-brick to retrieve a pointer into an attribute buffer.

During rendering, the offset of a cell's attribute is determined by counting the number of non-empty cells with lower linearized index inside a sub-brick: For an integer value v encoding one sub-brick, the offset of a cell with linearized index i in this sub-brick is computed as

$$\texttt{baseOffset} + \texttt{countbits}(v\&((1 << i) - 1)),$$



where `baseOffset` is the pointer to the sub-brick's first attribute, and the function `countbits` counts the number of set bits in an integer value. The function is available on current GPUs, so that, in combination with the binary voxel representation, a highly efficient and bandwidth-oblivious access operation is given. Attribute storage is illustrated in Fig. 6.4.

6.5 Rendering

The brick traverser interprets each bit in the BVR as a sphere of user-defined radius and calculates the first ray-sphere intersection of each view ray inside a brick. It then writes out the corresponding surface depth and color information. In addition, it accumulates the density of the volumetric foam in front of the surface as will be detailed later. To shade the resulting surface, we employ deferred shading and reconstruct normals from the generated depth buffer using central differences. The different images are then composited is illustrated in Fig. 6.5.

Since spheres in one brick can overlap adjacent bricks, an n-voxel-wide overlap is stored around each brick. The width of the overlap regions considers the maximum possible sphere radius in units of cell size. This allows for an independent traversal of each brick.

The rays are marched through the bricks in a DDA-like manner, yielding all cells that are hit along their ways. For every cell, its bit is fetched from the binary voxel representation, which also brings the bits of an entire sub-brick containing many adjacent cells into registers. When the bit is set, indicating the presence of a particle, the GPU computes the intersection between the ray and a sphere located at the cell center. To account for the sphere radius, the bits of cells in adjacent sub-bricks need to be fetched, too, and the corresponding particles are tested for intersections. If no intersection occurred, the ray steps to the next cell. Otherwise, the particle color (if available) and the distance of the intersection point to the viewer is written to the color and depth buffer, respectively, and the ray terminates. Figure 6.7(a) shows a fluid surface rendered as a set of (illuminated) spheres at original particle positions.

6.5.1 Performance Optimizations

Mipmap acceleration. Obviously, rendering performance is highly dependent on the desired particle radius, as this directly corresponds to the number of neighboring cells from which particle bits are fetched. To accelerate this process, a brick processor generates a 3D mipmap-structure of the surface particle bricks which stores a single bit at ever coarser levels, indicating if at least one of the 8^3 corresponding child cells at the finer level is overlapped by at least one particle. The brick traverser then uses this information in a hierarchical DDA algorithm to quickly skip empty space inside a brick. The resulting performance speed-up can be as large as a factor of 15 for bricks of low fill rate, since most of the rays can be advanced without intersecting a single particle.

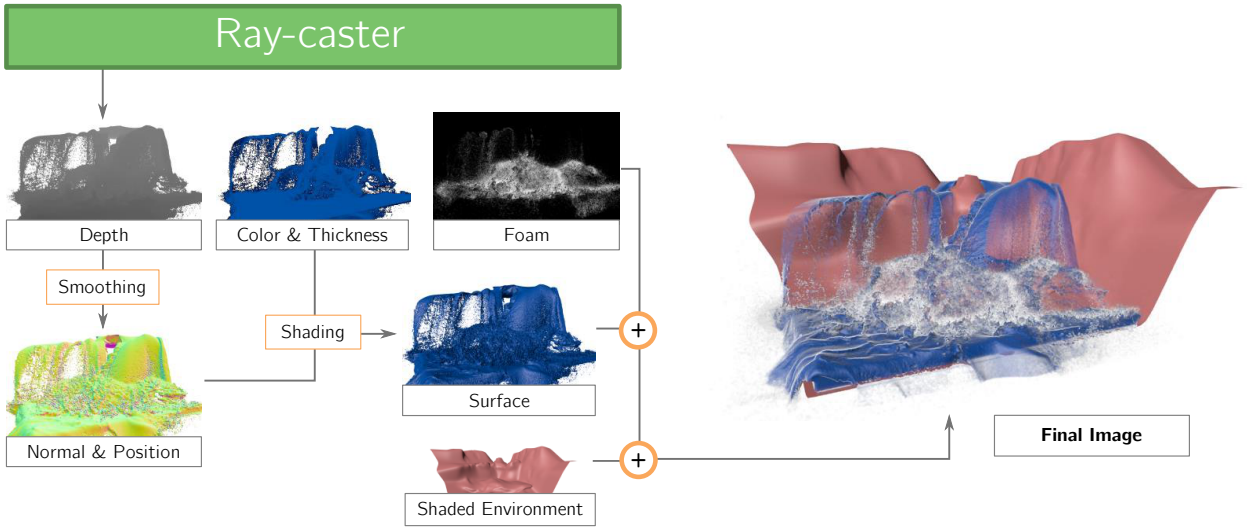


Figure 6.5: Surface depth, color and foam are output by the ray-caster. Normals and positions are reconstructed from smoothed depth values and used for surface shading. The environment is then blended with the semi-transparent fluid surface and foam.

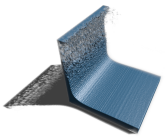
The mipmap processor works on batches of 256 bricks in multiple passes to generate the required levels bottom-up from the corresponding finer level. A single kernel is launched with one thread per sub-brick of the target level and iterates over all particles inside this sub-brick. This approach requires each thread to perform a significant amount of work, but completely avoids the need for atomic operations as each thread writes data to an independent sub-brick. To create the first level, neighboring particles are gathered inside the particle radius for each cell, and the bit of the target sub-brick is set if at least one particle exists inside this search radius. The coarser levels are then generated by checking the 8^3 bits of the next finer mipmap texture.

Sphere center shifting. Depending on the selected particle radius, the number of cells overlapped by each sphere—which directly corresponds to the number of cells to be investigated to determine a possible intersection in a cell—can be optimized by defining particle centers to be located either at cell centers or corners. The search radius r is determined by

$$r = \frac{\lceil d \rceil}{2}, \tag{6.1}$$

where d is the maximum particle diameter $\in \mathbb{R}$, expressed in units of cells. For odd-valued $\lceil d \rceil$, the resulting search radius is minimized by aligning particle centers and cell centers. For even values of $\lceil d \rceil$, this is achieved by locating the particle centers at the centers of the dual grid—i.e. at an arbitrary but fixed corner of each cell. Figure 6.6 exemplary demonstrates this for an even particle diameter of $\lceil d \rceil = 2$.

Thread coherency optimizations. As stated in section 2.3.2, instruction coherency is an important



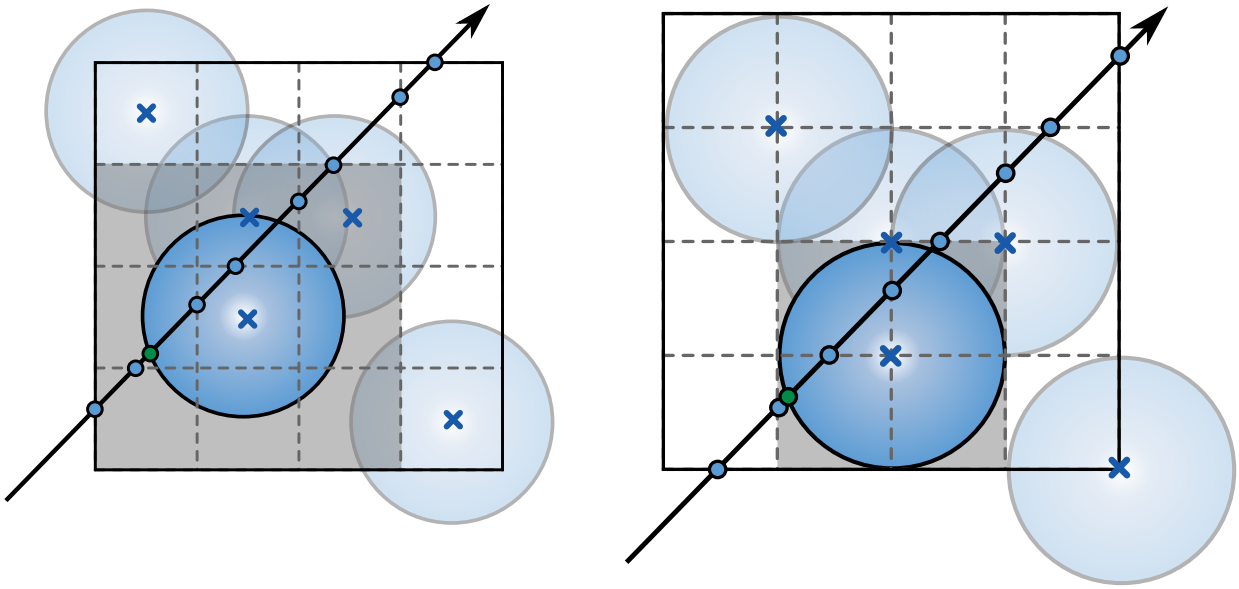


Figure 6.6: Influence radius of a particle of even diameter ($\lceil d \rceil = 2$ in this example) with particle centers (marked with an x) located at cell centers (left) and cell corners (right). Shifting the center to the cell corners reduces number of covered cells from 3^3 to 2^3 in 3D, which corresponds to the search radius that has to be considered in every cell along a ray.

factor in the overall performance of the rendering system. In a naive approach, sphere ray-casting is performed by iterating over all cells inside the search radius and, for each cell that contains a particle, intersecting the current view ray with the defined sphere. Regardless of whether an intersection was found, all neighbors need to be tested as only the closest intersection inside the currently examined cell is considered, which is not necessarily the first one determined. This results in a fixed number of loop iterations with potentially only a few threads of each SIMD lane performing sphere intersections simultaneously while the remaining threads encountered an empty cell in the current iteration.

To improve the instruction coherency, the whole process can be split into two operations: *Search* and *intersect*. During the search phase, all bits inside the maximum radius are examined and, if a particle is present, stored in a *local grid*. Conceptually, this grid is a small BVR brick of search diameter size centered at the current cell. Implementation-wise, it is a flat array of fixed size where each bit corresponds to a neighboring cell and indicates the presence of a particle after the search phase has completed.

In the intersection phase, all spheres represented by the set bits inside the local grid are tested for intersection along the view ray. Each thread loops over all *set* bits inside the local grid, calculates the position of the cell defined by the bit's index, and performs the intersection test. It is important to note here that this operation is only performed for the bits actually set in the local grid by extracting the next lowest set bit via the `firstbitlow` instruction. This ensures that each thread performs an intersection test in each iteration unless all potentially overlapping particles have already been tested. The total number of iterations for each SIMD lane is, thus, determined by the maximum number of

potential overlaps of all threads, which is usually significantly smaller than the number of cells inside the search radius.

To ensure the local grid is small and can be kept in registers without unnecessarily increasing register consumption, multiple passes of alternating search and intersection phases are performed depending on the particle radius by slicing the local grid along one direction.

6.5.2 Random Jittering

Figure 6.7(b) shows the same surface as in (a), but particle positions are quantized to the bricks' cell centers. To avoid the visible regular structure introduced by quantization, particle positions are randomly jittered inside the grid cells: When a particle in the cell with index (i, j, k) at time step t is rendered, it is offset by a random vector.The random vector is generated via three TEA hash [ZOC10] operations on the Morton ordered cell index and the current time step, yielding three random offsets for the x, y and z vector components. The jitter is calculated for each particle during rendering, and no additional memory is required. The maximum amount of jitter is limited by the width of the overlap region.

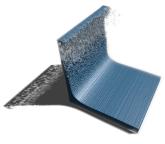
Figure 6.7(c) shows sphere rendering using position quantization and jittering. Compared to quantization only in (b), the regular structure is broken up entirely, and compared to rendering particles at their original positions in (a), the qualitative surface appearance can be judged as equal. Especially when image-based smoothing and deferred shading is performed, as described next, only very subtle differences in the renderings can be observed.

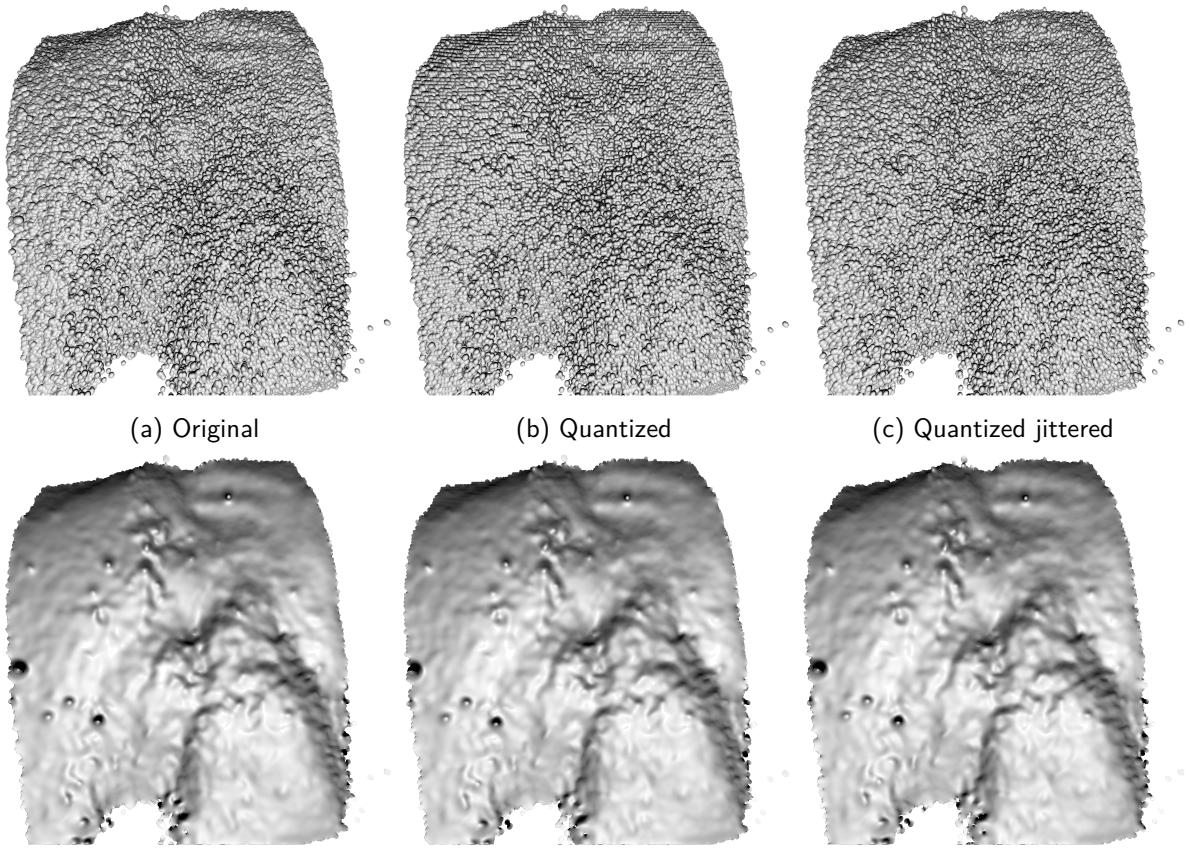
6.5.3 Image-based Surface Smoothing

To further enhance the surface's image obtained via sphere rendering, we apply a screen-space post-smoothing filter to the depth buffer. Resulting images are shown in the 2nd row of Fig. 6.7.

An effective filter has to fulfill the following goals: 1) stronger smoothing closer to the camera and weaker smoothing further away, mimicking object-space filtering in the presence of perspective projections, 2) preservation of both features and depth discontinuities, and 3) reconstruction of a "smooth" surface. While goals 2) and 3) can be achieved with the ROF de-noising model, major modifications are required to achieve the first goal. In the following we will first review the concept underlying ROF following the exposition and solver presented by Chambolle [Cha04], and we will then introduce the specific adaptations to reach our goals.

Total-variation-based image de-noising. For a piecewise smooth but unknown image $u : \Omega \to \mathbb{R}$, ROF filtering assumes an additive, stationary Gaussian noise $G(0, \sigma)$, such that an image $v = u + G(0, \sigma)$ is





(a) Original (b) Quantized (c) Quantized jittered

(d) Original (e) Quantized (f) Quantized jittered

Figure 6.7: Top row: Direct rendering as spheres. Bottom row: Same as top, but with image-based ROF smoothing applied.

observed. To make the problem tractable, a soft-constrained formulation is generally used to reconstruct u on a continuous domain Ω:

$$\arg\min_x \int_\Omega \left(\|\nabla u(\mathbf{x})\|_2 + \frac{1}{2\lambda}(u(\mathbf{x}) - v(\mathbf{x}))^2 \right) \, d\mathbf{x}, \tag{6.2}$$

where $\int_\Omega \|\nabla u(\mathbf{x})\|_2 \, d\mathbf{x}$ denotes the *total variation* of u, and $\lambda \in \mathbb{R}^+$ is an inverse data fidelity parameter dictating the amount of smoothing. Note that, to improve readability, we shorten $u(\mathbf{x})$ to u etc.

To solve Eq. (6.2), Chambolle presented a fast primal-dual solver that iteratively computes a non-linear projection $\pi_{\lambda K}(v)$ to obtain the smoothed image $u = v - \pi_{\lambda K}(v)$. On 2D domains, two dual variables $\mathbf{p} : \Omega \to \mathbb{R}^2$ are introduced and a gradient descent on the Euler equation yields the following fixed-point iteration:

$$\begin{aligned} \mathbf{p}^{(0)} &:= 0 \\ \mathbf{p}^{(n+1)} &= \frac{\mathbf{p}^{(n)} + \tau(\nabla \mathrm{div}\ \mathbf{p} - \nabla v/\lambda)}{1 + \|\tau(\nabla \mathrm{div}\ \mathbf{p} - \nabla v/\lambda)\|_2} \end{aligned} \tag{6.3}$$

Each iteration thus performs one update on $\mathbf{p}$ for the full domain Ω and $\|\cdot\|_2$ is a 2D vector 2-norm evaluated for each position $\mathbf{x}$ separately. Chambolle shows this to converge for certain values of τ to

$$\pi_{\lambda K} := \lim_{n\to\infty} \lambda \text{div } \mathbf{p}^{(n)} \tag{6.4}$$

Equation (6.3) is then discretized and, since ∇ and div are adjoint operators, forward and backward differences are respectively used. Finally, Neumann boundary conditions $(\nabla u)\nu = 0$ are imposed on $\partial\Omega$. Furthermore, Chambolle notes that $\tau = 0.25$ shows the best convergence in practice, although the theoretical reason is unknown.

Local estimate of λ. To mimic object space filtering (goal 1), we assume an *object space* stationary Gaussian density distribution $G_{obj}(0, \sigma)$ around each particle. This, if unfiltered, results in an instationary Gaussian noise $G_{scr}(0, \sigma_{ij})$ after projection, where ij are pixel positions. Non-uniform weights λ_{ij} related to G_{scr} therefore allow us to achieve high-speed previews by representing particles as spheres with $r_{obj} = \sigma =$ const in object space.

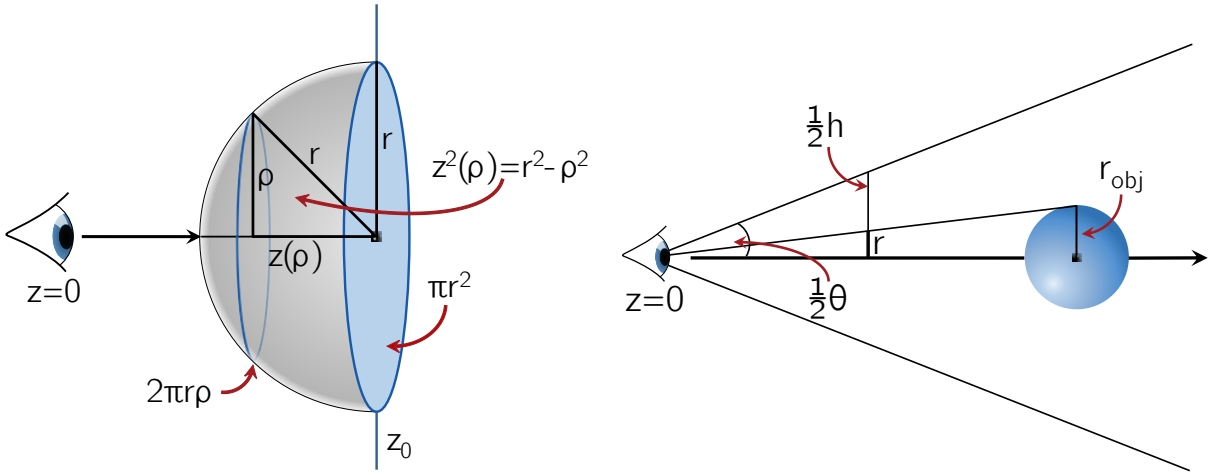
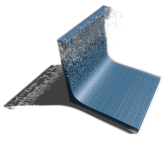


Figure 6.8: Quantities used in the estimation of the local filter weights λ_{ij}. Left: Estimating Var$[z]$. Right: Estimating the screen space radius r from r_{obj}.

Assuming the viewing ray to hit the sphere at its front-most point (Fig. 6.8 left), we estimate the expected value E$[z]$ and variance $\sigma^2 =$ Var$[z]$ of the depth buffer by integrating over concentric rings:

$$\begin{aligned} \text{E}[z] = &\quad z_0 - \tfrac{1}{\pi r^2}\int_0^r 2\pi\rho z(\rho)\, d\rho &&= z_0 - \frac{2}{3}r, \\ \text{Var}[z] = &\quad \tfrac{1}{2\pi r^2}\int_0^r 2\pi\rho\,(z(\rho) - \text{E}[z])^2\, d\rho &&= \frac{1}{18}r^2, \end{aligned} \tag{6.5}$$

where $r \in \mathbb{R}^+$ is the sphere's radius in screen space. For a perspective projection with field-of-view θ



and image height h, r is then estimated using the theorem of parallel lines (Fig. 6.8 right) as

$$r = r_{obj} \frac{h}{2 \tan\left(\frac{\theta}{2}\right) z} \quad = \quad \text{const} \cdot z^{-1}. \tag{6.6}$$

Therefore, we estimate the screen space noise to $G_{scr}\left(0, \sigma_0 / z_{ij}\right)$.

By combining $(v-u)^2 = \int_\Omega d\mathbf{x} G^2(0,\sigma) = |\Omega|\sigma^2$ and $v - u = \pi_{\lambda K} = \lambda \text{div } \mathbf{p}$ (Eq. (6.4)), an optimal λ can be estimated for a known σ using the method of alternating variables [Ber04]:

$$\lambda^{(n+1)} = \frac{\sqrt{|\Omega|}\sigma}{\int_\Omega \|\text{div } \mathbf{p}^{(n)}\|_2 \, d\mathbf{x}}. \tag{6.7}$$

In order to avoid the costly per-iteration log-reduce required to compute the norm of div $\mathbf{p}$, we assume this norm to be constant for coherent views and time steps. After combining all constants into a new global parameter λ_0, we arrive at the following update at pixel position ij:

$$\begin{aligned} \lambda_{ij}^{(0)} &= \lambda_0 \left(z_{ij} + \varepsilon\right)^{-1} \\ \lambda_{ij}^{(n+1)} &= \lambda_0 \left(z_{ij} - \lambda_{ij}^{(n)} \text{div } \mathbf{p}_{ij}^{(n)} + \varepsilon\right)^{-1}. \end{aligned} \tag{6.8}$$

Boundary conditions. To apply our filter only to those pixels for which the view ray intersected a sphere (foreground), we keep track of a binary foreground mask. To ensure proper support and boundary conditions for our filter on this oddly shaped domain Ω, we further extrude depth values by one pixel using a 3×3 mask that interpolates from valid pixels.

Implementation. The updates given by Eq. (6.3) and Eq. (6.8) can be mapped to a straightforward GPU implementation using compute shaders with one thread per pixel of the viewport. In addition to a linear depth buffer containing z and the foreground mask—which we obtain from one of our G-buffers—we store the two dual variables $\mathbf{p}$ as well as div $\mathbf{p}$ and the local λ for each pixel in four floating-point textures.

After fluid rendering in each frame, we initialize $\mathbf{p}$ and div $\mathbf{p}$ to 0 and λ to $\lambda^{(0)}$ given by Eq. (6.8) for each pixel. The depth buffer is extruded by one pixel without modifying the foreground mask. We then repeatedly perform two passes for a user-defined number of iterations: First, both $\mathbf{p}$ are updated according to Eq. (6.3), where the required derivatives are approximated using forward differences. Afterwards, div $\mathbf{p}$ is updated from backward differences of $\mathbf{p}$, and the new local lambda is calculated following Eq. (6.8).

After these iterations, we obtain the final depth value for foreground pixels by subtracting λ div $\mathbf{p}$ from the input depth value according to Eq. (6.4). Invisible non-foreground pixels are not updated. The

final depth is stored in the depth buffer, which is then used in deferred shading for normal and position calculation.

6.5.4 Transparencies

To further enhance the visual appearance of the fluid, translucent materials can be approximated with a fast screen-space approach [CLT07]. First, the shaded scene geometry is rendered into a separate background buffer (see Fig. 6.5). Then, when shading the fluid surface, we blend the contents of this buffer with the fluid color. The texture coordinates used in the background buffer lookup are given by the position of the currently shaded pixel. We distort these coordinates by offsetting them depending on the fluid's smooth surface normal to approximate refractions.

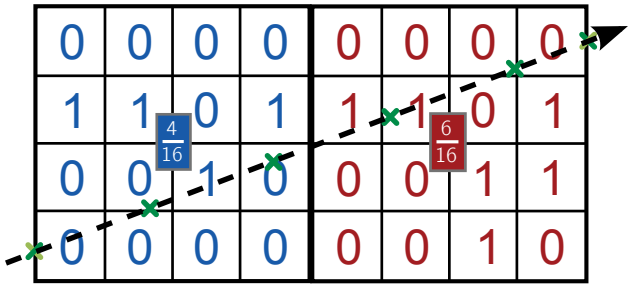
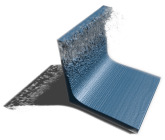


Figure 6.9: Foam density calculation: In 2D, two 4^2 bit-fields of a (4×8) sub-brick are queried fetched, and the number of set bits in either field is used as density value.

In addition, absorption is calculated by tracing each ray through the fluid body. For this, we employ fixed-step volume ray-casting through the each surface brick and extract the local particle density at every sampling point. The resulting densities are accumulated using a simple emission-absorption model. The process works by dividing every sub-brick into two 4^3 blocks, and by computing their density as the number of bits set (see Fig. 6.9). Since sub-bricks are encoded as 4 32-bit integer values, we can use a simple popcount on two integer values accessed in one texture fetch operation. The density at the sampling point is then determined by trilinear interpolation between adjacent cells. The resulting absorption value determines the amount of background color blended with the fluid surface color. Fig. 6.10 illustrates the effect.

It should be clear that, since we are using front to back ray-casting and can reconstruct accurate fluid surface normals by evaluating the particles' smoothing kernels, our approach can also simulate correct multi-bounce refractions and reflections. However, due to efficiency reasons we have not considered this option in the current work.



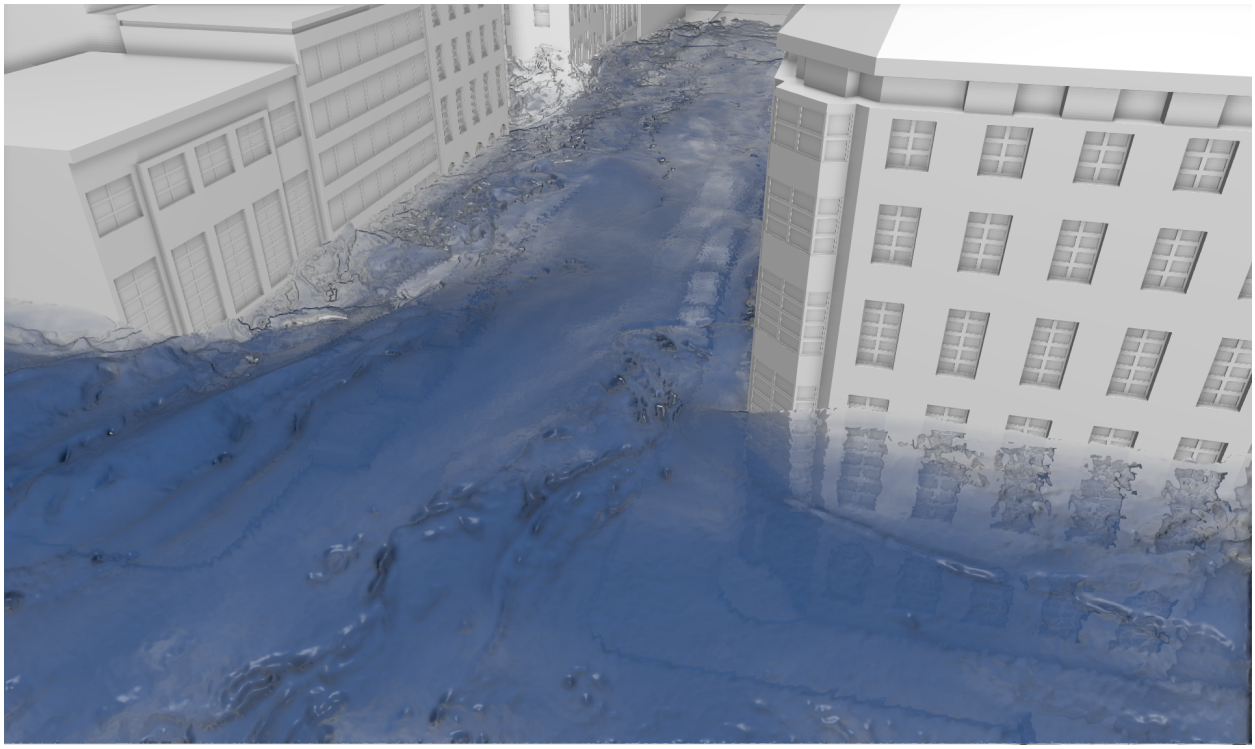


Figure 6.10: Transparency is determined by the absorption along each ray and combined with screen-space refractions.

6.5.5 Foam Classification and Accumulation

Once a requested brick has been converted and uploaded to the GPU, further processing of the quantized particles can be performed. Here, the binary voxel representation allows highly efficient operations, such as neighbor searches, which are very well suited to the GPU's design for massively parallel workloads. In the following we describe how to use this operation for classifying isolated particles. They can either be removed to expose the surface or remove noise, or they can be rendered as semi-transparent foam atop of the surface to give the fluid a more realistic appearance.

We propose to classify particles as foam depending on the number of neighboring particles in a certain surrounding. Isolated particles having few neighbors are selected as foam, whereas particle having more than a user-defined threshold of neighbors are selected as surface particles. We found that classifying particles as foam when less than 20% of the cells inside the particle radius are occupied yields visually plausible results in our test cases. Figure 6.11 shows the result of the classification (and foam rendering) for one time step of a FLIP fluid simulation.
Foam classification is performed by a second brick processor which is executed before mipmap classification in a similar fashion: For each sub-brick in a batch of 256 bricks, a single thread gathers the neighboring sub-bricks and counts, for each cell, the amount of set bits inside the selected search radius.

If a cell is classified as foam, its bit inside the source volume—which subsequently only contains potential *surface particles*—is unset. Finally, the so classified foam cells inside each sub-brick are stored in a separate *foam volume*. Conceptually similar to the transparency calculation, we divide each sub-brick into two 4^3 blocks and approximate the local density by counting the number of foam bits. Since the foam bricks are, in contrast to the surface bricks, only rendered by fixed-step ray marching, the resulting density is precalculated and stored as a single byte value for each of the two blocks. Thus the resulting foam volume requires only 16 bits for each 128 bit sub-brick. During rendering, early ray-termination is performed when the depth of the fluid surface is reached or the opacity along a ray has accumulated a value of 0.95.

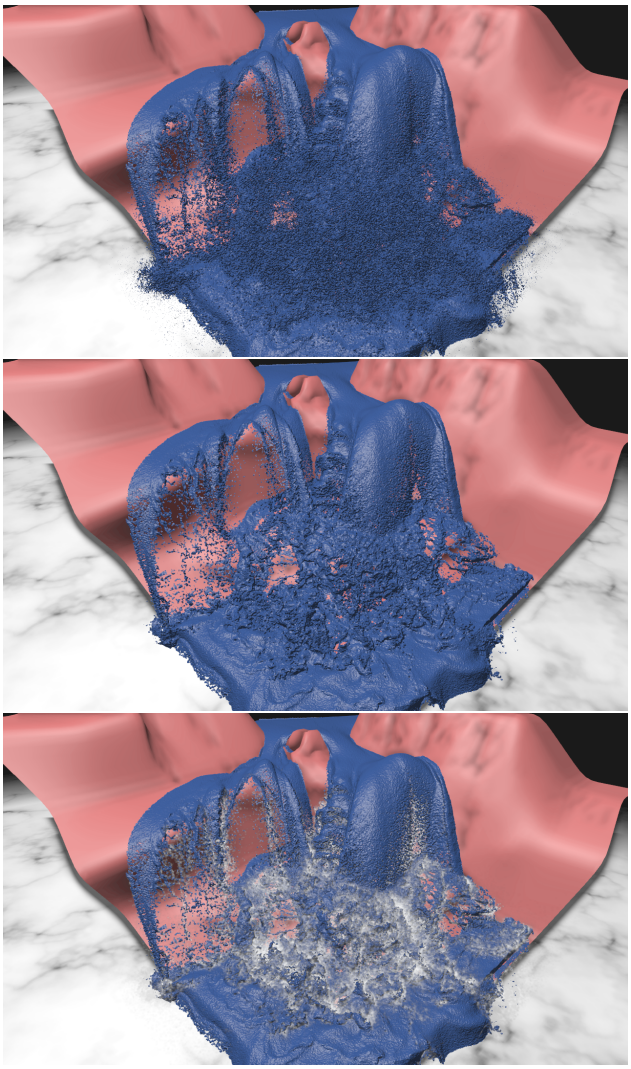


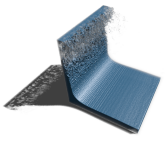
Figure 6.11: Isolated particles (top) can be determined and either removed (middle) or optionally rendered as volumetric foam (bottom).

6.5.6 Precise Rendering Mode

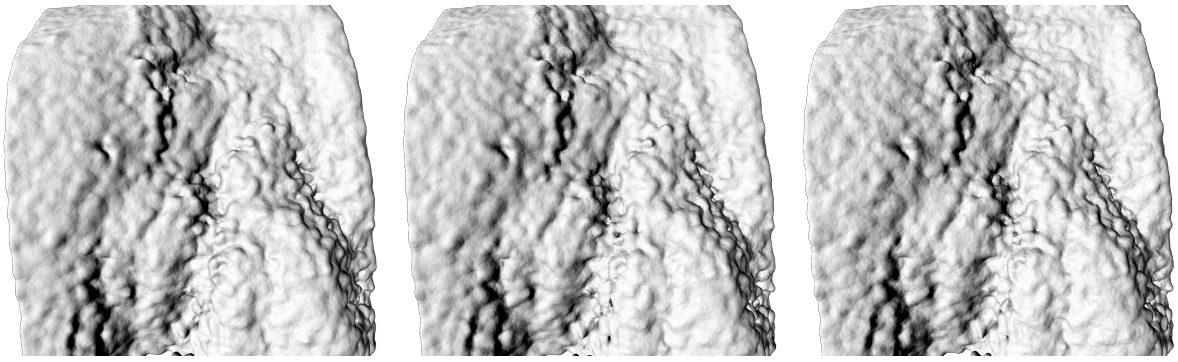
To improve the quality of the preview mode, our system can render a continuous least squares approximation to the liquid surface from the discrete set of particles. To achieve this, we follow the classical approach and compute at every cell center a weighted average of the particle densities contributing to this point. The surface is then defined as an isosurface in the generated scalar field

$$\Phi(\mathbf{x}) = \sum_j \frac{m_j}{\rho_j} W(\mathbf{x} - \mathbf{x}_j, h), \tag{6.9}$$

where W is a cubic spline kernel with support h, $\Phi(\mathbf{x})$ is the resampled data at position $\mathbf{x}$, and m_j, ρ_j, and $\mathbf{x}_j$ are the particle's mass, density, and position, respectively. The kernel's extent is set by the user. For FLIP particles the ratios m_j/ρ_j are assumed constant.



Resampling is entirely embedded into ray-casting, i.e., it is performed separately for every non-empty visible brick on the GPU. The algorithm works on batches of 128 bricks in a single kernel. Each thread is associated with a cell in the binary voxel representation and scatters the contributions of the particle—if the cell is filled—into the surrounding cells of a floating point *density brick* according to Eq. (6.9). While this approach requires atomic operations in contrast to a gather-approach, we found it provides superior performance due to the low fill rates of bricks containing the surface. Potentially filled bricks in the interior are likely to be skipped due to occlusions. Upon construction, the density bricks are rendered using isosurface ray-casting (see Fig. 6.12) and discarded afterwards.



(a) Original (b) Quantized (c) Quantized jittered

Figure 6.12: Isosurface in the resampled density field using original, quantized, and quantized jittered particle positions.

6.6 Design Decisions and Trade-offs

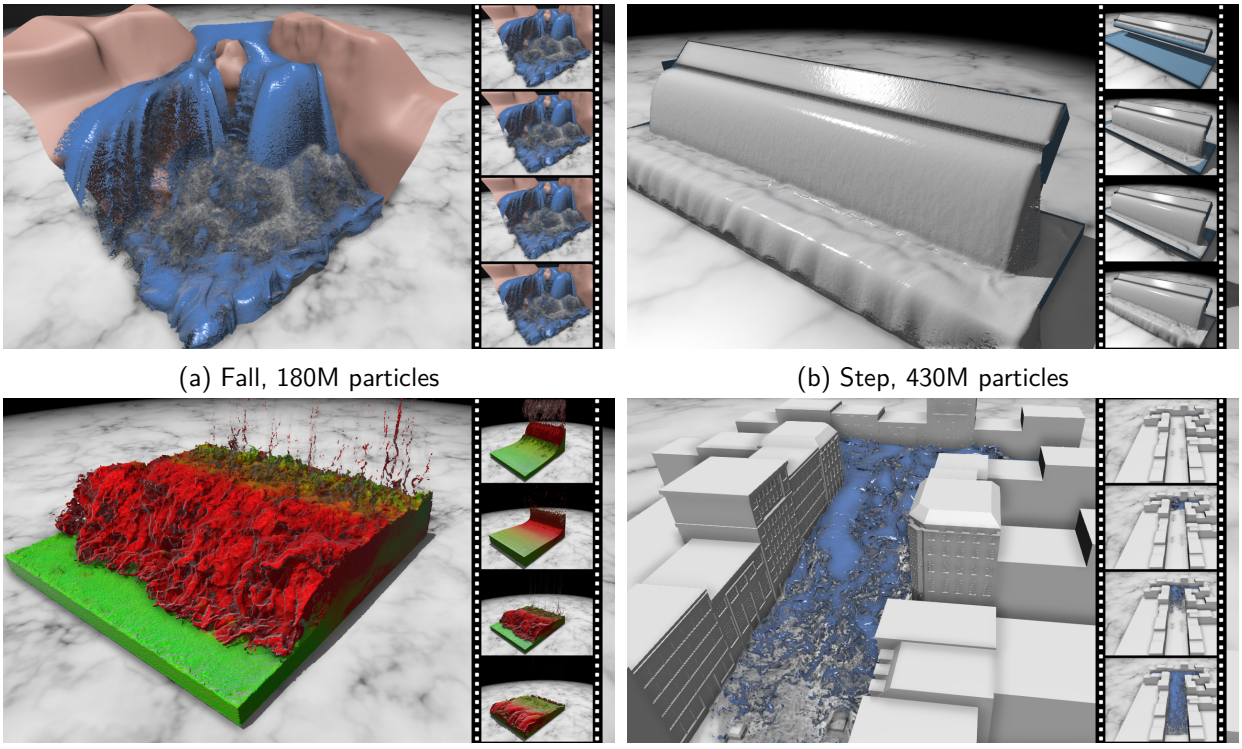
We now consider some of the decisions made in the design of our system to make it suitable for rendering very large particle data, including a discussion and comparison of alternative design choices. In particular, we want to emphasize the possible trade-offs that allow the user to choose between high quality and speed.

6.6.1 Binary Voxel Representation

Our binary voxel representation consumes one bit per cell of the adaptive spatial grid. Thus, if the fill rate of a brick is below $1/24$, in terms of memory requirements it would be more efficient to encode particle positions explicitly using 24 bits, that is, via three block-relative 8-bit coordinates. Our statistics demonstrate, however, that in practice only a small number of the non-empty bricks show this property, and that even compared to a 24-bit position encoding we achieve very good compression rates. Furthermore, in contrast to an explicit encoding, our approach enables efficient particle processing

and parallel rendering algorithms, and provides coherent memory access operations due to the specific sub-brick encoding.

For our statistics we use four FLIP fluid simulations. They are described in Table 6.1 and Fig. 6.13. The `Dam` dataset contains particle positions and velocities. The velocity magnitude is mapped to colors during brick conversion and stored as a particle attribute. Thus, we include one 16-bit pointer to the color array for every sub-brick in the memory requirements for `Dam`. All other datasets contain only positions. The compression ratio includes foam storage and mipmaps, both of which are calculated upon GPU upload and not cached on the CPU.



(a) Fall, 180M particles

(b) Step, 430M particles

(c) Dam, 180M particles

(d) City, 500M particles

Figure 6.13: Test datasets.

In particular, this shows that even the largest fluid simulation we preview can be stored in CPU RAM entirely. For browsing through the animation it is important that a huge fraction of all time steps can be stored in GPU video memory at once.

6.6.2 Bricked Representation

The bricked data representation is necessary to restrict data processing and rendering to those parts of the data which are in the view frustum and not occluded by others. Larger bricks cannot adapt so well to the areas where the fluid surface or splashes occur, and therefore encode larger empty areas. On

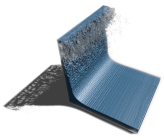


Table 6.1: Dataset statistics. 'Frm' and 'Part' give the number of simulation frames and maximum particles per frame. 'Res' is our selected grid resolution, 'Mem' gives the memory of the original sequence, 'Ratio' is the compression ratio we achieve, and 'Bpp' gives the number of bits per particles our binary voxel representation requires including mipmaps and foam storage. 'Total' values refer to the complete data set on disk, 'Visible' includes only bricks streamed to the GPU for the views in Figure 6.13.

					Total		Visible	
Name	Frm	Part	Res	Mem	Ratio	Bpp	Ratio	Bpp
Step	330	430M	4096^3	840 GB	1:28	3.66	1:62	1.63
Fall	366	180M	2048^3	700 GB	1:14	6.74	1:17	5.72
Dam	100	180M	1024^3	192 GB	1:32	2.82	1:43	2.08
City	223	500M	4096^3	588 GB	1:54	2.09	1:74	1.54

the other hand, our performance analysis in Section 6.7 indicates that larger bricks result in a better throughput in cells per second. This is confirmed in Fig. 6.14, which shows for the views in Fig. 6.13 decreasing rendering times, but increasing memory requirements with increasing brick size. According to these tests we have chosen a brick size of 64^3 as a good balance between rendering and processing times, and memory consumption.

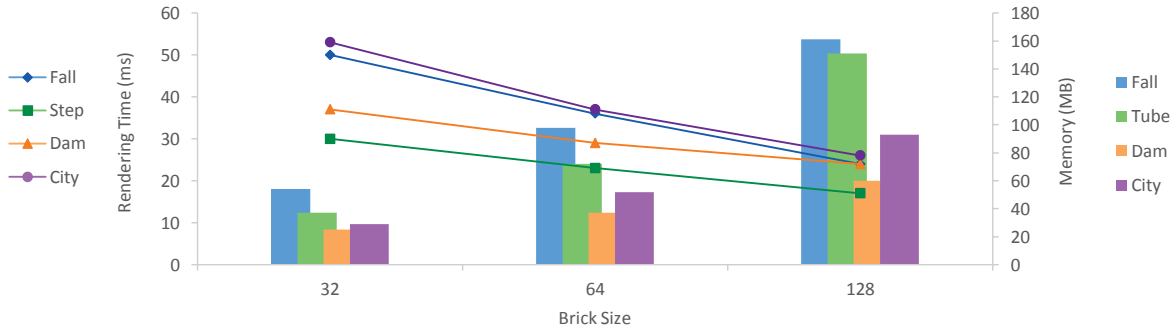


Figure 6.14: Influence of the brick size on rendering time and memory requirement.

Note that our approach does not make use of a LoD particle representation to reduce the number or rendered particles in every frame. This is possible in general, for instance, by using particle merging strategies as proposed in [FSW09], yet the particular strategies for fluid and foam particles in combination with sphere rendering need some further investigation.

6.6.3 Position Quantization

The conversion process quantizes particle positions to the cell centers of a regular 3D grid. Even though we can avoid the regular structure in the particle distribution by adding a random jitter to the

quantized positions, quantization always introduces some loss of positional accuracy. However, this does not introduce any significant artifacts due to the following reasons. Firstly, since the binary voxel representation is very compact, a high resolution discretization of the domain can be chosen to make the quantization error very small. Secondly, when rendering spheres and image-based post-smoothing, only an approximation of the fluid surface is displayed, rendering quantization errors negligible (see Fig. 6.7).

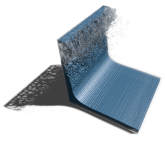
6.6.4 Volume Ray-Casting

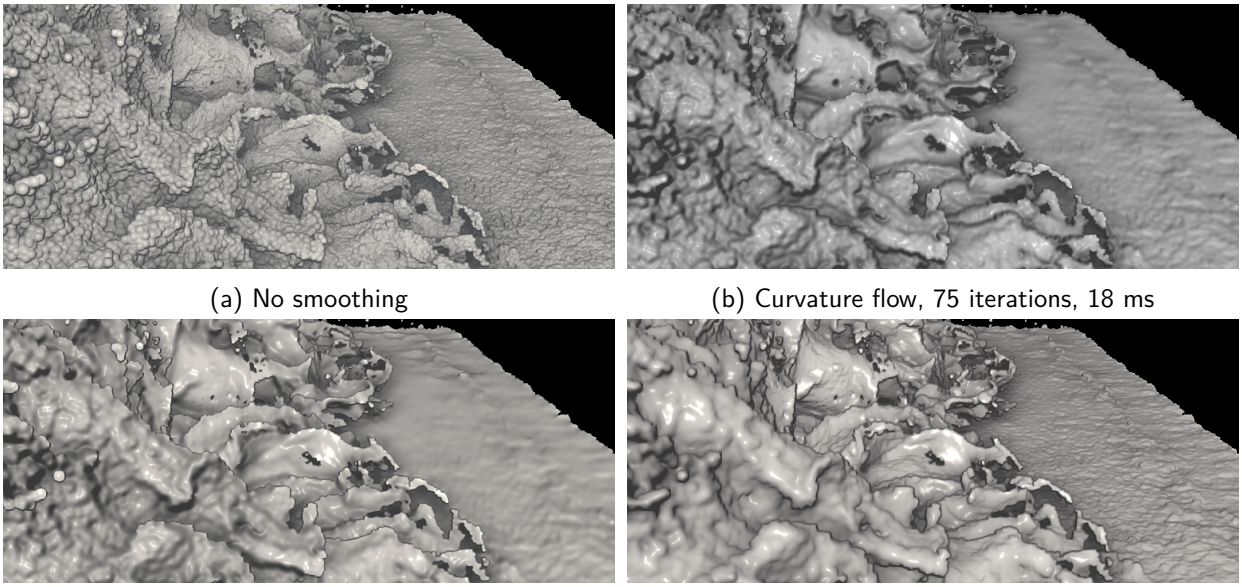
There are a number of reasons for using volume ray-casting in our previewing system: Firstly, the ray-caster can render from the compact binary voxel representation directly, without having to convert the binary data into a renderable vertex representation as required by rasterization. Secondly, compared to rasterization-based particle rendering, much better rendering rates can be achieved for reasons already stated in section 1.3. When rasterizing particles on recent GPUs as screen-space aligned quads consisting of two triangles, about 250M particles can be rendered in roughly 250 ms, not including the time required to realize the spheres' perspectively correct depth footprints in a pixel shader. Using the same particle size, the ray-caster renders 3D spheres about a factor of 10 faster. In addition, the ray-caster can exploit occlusion culling at the pixel level, which is problematic when using rasterization. Occlusion queries in rasterization can only be effective at a rather coarse granularity, i.e., if parts consisting of many polygons can be culled at once. Finally, due to the front-to-back traversal order, the ray-caster can render semi-transparent particles like foam without any modifications. The rasterizer, in contrast, either needs to sort the particles or the generated fragments per frame to ensure correct front-to-back blending.

6.6.5 Image-based Smoothing

We compared our proposed smoothing filter to two popular screen-space filters: bilateral filtering [PM90] and curvature flow [vdLGS09]. Figure 6.15 shows the same data set and view processed with the three different filters to demonstrate the differences.

Our filter has the distinct advantage of having a fixed and very compact stencil size. This stencil size is independent of the desired amount of smoothing. Thus, the filter has very light memory bandwidth requirements and outperforms even simple (i.e., non-depth-adaptive) bilateral filtering. While the bilateral filter can also be extended to locally adapt the radius to a desired world-space filter size, this imposes severe performance constraints as the filter is not separable. Furthermore, iterative application of smaller bilateral filters as well as separable bilateral filters [PV05] are approximations that may give suboptimal and undesirable results, such as overly flat surfaces and overly sharp depth discontinuities.





(a) No smoothing

(b) Curvature flow, 75 iterations, 18 ms

(c) Bilateral, 15 ms

(d) Our method, 75 iterations, 13 ms

Figure 6.15: Screen-space smoothing comparison

Compared to curvature flow, our filter is better at preserving sharp features and depth discontinuities, and the amount of smoothing does not come from adding iterations, but from altering the smoothing parameter λ_0.

6.7 Performance

Here we evaluate the performance of all components of our system and provide accumulated timings for the most time-consuming operations. All timings were performed on a dual quadcore Intel Xeon X5560 with 48 GB main memory, and an NVIDIA GTX Titan with 6 GB video memory. Data was accessed from a solid state drive delivering up to 500 MB/s read peak-performance. We used a viewport resolution of 1280 $\times$ 720 pixels for all rendering passes. All times given in this section are in milliseconds. The particle diameter was set to 3 cells, with a maximum jitter of 0.5 cells.

Table 6.2: Per brick processing statistics for different brick sizes. 'Conv', 'Foam', and 'Mip' are the times for CPU conversion, foam classification, and mipmap generation, respectively.

Size	Conv	Foam	Mip
32	.06 $\pm$ .02	.02 $\pm$ .01	.01
64	.22 $\pm$ .06	.15 $\pm$ .08	.04
128	.91 $\pm$ .37	1.06 $\pm$ .41	.27

Table 6.2 gives average times for the operations performed during previewing. The times for brick sizes 32^3, 64^3, and 128^3 are compared. The statistics were performed using the bricks in all datasets and computing the times required for processing each of them. We give the average times and the interval

in which the times vary. Only mipmap generation does not depend on the brick's fill rate and is constant for a fixed size. The times of all other operations vary depending on a brick's fill rate.

Another exemplary experience shows the impact of the particle influence radius on the rendering and processing times. Here we used bricks of size 64^3, and we analyzed sphere rendering and foam classification as the representative operations. Figure 6.16 shows exponentially increasing times due to the increasing number of neighbor particles to be considered in both operations.

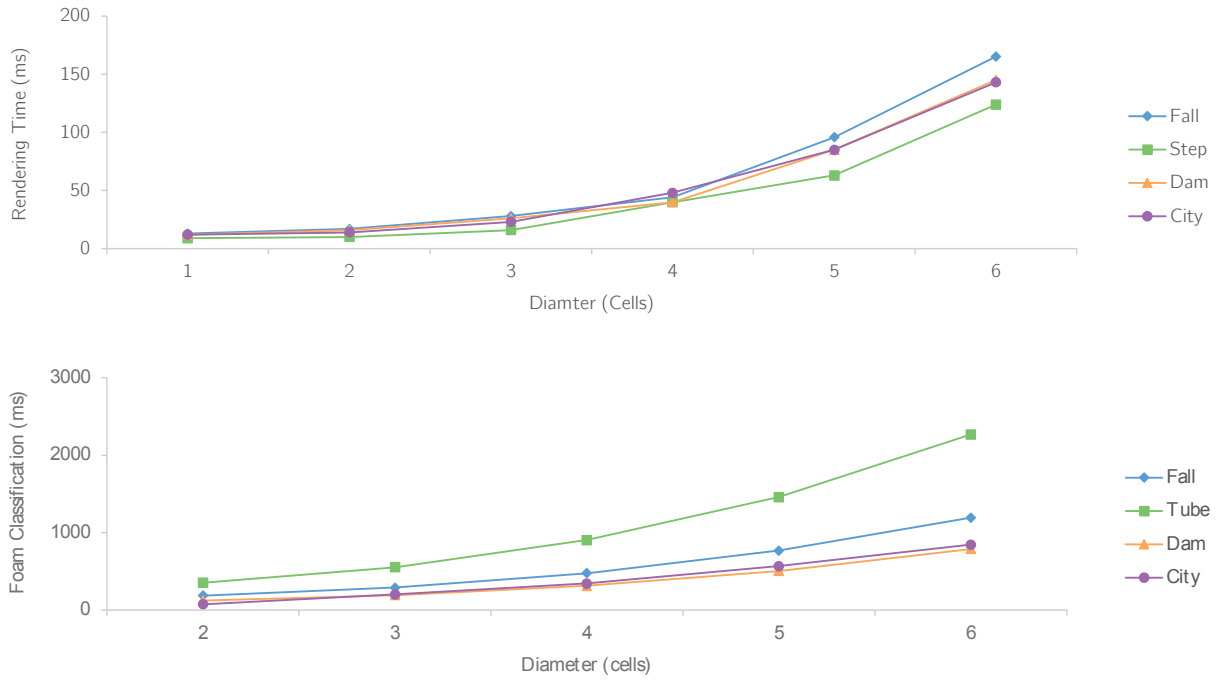


Figure 6.16: Influence of particle radius on rendering (top) and foam classification (bottom) performance.

To analyze the concrete previewing times for our test datasets, Table 6.3 summarizes statistics for average views where the whole dataset is visible on screen. Experiments were only performed once for the given viewport size, because all rendering times scale linearly in the number of pixels. We assume that all converted bricks, foam bricks and brick mipmaps are residing in GPU cache.

The first observation is that only a small fraction of all bricks need to be accessed to generate the views. This is because occluded bricks do not have to be touched. Another observation is that the rendering of volumetric foam has a significant effect on rendering performance, as seen for the turbulent datasets exhibiting violent splashes and foam. In this case the view-rays have to be traversed through the volume for a much longer distance compared to the rendering of an opaque surface. Post-smoothing using 75 iterations on the GPU requires about half the time sphere rendering does.

System Behavior. The responsiveness of our system was analyzed in a previewing session, where a single time step is explored by the user interactively, performing a number of view changes and zooms.

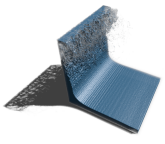


Table 6.3: Memory and performance statistics for the views in Fig. 6.13. Listed are consumed GPU memory ('Mem') for surface and foam, the total number of bricks, surface- and foam particles, the times for sphere ('Sphere') and foam ('Foam') rendering, absorption estimation for transparencies ('Trans'), and post-smoothing ('Smooth').

Data	Mem	#Bricks	#SurfP.	#FoamP.	Sphere	Foam	Trans	Smooth
Fall	210 MB	3175	46.6M	4.9M	32 ms	12 ms	14 ms	13 ms
Step	157 MB	2366	254.2M	27.8M	18 ms	3 ms	12 ms	13 ms
Dam	83 MB	1162	95.1M	6.8M	21 ms	10 ms	13 ms	16 ms
City	127 MB	1919	213.3M	45.6M	19 ms	2 ms	12 ms	13 ms

For each frame we measured the *frame time* until all required bricks are converted, processed, and rendered. Furthermore, we counted the number of requested bricks per frame. Disk I/O is *not* included in the timings, and the system started from "cold caches".

The performance graphs for the four data sets in Fig. 6.17 show the measured frame times, the required GPU memory, and the number of requested bricks. A peak in the frame time is observed at the beginning. This peak results from the particular strategy we use on the GPU to request bricks not yet residing in video memory. After each frame, the request buffer is read back to the CPU and requested bricks are converted and uploaded to the GPU. Since a ray terminates as soon as a brick not yet available on the GPU should be rendered, this rendering can only be performed in the next frame. Thus, at the beginning, when no bricks are cached in GPU memory, our strategy results in a delay of several frames. The number of frames is equal to the maximum number of bricks that are requested by a ray. Nonetheless, one can see that the first frame requires only roughly 700 ms. Note that this time also gives the rate by which one can scrub through an entire fluid sequence, where in every frame the GPU cache is clear and bricks are newly requested.

The graphs showing for each frame the number of requested bricks and the memory requirements reveal the advantages of embedding the compact particle representation into front-to-back ray-casting. Since only non-empty and non-occluded bricks are rendered, a rather small sub-set of all bricks needs to be uploaded and processed on the GPU. Furthermore, because the ray-caster can render directly from the binary particle representation, avoiding any data conversion on the GPU, the memory consumption on the GPU is kept very small.

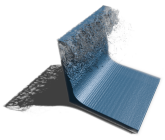
6.8 Conclusion

We have presented a rendering system for very large particle-based fluid simulations. Due to the compact particle representation and the intertwining of particle processing and rendering we could demonstrate interactive previews of entire fluid sequences. Our approach is able to display a smooth fluid surface,

Figure 6.17: System behavior during data exploration of a single time step with different views and zooms. Note that the graph for uploaded bricks in the left column is capped at 50 for better visualization, with the number of bricks in the first frame explicitly labeled.

in particular due to the use of an improved image post-smoothing method. We could demonstrate interactive previews of simulation data comprising up to 500 millions of particles per time step.

In the future we plan to extend our system towards a high-quality rendering system including advanced fluid and foam illumination effects. This is possible in principle, because our implemented neighbor search operations can be used to evaluate particle kernels and compute accurate density volumes and surfaces therein. To avoid performing huge numbers of operations in the fluid body, we will further investigate the construction of effective space-leaping techniques on the fly during rendering.



7

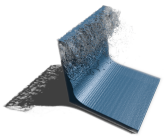
Surface Rendering with the RLE Sample Buffer

Figure 7.1: A 1 billion triangle model is rendered on a 1920×1080 viewport using rasterization and ray-casting simultaneously. Ray-casting works on a sample-based surface LoD representation at an effective maximum sampling resolution of $8K^2 \times 32K$. Red and green surface areas indicate parts of the model that are rendered via rasterization and ray-casting, respectively. On a GTX 680 graphics card the hybrid approach always renders the model in less than 30 ms ($>$ 33 fps) at a screen space error below pixel size.

7.1 Introduction

The previous chapters presented solutions for various particle-based data sets. In both cases, a representation based on Cartesian grids offered an intuitive alternative to rasterization-based approaches as the underlying data was of volumetric nature, given at discrete irregular points of support inside the domain. This chapter addresses large datasets already present in a triangle-based surface representation.

As already stated in chapter 1, pure rasterization has severe limitations for large scenes, whereas sample-



based representations are capable of elegantly solving a variety of problems by straightforward LoD generation, guaranteed rendering error bounds and an efficient integration of on-demand data streaming in combination with occlusion culling. On the downside, volumetric resampling as e.g. performed by the GigaVoxels pipeline [CNLE09] may significantly increase the memory requirements. More compact representations have since then been sought after by building sparse representations up to the level of single voxels [LK10, Áfr12]. While this reduces the amount of required memory for the voxel data itself, it also introduces additional overhead to store the huge number of pointers inside the sparse octree and diminishes the ray-casting performance due to potentially deep trees and decreased memory locality.

Rendering of binary voxelizations is highly reliant on maintaining a voxel-to-pixel ratio of as close as possible to 1:1 to correctly resample the surface. While undersampling will lead to aliasing artifacts similar to triangle-based rendering, oversampling poses a much more severe problem: The storage in a sparse data structure prohibits utilization of hardware-supported interpolation, and even low-quality manual trilinear interpolation is too costly to be performed at every sampling point along a view ray. This results in a blocky surface appearance in the case of oversampling as DDA ray-marching intersects the ray with the surface of the cubical voxels (Fig. 7.2). For voxel attributes such as colors and normals, this problem can be concealed by employing screen-space filtering along the surface, but the blocky appearance of silhouettes will still greatly diminish the visual quality of the image. Laine et al. [LK10] solve this by storing additional sub-voxel planes (*contours*) which more closely resemble the surface but come at the cost of additional storage memory.

Figure 7.2: Oversampling of a binary voxelization. Visualized are the normals stored at each voxel.

Regardless of the inherent advantages of voxel-based representations, the highly optimized rasterization-based pipeline comes with its own virtues: large triangles benefit from the massive throughput of current GPUs, and geometric oversampling in close-up views does not pose a problem. A hybrid combination of both approaches, thus, lends itself to achieve optimal performance and quality for every possible combination of triangle sizes, model complexities and viewport resolutions. This approach has already been taken by Dick et al.in terrain rendering [DKW10], where a performance increase was achieved by combining height field ray-casting and rasterization even despite the presence of a geometry LoD.

7.1.1 Application Goals

In this chapter we propose a hybrid GPU pipeline for computing eye-ray intersections with arbitrary models. It performs GPU rasterization and ray-casting simultaneously, deciding at run-time which technique to use for each part of the model.

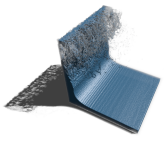
To efficiently perform ray-casting, we introduce a novel compact sample-based surface representation, the *Run-Length Encoded Sample Buffer (RSB)*. It can be created efficiently at multiple resolutions and provides an effective mechanism to reduce the number of evaluated surface samples. Compared to a triangle mesh, regular grids or sparse voxel octrees, the proposed representation has a significantly lower memory consumption. Thus, it is less sensitive to bus bandwidth limitations. The representation is built on a regular 2D sampling structure, on which parallel ray traversal can be performed efficiently in front-to-back order.

We demonstrate that the proposed graphics pipeline can be implemented efficiently on recent GPUs, and that significant performance gains can be achieved for high-resolution polygon models. To the best of our knowledge, for the first time we can show that a rendering pipeline based on ray-casting can be faster than rasterization for eye-ray intersections of arbitrary polygon models. Since the RSB can be constructed from any available surface representation, it can also be used for rendering isosurfaces in large volume data sets. To enable interactive selection of different isosurfaces, we have implemented the construction of the sample-based representation on the GPU. We show that for data sets as large as 4096^3, even construction of all data displayed in a typical view requires less than 200 ms from scratch. Compared to direct volume rendering, the memory requirement at run-time is reduced of a factor of up to 10.

7.2 System Components

For static triangle data, which the following elaborations will focus on, our framework (see chapter 4 for a general overview) is configured as follows:

- All data generated from a preprocess is accessed by a standard streaming supplier as detailed in section 4.4.
- Two data caches contain RSB data, and two additional data caches contain triangle index and vertex data.
- A brick traverser performs ray-casting of the RSB structure
- Two copy brick processors distribute the uploaded data into the required buffers without any further processing.



- The standard rasterization pipeline accesses the vertex and index data and is used in conjunction with the RSB ray-caster to generate the final image.

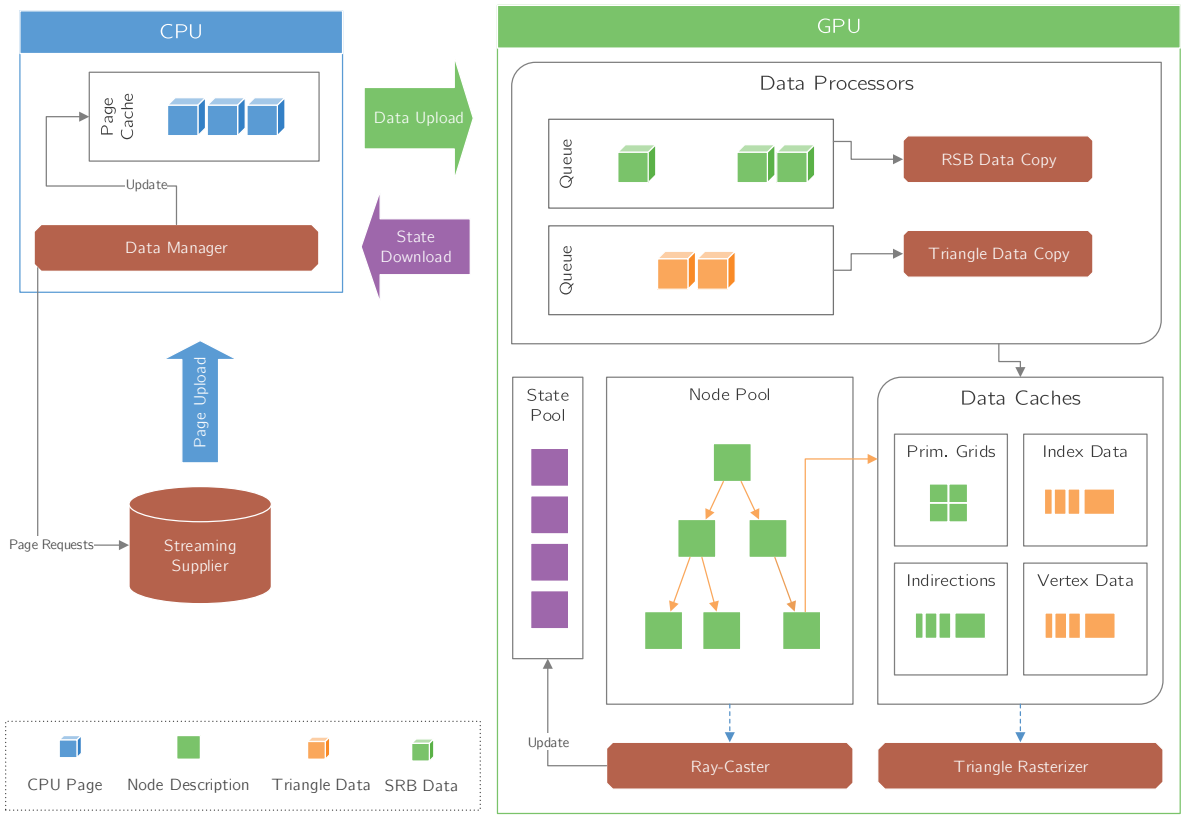


Figure 7.3: Customized components of our rendering framework: A streaming supplier loads pages of preprocessed RSB- and triangle-data, which are distributed into the data caches by two copy processors. A specialized brick traverser works alongside a standard rasterizer to generate the final output image.

Figure 7.3 depicts our framework configuration along with the mentioned changes from the standard configuration. In the following, we will elaborate in detail on the RSB data structure and its rendering, and we present the interaction of ray-casting and rasterization to chose the fastest possible rendering algorithm for each frame.

7.3 Data Representation

The RSB is influenced by the Layered Depth Cube (LDC) introduced by Lischinski and Rappoport [LR98], which samples a model from three mutually orthogonal directions onto regular grids. Building on this concept, Bürger et al. [BHKW07, BKW10] proposed GPU methods for ray-tracing secondary effects and for surface painting, but both works were not concerned with high resolution data sets and required

considerable amounts of GPU memory. Similar to our concept, Novak and Dachsbacher [ND12] proposed splitting the model into parts which can be represented as height fields, so that GPU height field ray-casting can be employed.
For high-resolution polygon models, however, finding the local piecewise height field parameterizations requires an extensive preprocess. Furthermore, it is not guaranteed that any such height field at a reasonable size exists for a desired maximum geometrical error. The effectiveness of the approach strongly depends on the surface geometry—for those reasons, their work is focused on secondary effects where local geometrical errors have a less noticeable impact.

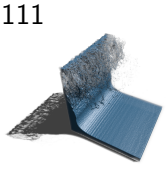
Our method requires a two-step preprocessing of a triangle mesh which we will describe in this section. Similar to Novak and Dachsbacher, we start with a spatial subdivision followed by a resampling of each surface part into an RSB. However, we use a regular space partitioning which can be constructed efficiently. In this way, we also support non-static data such as isosurfaces in scalar fields, where rebuilding an adaptive acceleration structure would be too costly during runtime. In general, the regular subdivision leads to surface parts which cannot be represented by a single height field any more. We will show how our structure resolves this problem and still allows for very efficient ray-casting. Furthermore, the RSB allows the creation of several levels of detail with guaranteed geometric error to further speed up the rendering for larger viewing distances.

7.3.1 Model Partition

Given a triangle mesh, its bounding box is partitioned into a set of bricks in a sparse octree. For each brick containing a part of the surface, a 2D sampling grid as described in the next section is constructed. The resolution of these grids is user-defined, whereas the resulting level of the tree depends on the size of the finest geometric details contained, i.e. the size of the smallest triangle inside the leaf nodes. In our current implementation, the size of a sampling cell in these grids is equal to half of this smallest triangle size. For instance, for the David model in Fig. 7.1 this corresponds to an effective finest resolution of $8K^2 \times 32K$ samples with nearly the same level generated for all surface parts due to the relatively uniform triangle size. The rationale behind this is that a sampling grid is rendered if its cell size is approximately the pixel size. Thus, when triangles are larger than this size, they would cover many pixels and rasterization would be preferred. Figure 7.4 shows a partitioning that has been generated using these rules.

7.3.2 Sample-based Data Structure

Each brick stores the triangles contained in it, clipped at the brick boundaries. From those triangle lists an RSB is built in the second step of the preprocess for each brick. In general, the data structure can be built from any renderable surface representation as well as from existing binary voxelizations of any kind. As the concept is derived from the orthogonal fragment buffer (OFB) proposed in [BKW10], we



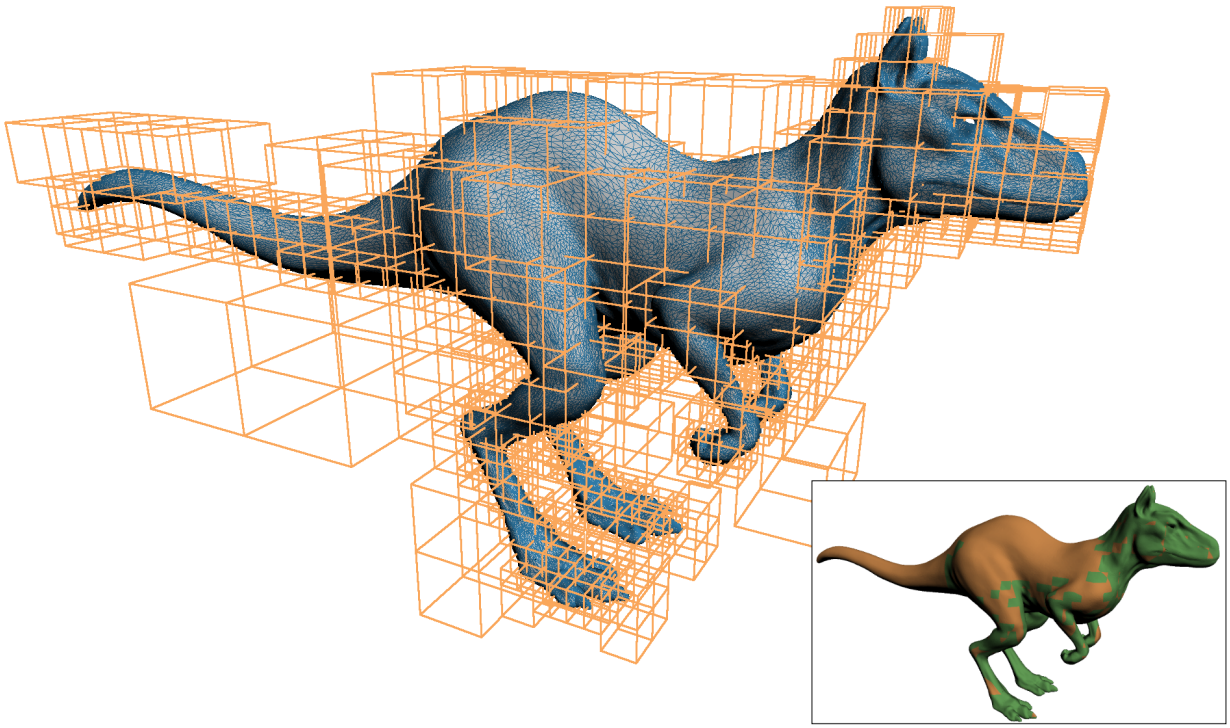


Figure 7.4: Subdivision of a triangle model into bricks generated by the preprocess using our refinement rules. The finest octree level is determined by the smallest triangle in each leaf. The inset shows the decisions of the render oracle for each surface part (rasterization: red, ray-casting: green).

will describe the conceptual creation of the RSB based on this structure and detail on the efficient GPU implementation later.

An OFB resamples a surface along three orthogonal directions and generates for each direction a set of depth layers, each stored in a 2D sampling grid (see Fig. 7.5). It is constructed by depth-peeling the surface from three orthogonal directions at a fixed resolution. Each depth layer stores the distance of the surface from the respective sampling grid, which we will refer to as the *sample value*. The depth values are quantized to the grid resolution to avoid holes in the surface representation. In addition, surface attributes such as normals and colors are rendered directly into each layer.

This original OFB format, however, has severe limitations for large polygon meshes: First, for parts of the model with high depth complexity but low fill rate in each depth stack, it introduces a substantial memory overhead. Second, ray-casting this structure requires to search all depth layers to find the first sample that is hit by a view ray. For high depth complexities and high resolution sampling grids, this introduces a severe performance bottleneck.

To address these shortcomings, we only store a single 2D sampling grid of the same resolution, the *primary grid*, for each brick. The sampling direction closest to the average normal of all triangles in a brick is selected as primary direction. All sample values and attributes of all OFB layers are stored at

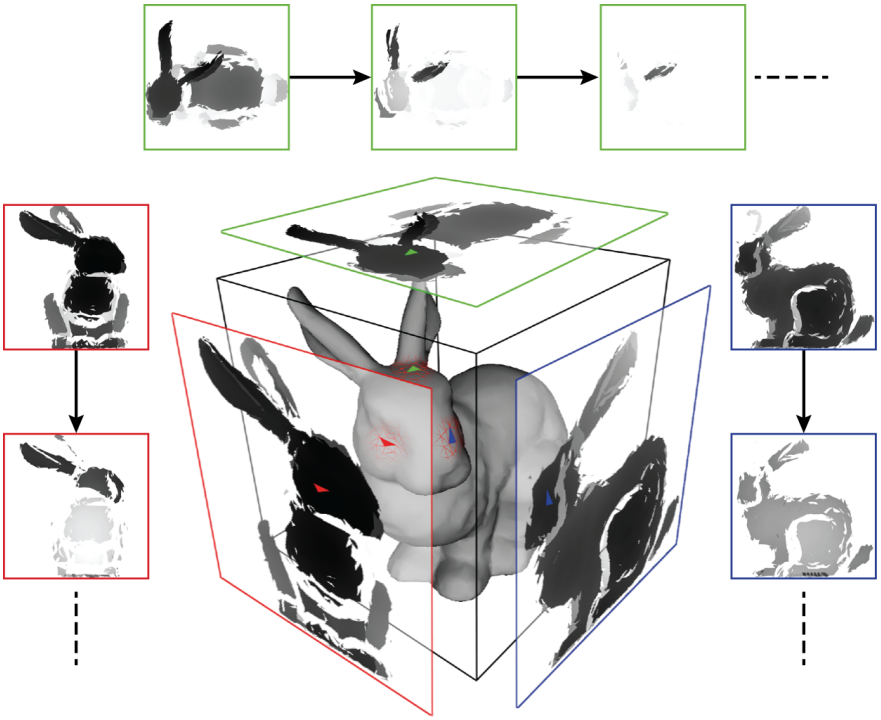


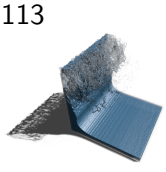
Figure 7.5: An OFB is created by depth-peeling a mesh from three orthogonal directions; the distance of the surface from the respective sampling grid are stored in several layers of 2D sampling grids.

the respective position in the primary grid; conceptually, the OFB layers are treated as a 3D grid for this step.

For each OFB direction, multiple layers can be present. This requires the storage of a variable number of values and attributes for each sampling position in the primary grid. For this reason we employ a two-level data structure consisting of the 2D primary grid and a 1D *indirection buffer*. If a grid position contains only a single sample value, it is stored directly in the primary grid along with all its attributes, much alike a traditional height field. In any other case, all sample values for a grid position are written to the indirection buffer in ascending order, followed by the attributes of all samples. A pointer to the first written value as well as the number of values is stored at the respective position of the primary grid instead (see Fig. 7.6).

To further reduce the storage overhead, a run-length encoding of sample values is performed during construction: For each span of successive values to be written into the indirection buffer, only the first value is stored along with the total length of the span.

For a sufficiently high resolution of the initial 3D space partitioning, the 2D sampling structures as described will only contain very few samples in typical cases as will be shown in section 7.5. Even if this is not the case, we will demonstrate that the compact span representations can be tested very efficiently to find possible intersections between a ray and the surface.



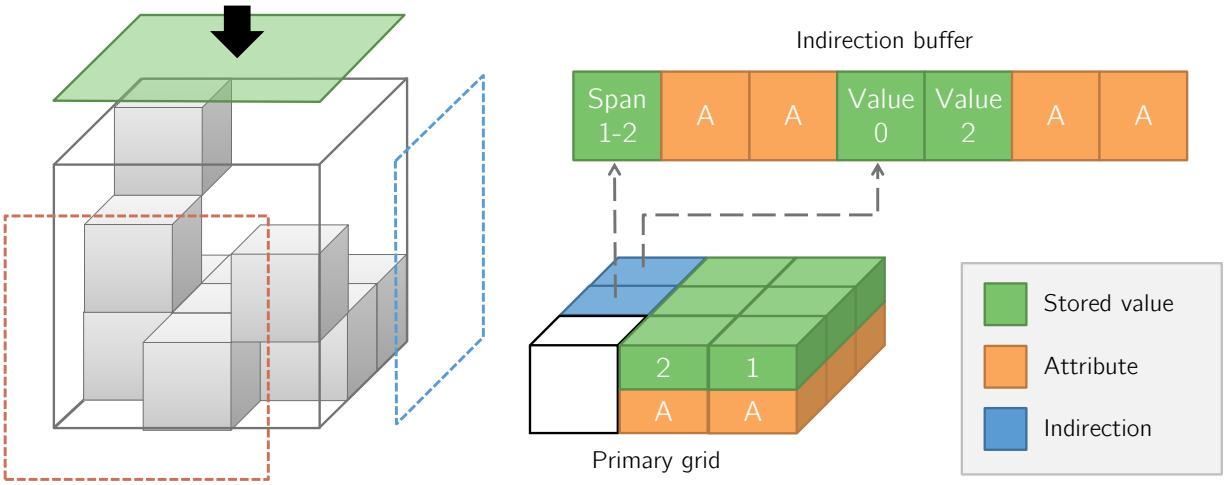


Figure 7.6: Conceptual construction of an RSB: The samples of a binary voxelization (left) are stored into a single 2D structure, the primary grid (right). In this example, the top-down view is selected as primary direction. If only a single sample exists for a grid position, its distance from the primary grid is stored directly along with its attributes. Otherwise, a pointer into the indirection buffer containing the sample values is stored. Successive voxels are stored as a single compact RLE span.

7.3.3 Preprocess Implementation

Generation of an RSB can be performed from any renderable representation or existing voxelization. Previous works [Pan11, SS10] demonstrated fast voxelization and octree creation of triangle meshes, but did not consider problems arising from limited GPU- or even CPU memory. To address these, we employ a 4-phase construction process targeted at flexibility during our offline preprocessing. We will detail on this process exemplary for triangle data in the following (see Fig. 7.7 for an illustration).

Partitioning:
Preprocessing starts by generating the octree in a top-down manner from the input data and assorting triangles to the leaf nodes, similar to the construction process employed for big SPH simulations in section 5.4. We perform two sequential runs over the input data on the CPU: First, the corresponding leaf node is determined based on a user-defined resolution of the RSB structure—typically 64^3 samples. Starting from the root, each triangle descends the octree until the level dictated by its size is reached, creating all nodes covered by it along the way. For reasons of LoD creation, we create either all or none of the children of each brick. Secondly, triangles are stored into the respective leaf nodes, duplicated if necessary and clipped to the cell boundaries. The average normal direction and thus preferred primary grid plane is determined in this process from the area-weighted sum of all triangle normals. Finally, we re-index the triangles inside each brick.

If a resulting leaf brick contains only triangles which are at least two times as large as the cell size, though the sampling structure is created to facilitate the construction of a LoD hierarchy, it is deleted upon finishing this process. We will call such bricks *unresolved*. An additional criterion takes into

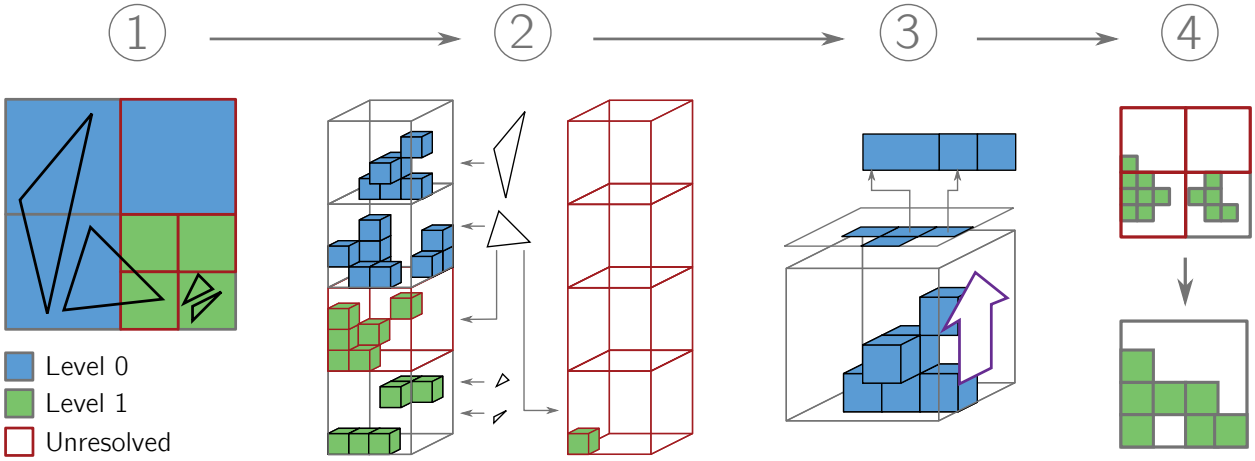


Figure 7.7: Preprocess outline: The scene is partitioned into an adaptive octree (1). Each brick is voxelized in parallel (2), and RSBs are created (3). Finally, higher levels of the hierarchy are constructed by merging child RSBs (4).

account the fill rate of a sampling structure. For a constructed RSB, the fill rate measures the fraction of effectively used entries in this representation. If this fraction is too low, which indicates that a huge number of entries are wasted, the respective grid is also classified as unresolved.

Voxelization:

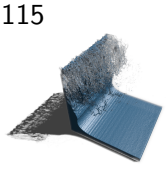
From the triangles of each so-created leaf node, a binary voxelization is created. Our pipeline employs standard rasterization with multisampling from three orthogonal directions to a framebuffer of brick resolution. With depth testing disabled, each fragment is quantized to a voxel position in the pixel shader, and voxel attributes are accumulated in global memory and stored in 3D textures of brick resolution along with a triangle counter per voxel.

To perform this process in a scalable way it is interleaved with the next pipeline step and executed on batches of fixed size. The adaptive subdivision generated during partitioning results in a soft upper limit of triangles per brick and, thus, the required data can be easily held in memory if the batch size is chosen reasonably.

RSB Creation:

After a binary voxelization of the bricks is obtained, an RSB is created for each brick separately. We perform two sweeps over the voxel grid with one thread associated to one texel (a *column*) in the resulting primary grid. In the first sweep, the number of voxel spans along each column is determined and stored in the primary grid. In the second sweep, the averaged attributes along with the span offsets and sizes are either written to the indirection buffer or directly into the primary grid if only a single sample was found. The according indirection buffer write offsets are determined by a parallel prefix sum over the primary grid in between the two sweeps. Finally, all RSB data of the batch is downloaded to the CPU and the intermediate full voxel grids are deleted from GPU memory.

Hierarchy Generation:



Once all RSBs have been created, the multiresolution hierarchy is generated bottom-up by merging 8^3 child bricks. To determine the primary grid plane, the average normal of the child nodes is calculated, and the result is stored along with the octree node to be used in the next level. To improve on the resulting average, we store the total area of all contained triangles with each node. Parent RSBs are then created as in the previous phase, but directly from the child RSBs instead of a temporary voxel brick. The RSB data of previously created unresolved bricks is deleted in the process, and the triangle- and sample data of all remaining bricks is stored on disk along with the sparse octree structure to be accessed by our streaming supplier during runtime.

7.4 Hybrid Rendering

In every frame, the octree that was computed in the preprocess is first traversed on the CPU, and the bricks which have to be *rasterized* are determined. Here, a brick is only considered if it satisfies a user-defined screen space error, usually one pixel or below, or if the finest octree level is reached. Unresolved bricks are always rasterized. Bricks for which both the triangles and an RSB are stored are rendered via rasterization, if a) the RSB would be oversampled or b) a *render oracle* determines that rasterization will be faster than ray-casting. The oracle is evaluated on the CPU for every brick that is selected for rendering and determines the most efficient rendering technique for that brick. We use a modified version of the oracle proposed in [DKW10] for hybrid terrain rendering. The cost for rasterizing a single brick are calculated as

$$t_{\text{rast}} \approx \mathcal{O}(T) \approx c_1 \cdot T + c_2. \tag{7.1}$$

Here, T denotes the number of triangles contained in the brick. c_1 and c_2 are hardware-specific constants based on the expected triangle throughput of the GPU and the overhead that comes from issuing a single draw call.

The performance of the ray-caster, in turn, is given by

$$t_{\text{ray}} \approx \mathcal{O}(R \cdot S(Eye)) \approx c_3 \cdot R \cdot S(Eye). \tag{7.2}$$

R denotes the number of rays spawned to ray-cast the brick, which we estimate by clipping the back faces of the bounding box at the view frustum and computing the screen-space coverage in pixels. S calculates the average length of a ray projected into the bricks' primary grid. This value is then used to interpolate between two hardware-specific constants s_1 and s_2 approximating the cost of a ray orthogonal to the primary grid (s_1) and parallel to it (s_2).

The render oracle evaluates both cost functions and selects the cheapest rendering method. An example of the resulting view-dependency of the oracle is shown in Fig. 7.8.

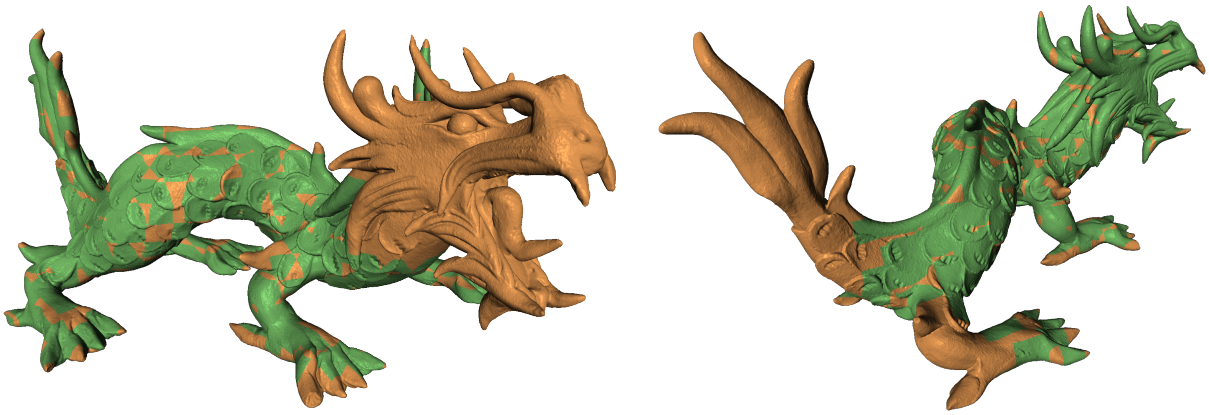
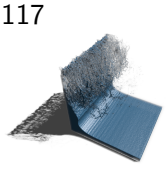


Figure 7.8: View dependency of our render oracle. Depending on the number of triangles and the brick's extent on screen, the oracle determines which parts of the surface should be rasterized (red) and ray-cast (green).

The bricks selected for rasterization are rendered in front to back order. To reduce the amount of uploaded data and the workload for the rasterizer, hierarchical occlusion culling is performed [BWPP04]. All bricks which are not rendered via rasterization are flagged for ray-casting in the node pool, and this information is propagated up the tree. The octree traverser will then only consider these bricks during rendering and only descend into subtrees that contain at least one brick flagged for ray-casting. The flags of all bricks inside the current view frustum are updated in each frame by evaluating the oracle based on the current size of the brick's projection onto the screen and the contained triangle data. As in previous framework configurations, RSB data is uploaded upon GPU request to utilize per-ray occlusion culling. The ray-caster also considers the depth buffer generated from the rasterization pass to perform early ray termination.

7.4.1 RSB Ray-Casting

Ray-casting a brick is performed using an extension of standard height field ray-casting: For each brick a ray is projected into the 2D sampling plane and DDA-like ray marching is performed in front-to-back order to find the grid cells hit by the ray. For every cell a flag-bit indicates whether a single sample or a pointer into the indirection buffer is stored. In the first case, a single height value needs to be tested for intersection; in the later case, the ray is tested for intersections with the sample spans stored at each cell, i.e., whether it intersects one of the 3D cells associated with the stored samples. Intersections with spans consisting of multiple samples are handled in a single test. Once an intersection is found, the surface attributes can be read from the data buffer using the index of the sample that was intersected.



If no intersection is found, ray marching continues with the next cell until the end of the sampling grid is reached.

To further accelerate the ray traversal process, we employ 2D maximum/minimum mipmaps [OKL06, TIS08]. A mipmap representation can effectively reduce the number of ray-casting steps until the first intersection, and in our particular scenario it can be used to discard those cells where an intersection cannot occur. For each brick, such a min/max hierarchy is computed in the preprocess. These hierarchies are then used in the traversal process to skip regions that are completely below or above the ray.

7.4.2 Secondary Effects

The integration of a ray-caster into the rendering pipeline provides an easy integration of various secondary effects. As in our previous applications, GPU-driven traversal of the tree in a secondary effect rendering pass enables the framework to request bricks that were previously culled or rendered by the use of rasterization. In addition to the hard shadows included in most of the images in this chapter, Fig. 7.9 demonstrates this at the example of object-space ambient occlusion and soft shadows.

Support for secondary effects must already be considered in the preprocess of a data set as this requires storage of unresolved bricks. In addition, a minimum level of subdivision can be set regardless of the triangle sizes to maintain a desired minimum visual quality. To keep the resulting data sets compatible with our render oracle, we still store the triangle data at nodes of "correct" level, which may now be inner nodes instead of leaves. All children below these nodes then only contain RSBs and are not considered in the oracle evaluation, but solely used in the tracing of secondary effects.

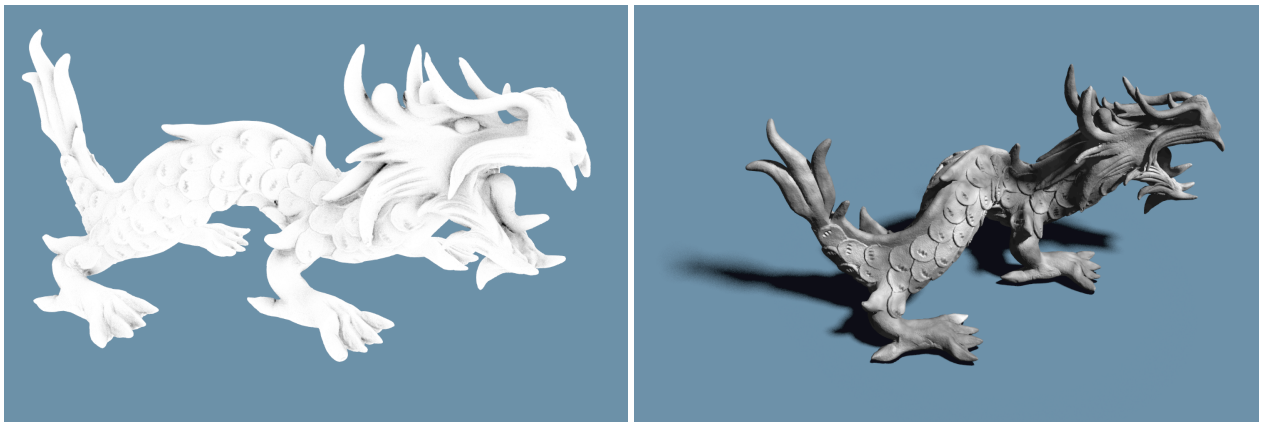


Figure 7.9: Secondary effects integrated into our rendering framework: Ambient occlusion (left) and soft shadows (right). These effects are traced on the RSB regardless of the oracle's decision for the primary rays.

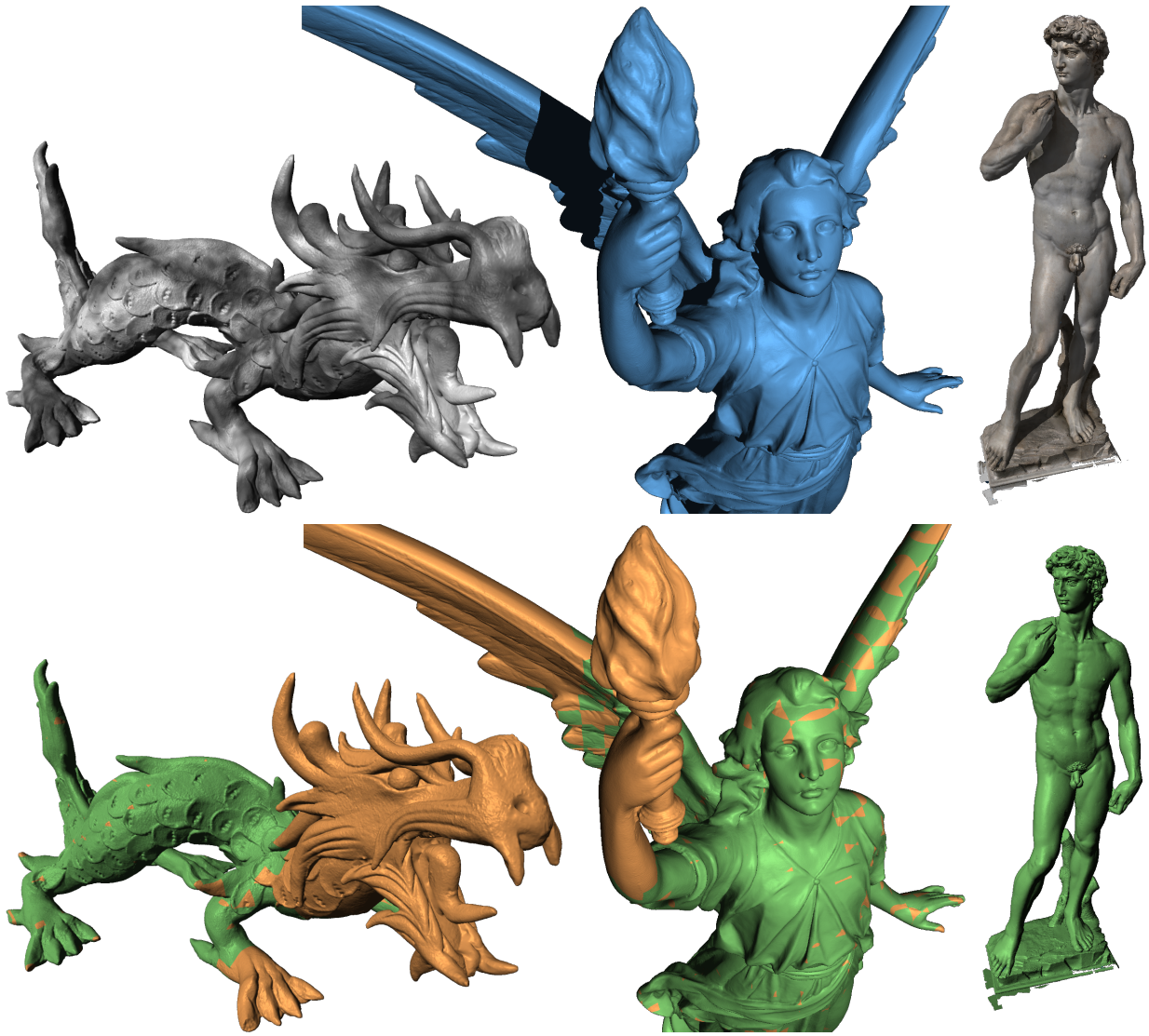
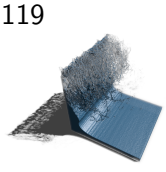


Figure 7.10: Dragon, Lucy, and David (view 1). The top row shows the rendered images, the bottom row the used rendering techniques for each brick (rasterization: orange, ray-casting: green).

7.5 Results

In this section, we analyze the performance and memory consumption of the hybrid rendering pipeline in comparison to pure rasterization and ray-casting of our RSB on the GPU. All timings were measured on a standard desktop PC, equipped with an Intel Core 2 Quad Q9450 2.66 GHz processor, 8 GB of RAM, and an NVIDIA Geforce GTX 680 graphics card with 2048 MB of local video memory. Rendering was always to a 1920×1080 viewport unless mentioned otherwise. Models with an inverse aspect ratio were rendered in landscape mode. The screen space error-tolerance was set to one pixel. In all polygonal test scenes, per-vertex attributes like colors and normals were resampled to the RSB.



As can be seen from all presented results, transitions from rasterized to ray-cast bricks are seamless in all cases and no noticeable quality differences or cracks exist between the two rendering methods. Since ray-casting does not provide a cost-efficient method for anti-aliasing, techniques such as MSAA need to also be disabled for the rasterizer. To compensate this, we apply SMAA [JESG12] as a post-process in our current implementation, which lead to overall good result.

Table 7.1: Model statistics: number of triangles, effective sampling resolution, total file size (triangle data, RSB, LoD), and size of triangle data only (in GB).

Model	Tris	Res	Mem
Dragon	9 M	$2K^3$	0.15 / 0.12
Lucy	28 M	$8K^3$	0.74 / 0.57
David	950 M	$8K^2 \times 32K$	24.44 / 18.80

Table 7.1 provides information concerning the used polygonal models. For the views in Fig. 7.10 and Fig. 7.1, Table 7.2 compares the performance of pure rasterization, pure RSB ray-casting, and the hybrid approach. Numbers denoted with a $\sim$ are estimates, because not all visible data could be stored in GPU memory at the required resolution. In this case, the rendering times would be vastly dominated by CPU/GPU data transfer.

Table 7.2: Rendering times in ms and GPU memory requirements in MB for a single frame as depicted in Figs. 7.10 and 7.1 (views from Fig. 7.1 are labeled 2 and 3 from left to right). Numbers denoted with a $\sim$ are estimates without CPU/GPU memory transfer.

Views	t_{rast}	t_{ray}	t_{hybrid}	Mem_{rast}	Mem_{ray}
Dragon	9.8	11.7	7.6	145	11
Lucy	25.0	29.8	21.8	477	58
David 1	$\sim$ 900.0	9.7	9.7	13083	15
David 2	$\sim$ 190.0	23.2	21.4	3810	65
David 3	79.4	34.1	27.5	1840	91

Especially for the David model, the hybrid pipeline shows a significant speed-up over pure rasterization. This is in particular due to the use of the sample-based LoD hierarchy to reduce the required memory per frame. Regardless of the view, the David model can always be rendered in less than 30 ms. On the other hand, even for the smaller models which fit entirely into GPU memory, and where large parts are rasterized, a noticeable performance gain is achieved. This demonstrates the efficiency of sample-based GPU ray-casting and emphasizes the importance of avoiding pixel overdraw via early ray-termination. This is confirmed by Fig. 7.11, where a sequence of views around the Lucy model was performed. The hybrid approach is always at least on par with pure rasterization and ray-casting, and usually

outperforms both. The memory requirement is reduced significantly. For comparison, the sparse voxel octree representation [LK10] of the Dragon model requires about 60 MB of GPU memory, whereas our representation uses only 30 MB.

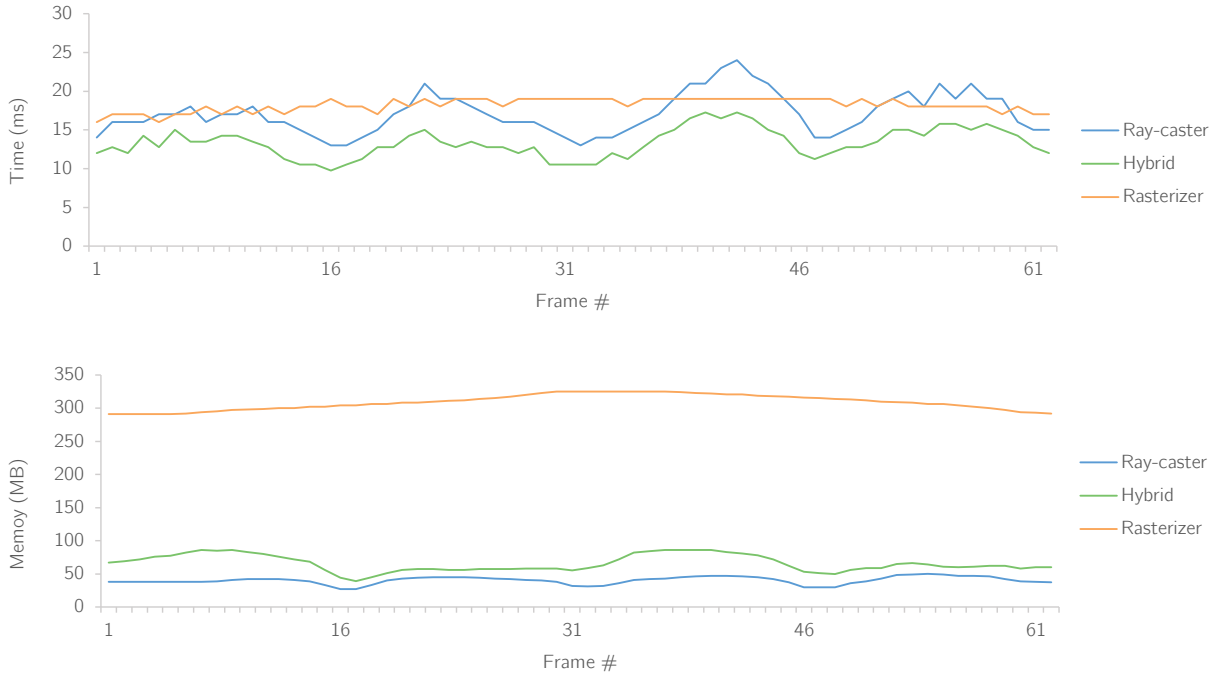


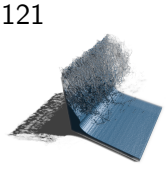
Figure 7.11: Recorded flight around Lucy. Render times and memory consumption are analysed.

Fig. 7.12 illustrates why ray-casting can be performed on the RSB very efficiently. It shows the number of surface samples stored at the cell of the 2D sampling grid where the surface intersection is actually found. Interestingly, in most cases only one single sample is stored, meaning that locally the surface is a height field over the 2D sampling domain. This confirms our expectation, that at a reasonable subdivision depth the surface parts in each brick tend to become height fields, even though the domain was chosen to be aligned with one of the brick faces.

7.6 Isosurface Ray-Casting

RSBs can also be constructed very efficiently from a scalar volume field. The process is similar to the one described for triangle data in section 7.3.3, yet RSB creation is performed during runtime in our rendering framework. The preprocess generates an octree subdivision of the volume data up to the finest available level and stores min/max values as well as a two-voxel wide overlap along with each brick.

Our framework (Fig. 7.13) is extended by a RSB brick processor that performs conversion from volume



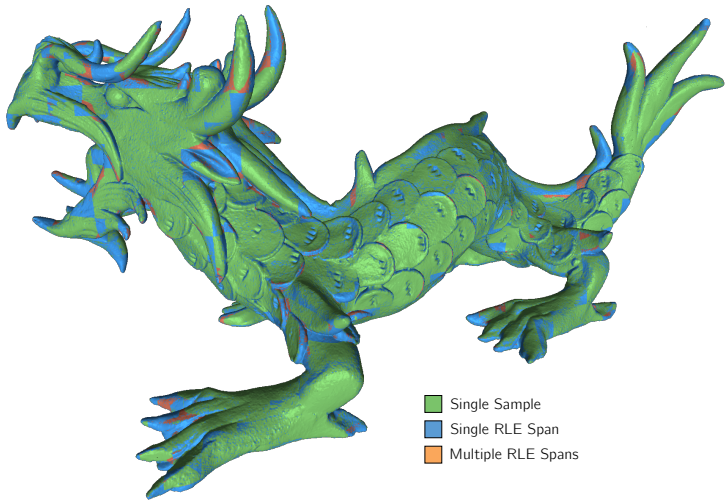


Figure 7.12: Color coding of number of samples that are stored at the grid cell where the surface intersection is found.

bricks to RSBs whenever required. To ensure a conservative approximation of the selected isosurface, we perform three sweeps through the density brick from orthogonal directions to detect the surface and write out a binary voxelization along with the average surface normal into a temporary brick buffer in this process. The resulting voxelization is then converted into an RSB as already presented. A brick is entered into the conversion queue whenever it becomes requested by the GPU and the RSB data is not yet residing in the GPU data caches, or if the selected isosurface has changed since its last conversion.

If a brick should be rendered at a higher sampling rate than the brick resolution, we fall back to classical isosurface ray-casting using trilinear interpolation.

For an evaluation, the following volume data sets were used: Ejecta is a simulation of the impact of a supernova ejecta on a companion star. It has a resolution of 4096^3, stores 2 bytes per voxel, and consumes 128 GB for the finest level of detail. Richtmyer is a simulation of a Richtmyer-Meshkov instability. It has a resolution of $2048^2 \times 1920$, stores 1 byte per voxel, and consumes 7.5 GB. Timing and memory statistics for Direct Volume Rendering (DVR) and Sample-Based Rendering (SBR) related to the views in Fig. 7.14 are given in Table 7.3. The viewport size was set to 1920 $\times$ 1080. For a fair DVR comparison, we simply disabled the SBR extension of our volume rendering system, visualizing each brick using classical isosurface ray-casting. Otherwise, the exact same system was used, including a LoD hierarchy on the volume data.

In all cases, we chose a brick size of 32^3. While smaller bricks can decrease the memory requirements for a particular isosurface, a spatially coherent organization of bricks on the GPU becomes more difficult and GPU cache mechanisms work less effective. In our experiments, and confirmed by state-of-the-art volume rendering systems [FSK13, GGM10], the selected brick size has shown a good trade-off.

It can be seen that the sample-based isosurface representation has a significantly lower memory footprint

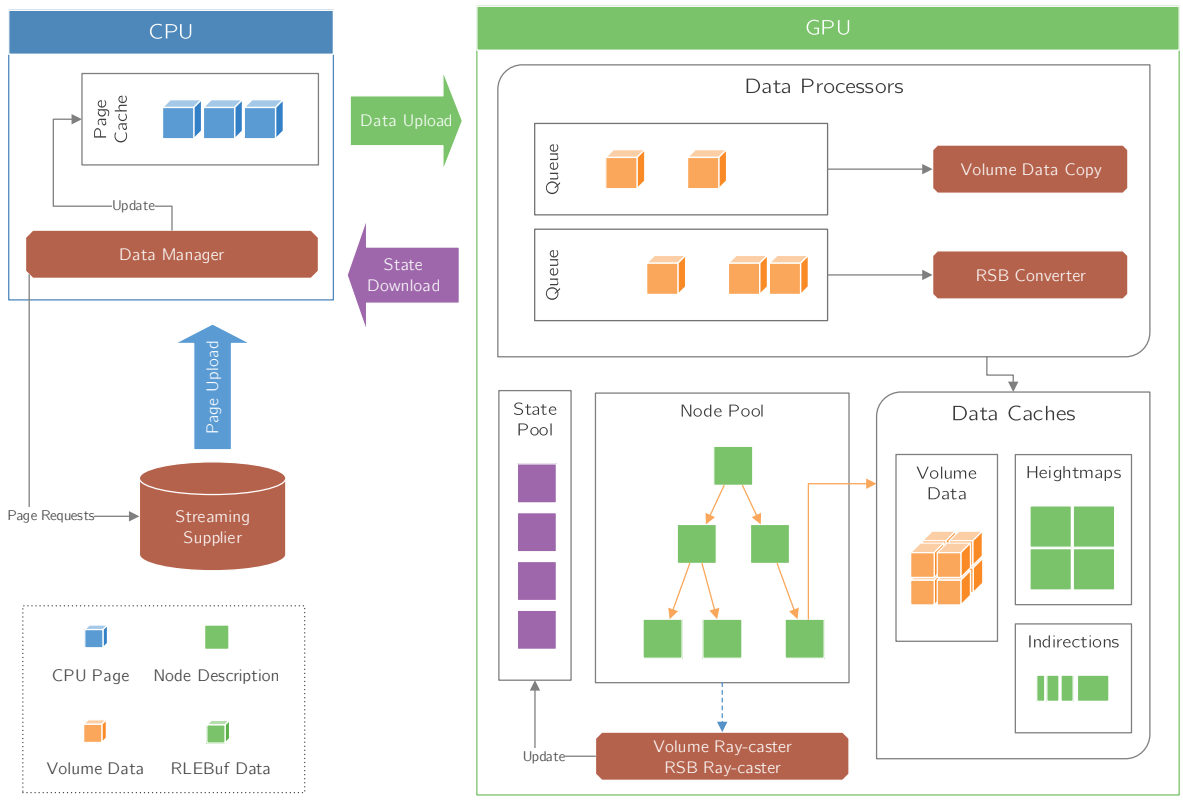
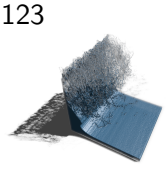


Figure 7.13: RSB rendering system for isosurface ray-casting. Bricks containing scalar density values are either directly rendered by an isosurface brick traverser or converted by a brick processor into RSBs when requested by the GPU.

on the GPU than DVR. In DVR, especially for large viewports even the data required for rendering a single frame might exceed the available GPU memory. Due to the compactness of our representation, the entire data needed to render the isosurface in the Richtmyer data set fits onto the GPU. One can also see that the sample-based representation can be built at rates vastly exceeding CPU/GPU bus bandwidth. Thus, even if the isosurface is changed, noticeable losses in performance are not introduced.

7.7 Conclusion and Future Work

We have presented a hybrid GPU pipeline for the rendering of polygon models with high geometric complexity. The pipeline does not replace but evolves the current graphics pipeline abstraction in that it uses rasterization and ray-casting simultaneously to generate eye-ray intersections. We have demonstrated that for very large triangle meshes at extreme resolution the hybrid pipeline can overcome performance limitations of pure rasterization. This has been achieved by introducing a sample-based surface representation which can be compactly encoded and traversed efficiently by many rays in parallel



	DVR		SBR		
Data set	Mem	t_{ray}	Mem	t_{ray}	Build (ms)
Ejecta	2.98	412	0.30	23	196
Richtmyer	1.02	29	0.22	21	134
Richtmeyer$_0$	3.24	819	0.73	67	324

Table 7.3: Memory and timing statistics for direct (DVR) and sample-based (SBR) volume rendering. Mem: memory consumption (in GB) for a single frame. t_r: rendering times (in ms, including data upload to GPU if required). Build: time required to convert all visible surface parts (with LoD error below one pixel) for the depicted images. Last row: statistics for the selected isosurface at the finest LoD of the entire Richtmyer data set.

on the GPU. Due to the ability to efficiently compute a LoD hierarchy on this representation, GPU resources can be employed effectively at run-time. We see our work as a step towards next generation rendering architectures that scale with respect to the overall GPU throughput and can thus satisfy the future demands of real-time graphics.

In the future, we will in particular investigate the parallelization of sample-based ray-casting on multi-core architectures. Multi-core architectures provide parallelism only to a modest degree, but they are highly optimized for minimizing latency in a single sequential task. Since sample-based ray-casting often suffers from latency issues caused by adjacent rays following different patterns through the sampling grids, it can be a good candidate for multi-core parallelism.

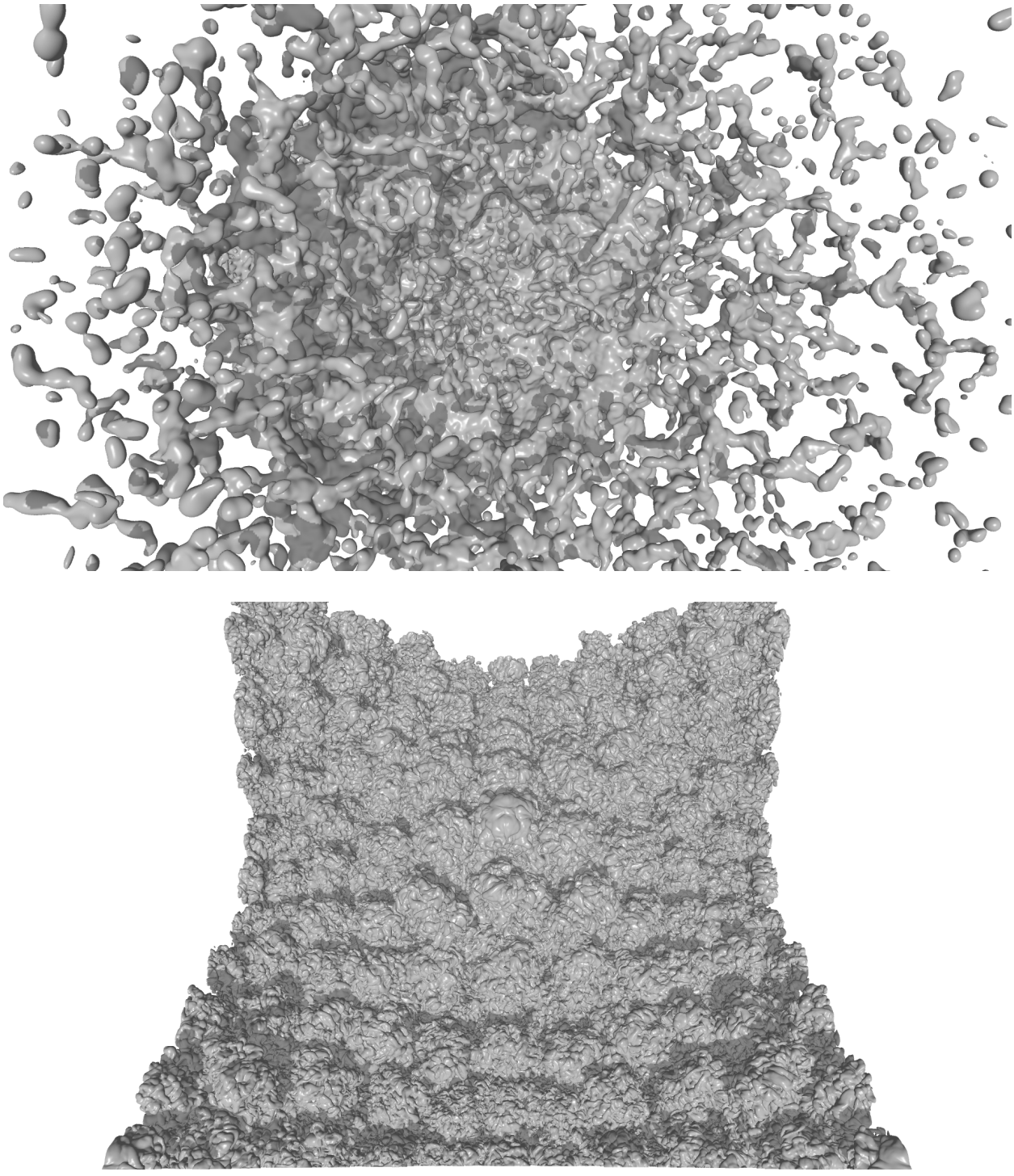
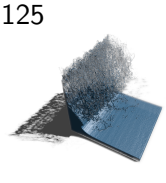


Figure 7.14: For the Ejecta (4096^3) and Richtmyer ($2048^2 \times 1920$) data sets, isosurface rendering onto a 1920×1080 viewport takes always less than 25 ms.



8

Interactive Reconstruction using Binary Grids

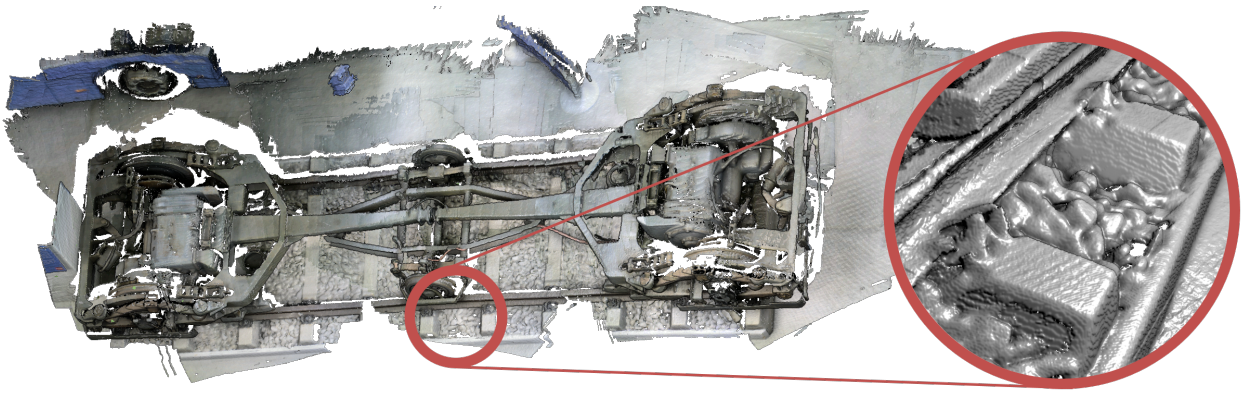
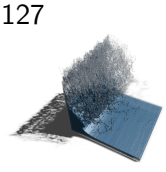


Figure 8.1: Reconstruction of the gears of a train. The scene is over 10 meters long. Data is gathered at varying levels of detail with a voxel resolution of up to 1 mm (2.12 mm on average over the whole scene) and requires 56 MB for surface- and 173 MB for color data, whereas previous works require 1.3 GB for the same level of detail.

8.1 Introduction and Related Work

Recent works in real-time scene reconstruction using active depth sensors have shown effective scalability in the scanned spatial extents and resolutions by using adaptive space partitions on the GPU, either encoded hierarchically [CBI13] or via spatial hashing [NZIS13]. We see, on the other hand, that regardless the preferred encoding, scalability is increasingly limited by the amount of memory that has to be moved—via read and write operations—and stored during online reconstruction. This becomes especially important in scenarios where the scene data is so large that it cannot be stored in local GPU or even main memory, which is likely the case when mobile scanning devices are mounted to moving vehicles and sent out for 3D reconstruction, surveillance and monitoring. In many real-world scenarios like



disaster management and damage surveys, construction monitoring, or vegetation and plant screening the use of such technologies will continue to increase in the near future.

Our research is motivated by the widening gap between data generation throughput and memory capacity, which will make large-scale and full-resolution online reconstruction prohibitively expensive. To overcome this limitation, the question needs to be addressed to what extent the amount of data required for scene reconstruction can be reduced. Most online reconstruction approaches using active depth sensors like Microsoft Kinect adopt incremental volumetric fusion of depth samples into a truncated signed distance field on a regular multi-block grid, accompanied by a cumulative weight field to average out noise and outliers. This approach goes back to early work by Hilton [Hil96] and Curless and Levoy [CL96], which has been extended further and adapted to online reconstruction. For thorough introductions to the field, including a consistent collection of related work, let us refer to [IKH*11, CBI13, NZIS13]. While these approaches enable high quality scalable reconstructions, they come at the additional expense of storing at least two scalar values for every volumetric sample, resulting in 4-6 bytes per voxel depending on the used internal data representation.

To keep memory requirements moderate, previous works acquired large scenes with a maximum voxel resolution of 4 to 10 mm, thereby undersampling the depth- and color resolution of the depth sensor. When the sensor resolution is matched, memory requirements increase considerably and GPU memory becomes occupied quickly. This problem will continue to increase in the future, especially on mobile devices equipped with lightweight sensors and limited on-board memory such as Google's project Tango, due to improvements in sensor hardware and the need to match the available resolution as pose estimations of higher accuracy become available [KSC13].

To avoid the memory overhead of a regular grid even when it is constraint to a narrow band around the scanned surface, some previous works made use of explicit surface representations using point samples [RL00, WWLG09]. Keller et al. [KLL*13] used such a representation for online reconstruction from depth images, including noise reduction via geometric averaging and spatial as well as temporal outlier removal. Explicit surface representations, on the other hand, must store the 3D positions of acquired surface points, and require rendering the point set for reconstruction and pose estimation.

Our work builds upon previous approaches and aims at reducing the memory requirements to enable scalability in both the scanned spatial extents and resolutions. In chapter 6, we presented binary voxel grids as a memory-efficient scene representation for particle-based data which comes at very little preprocessing time. These properties make the data structure a perfect fit for online reconstruction algorithms, as these require building and updating dynamic scenes at rates that, in the optimal case, are able to keep up with the rate at which data is delivered by a depth sensor. The basic idea is to perform scene reconstruction via volumetric fusion, but to encode the required weight and distance information only in two bits per voxel. This binary presentation can then be rendered efficiently by hierarchical DDA ray-marching—in contrast to our previous system, we do not require spheres of larger radius and resort to rendering each voxel as a cubical element. By once again employing deferred shading and

screen-space smoothing, our solution is able to display the reconstructed surface at no loss of visual accuracy, but requires up to an order of magnitude less memory than previous approaches.

We extend on our previous solution by adding a rapid GPU-based build process from a constant stream of RGB-D images. In contrast to our previously presented approaches, this adaption of our framework is entirely GPU driven, whereas the CPU is merely required to initiate render calls and download the generated data if caching in CPU memory needs to be performed. Consequently, a few alterations in data- and control flow need to be conducted. In addition, a number of novel changes have to be added to the fusion pipeline introduced in related works to exploit the potential of the binary voxel representation, in particular:

- We demonstrate windowed fusion of image streams to avoid storing distances and weights. Deferred processing of small sets of cached depth and color images is performed in regular intervals. We demonstrate memory saving of over 90% compared to previous approaches.
- We use the smoothed image of the ray-traced voxel surface for ICP-based pose estimation. A comparison shows similar accuracy and robustness as level-set rendering in a distance volume.
- We organize the scene data hierarchically in a hash map of adaptive octrees to support scenes of unlimited spatial extent without degrading voxel ray-casting performance.

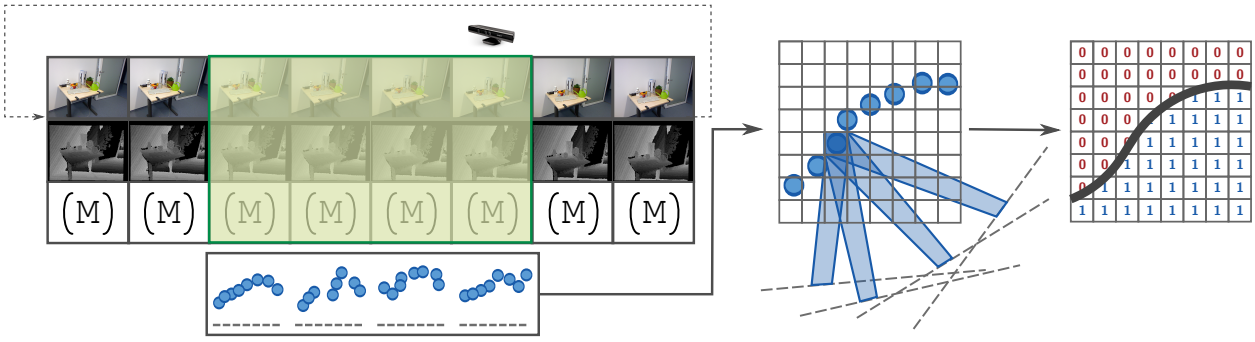
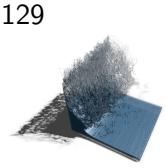


Figure 8.2: Windowed fusion. Captured sensor frames are stored in a cache of fixed size. M is a frame's pose estimation matrix. A window (green) of size n $(= 4)$ is processed in regular intervals of $\frac{n}{2}$ frames. Deferred processing projects each voxel into all frames of the cache window and computes averaged surface distance values. Subsequently, only a single bit indicating the sign of the final distance is stored in a binary voxel grid, and distances are deleted.

Figure 8.2 illustrates the basic concept underlying windowed fusion. One important observation motivating our approach is that the deferred distance and weight computation using a fixed number of acquired depth and color images can bypass the need to store a volumetric distance and weight field. As a consequence, the memory requirements are low and fixed over the complete reconstruction process. Furthermore, even at the resolution of the sensor hardware, the memory consumption of the binary voxel grid is considerably below that of a distance and/or weight volume. Once the scene has been scanned, a



smooth surface can be reconstructed from the binary representation, for instance, by using the method described in [Lem10] or mesh-based smoothing algorithms.

8.2 Volumetric Fusion

To reconstruct a surface from potentially uncertain measurements of surface points, volumetric fusion uses a truncated signed distance field (TSDF). At a discrete set of points, located on a regular multi-block grid restricted to the narrow band around the surface, the signed distances to the next measured surface points are computed, and these distances are averaged over a sequence of measurements [CL96]; this process is referred to as *integration*. In the distance field the surface is the zero level-set, and it can be reconstructed efficiently using surface fitting techniques [LC87] or GPU volume ray-casting [KW03].

When a depth image from a certain camera position is available, distance field computation is performed by sweeping through the voxel grid, projecting each voxel center into the depth image, and computing the distance between the voxel center and the acquired sample depth. A running average of all distances of a voxel is computed via a weight value which counts how often a voxel distance was updated. The weights compensate for the measuring inaccuracy in the captured depth data by averaging out these inaccuracies as well as outliers. The use of a truncation region determined by the noise characteristics of the sensor restricts the influence of each depth sample to its uncertainty region. It thus confines all calculations to a narrow band around the surface, reducing effectively the memory requirement and computational cost for updating distances and weights.

Our evaluations are based on a Microsoft Kinect 1, which captures depth and color data at a resolution of 640 $\times$ 480 pixels. It can be assumed that the uncertainty of each depth sample grows quadratically with increasing distance from the sensor [NIL12], whereas its size on the image plane grows linearly. To deal with a moving camera—which is typically the case—the current camera pose is estimated using the iterative closest point (ICP) method [BM92, CM92], and the new, noisy depth image is aligned with the rendered output of the fused surface.

In addition to the uncertainty in the measured depth values, due to perspective area foreshortening each depth- and color-sample in the sensor's xy-plane represents a scene sample of a size depending on the measured depth. The more distant the sample is from the camera, the larger is its extent in world space. Thus, using depth samples to compute the distance field on a fixed resolution grid—as it is typically done—will usually over- or undersample the sensor data, introducing loss of information or increasing memory usage. To avoid this, we locally adapt the resolution of the voxel grid on which samples are fused.

8.3 Windowed Fusion

Windowed fusion of scanned images is motivated by the way in which scanning is usually performed: As the camera is moved through the scene, each voxel receives relevant updates only over a few seconds. Then the observed surface is of final quality and does not need any further updates. We mimic this by windowing exact distance and weight calculations to few cached sensor frames.

8.3.1 Hierarchy Extension

To make the fusion process scalable in the spatial scene extent, the entire domain is partitioned into a set of evenly sized sub-domains (*world chunks*) with a side length of 2 meters. Chunks are created on demand and stored in a GPU hash table as proposed in [NZIS13]. Each chunk stores a sparse octree that discretizes the corresponding part of the domain using a voxel grid of size 8^3 at an *adaptive* spatial resolution. Fig. 8.3 illustrates the adaptive scene representation we employ; its details will be discussed in section 8.4.

The rationale behind the extension of our sparse octree scheme to a 3-level data structure is twofold: Firstly, as the extent of the domain is not known in advance and grows during runtime, a brick size that provides a good balance between memory consumption and rendering speed by restricting the tree depth cannot be determined in advance. In particular, if scanning is performed over a longer period of time, an overly deep tree of small bricks can degrade ray-casting performance. Secondly, world chunk creation can be performed at little overhead on the GPU and allows us to partition the scene in small and completely independent trees, reducing the number of threads that simultaneously update a single tree and thus the number of required locks and allocation stalls (as we will detail in section 8.4). The scheme also allows for a simple and efficient chunk-based data streaming solution (section 8.6).

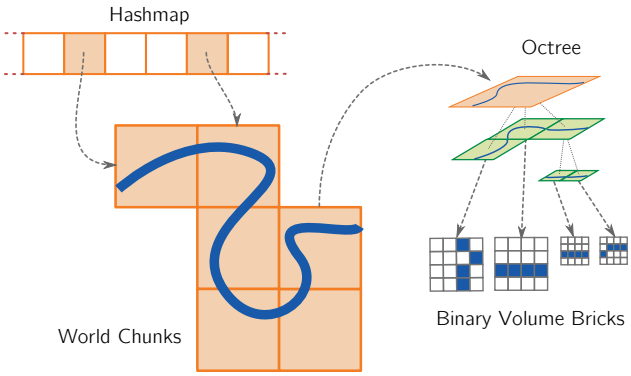
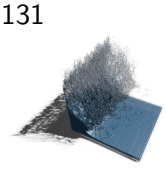


Figure 8.3: Hierarchical scene representation. The domain is partitioned into world chunks, each storing a sparse octree representation of a part of the scene. Leaf nodes store the binary scene encoding at different levels of detail.

While previous works already introduced multi-level data structures for volumetric fusion, they either used a very shallow hierarchy without storing surface data at multiple resolutions [CBI13], or they relied on reconstruction after scanning was completed [KSC13] without the need for intermediate rendering of the generated surface.



8.3.2 Deferred Processing

To maintain a compact scene representation, our goal is to avoid storing distances and weights. Therefore, we perform *deferred processing* of depth and color images: All distance and weight computations are deferred until a certain number of n images have been cached in GPU memory. At this point, we collect all bricks that were completely visible during at least one frame of the current cache. For each of these bricks, every voxel is projected into all cached images at once by a single GPU thread, and the distances and weights for each of them are accumulated in-turn. Subsequently, only a single *sign bit* is stored for each voxel, indicating the sign of the computed distance of the voxel center to the surface.

To smooth out tracking shifts from accumulating errors of the pose estimation, a number of frames are shared between subsequent caches. We maintain a ring-buffer cache of size $2n$ and shift a window of size n over it in steps of $\frac{n}{2}$ frames. This effectively doubles the interval at which integration is performed and greatly improves the reconstruction quality. Along with every cached depth- and color image we store the current pose estimation matrix.

8.3.3 Intermediate Sub-Cache Processing

Deferred processing as described introduces the following problem: Because a new scene representation is generated only every $\frac{n}{2}$-th frame—with n often being as large as 60—immediate visual feedback is prohibited and robust pose estimation becomes difficult. To overcome this problem, an additional processing pass is performed in each frame using the most recent $\hat{n}$ cached images (with $\hat{n} \ll n$). All newly created bricks that have not yet received an update from the cache are integrated from these $\hat{n}$ frames. These bricks are flagged as *uncertain* and will have their data replaced as soon as the next cache processing is performed. We will subsequently call the sequence of the last $\hat{n}$ frames the sub-cache.

8.3.4 Surface Confidence

Without explicit storage of voxel weights, the uncertainty of a stored sign bit can no longer be accessed once a cache has been integrated. Thus, voxels which are visible in only a small portion of the currently cached depth images may signal an event—a positive or negative distance sign—at less confidence than an event was signaled before at this location. We counteract this by storing an additional *confidence bit* with each voxel. This bit is set if the voxel has gathered a prescribed weight during processing. Each brick also stores the average estimated error of all its contained voxels, which is updated after cache integration. In successive caches, confident voxels can only be replaced if the new voxel will be flagged as confident again, and if the average error of the used depth samples is lower than the brick's previously estimated error.

Bricks processed from a sub-cache only contain non-confident voxels. All non-confident voxels are ignored for high-quality renderings and surface extractions, and they are only required for pose estimation during runtime. Figure 8.4 visualizes the classification for several frames.

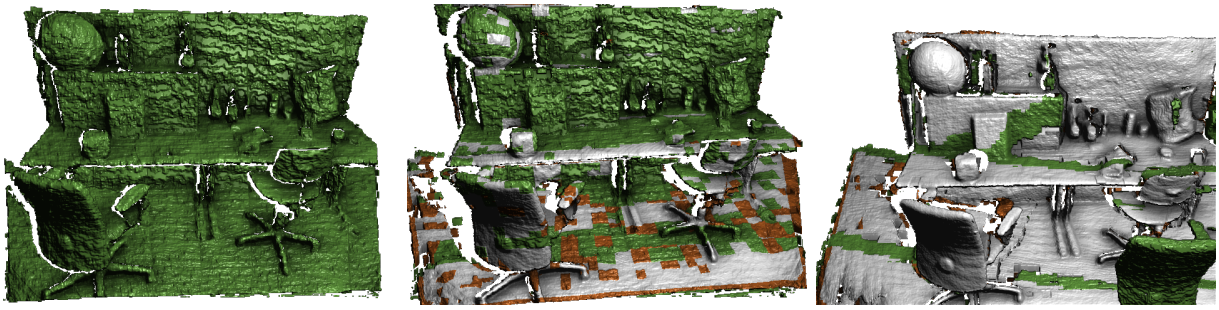


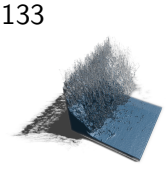
Figure 8.4: Before processing the first cache (left) the scene exists entirely of non-confident voxels from a sub-cache (green). During integration of the first cache (middle) more and more bricks are processed and confident voxels are produced (white). As scanning progresses, only a few new allocated bricks are processed from sub-caches (right). Orange voxels have been integrated, but were not flagged as confident. (The data set is part of the TUM RGB-D SLAM benchmark suite [SEE*12].)

8.4 Data Structures and Algorithms

In this section, we discuss the concrete implementation and data structures. All steps of the proposed algorithm are implemented on the GPU in compute shaders. Necessary CPU/GPU synchronization points are minimized by using indirect dispatching whenever possible, requiring only a single sync after windowed fusion to retrieve the root nodes of created trees for rendering and data streaming.

Figure 8.5 provides an overview of the resulting framework configuration. In particular, the setup has been changed as follows:

1. An *RGB-D supplier* delivers a constant stream of RGB-D images without any requests from the data manager.
2. No seperate state pool is employed for GPU feedback. Instead, a *chunk pool* provides a third hierarchy on top of the octrees stored in the node pool and contains the chunk metadata that needs to be downloaded to the CPU.
3. Brick- and data structure creation is entirely performed on the GPU. Consequently, the used processors generate data from the cached RGB-D images and directly fill the node pool, chunk pool and brick data caches.
4. Caching on the CPU is only performed for chunk data streamed out of GPU memory. No CPU caching needs to be performed for data delivered by the RGB-D supplier.



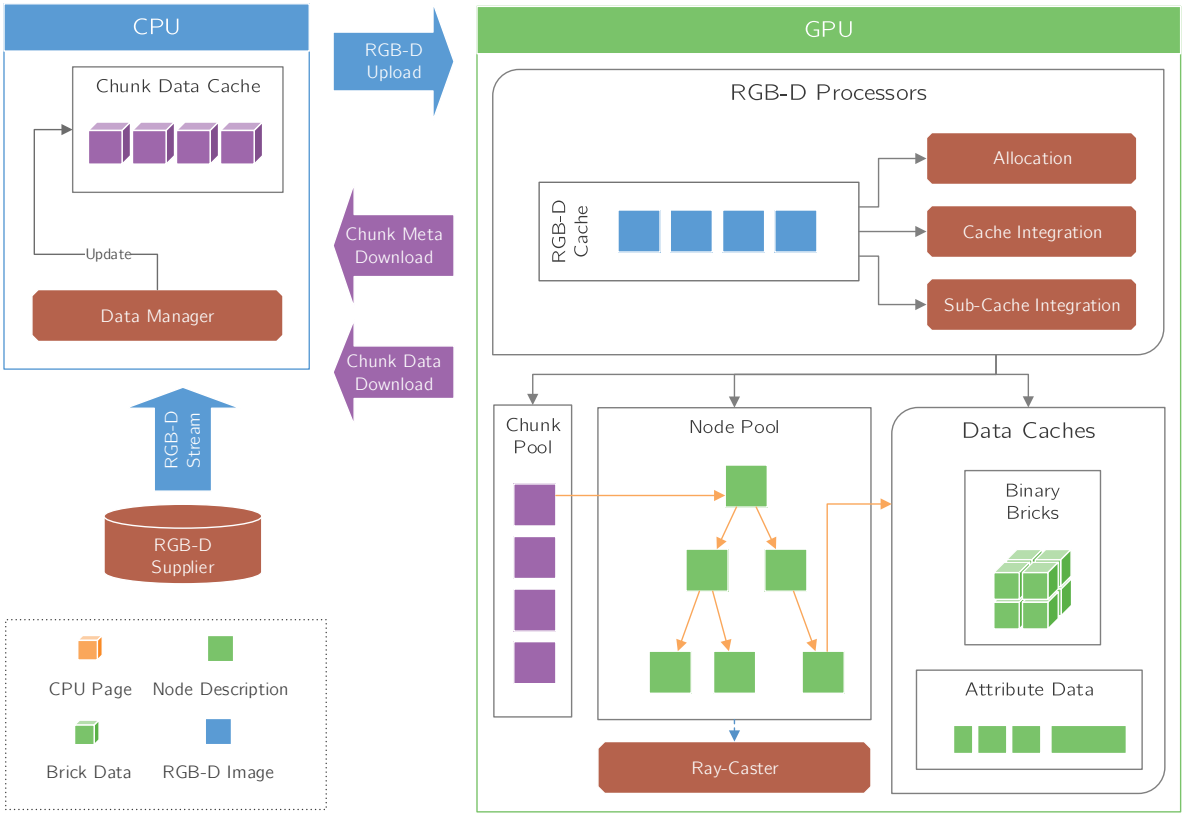


Figure 8.5: System Design. In contrast to our previous applications, the GPU is responsible for creating and managing nearly all of the required data caches. The CPU uploads streams of RGB-D images, from which the GPU fills our 3-level data structure of world chunks, octree nodes and binary data bricks. In each frame, only a small buffer of chunk metadata is transferred to the CPU along with chunk data that need to be streamed out of GPU memory.

8.4.1 Data structures

We build upon a three-level data structure as already introduced (see Fig. 8.3 for an illustration):

Chunks. At the first level, the domain is divided into evenly sized chunks. Each chunk stores its world-space position and a reference to the root of its associated octree (or NULL if the tree has not been generated yet):

```
struct Chunk {
  uint position;     // Z-Ordered
  uint rootPointer;
  uint offset;
}
```

Chunks are stored in a hash table as proposed by Nießner et al. [NZIS13]. Positions are hashed to buckets in a buffer of fixed size, where each bucket comprises a fixed number of slots. In case of an

overflow, the corresponding entry is placed in the next available slot and a list pointer of the last entry in the correct bucket is set to this slot. Given a careful selection of bucket size, overflows rarely ever happen and insertion and retrieval can be performed with minimal overhead. Furthermore, as we store comparably large chunks, the hash map access time is negligible.

Trees. All octrees are stored as previously in a linear *node pool* and indexed using a single pointer to index 8 child nodes which do not necessarily contain data:

```
struct TreeNode {
    uint childPointer;
    uint dataPointer;
    uint positionAndLevel;
    float avgError;
}
```

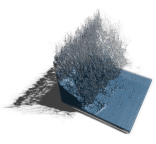
Binary volume bricks. At the leaf nodes, `dataPointers` reference cubic voxel bricks of fixed size in a 3D data pool. It is worth noting that the leaf nodes are at different spatial resolutions, depending on the uncertainty of the samples they store. Each voxel stores two bits: the sign and the confidence bit. In practice, we use two separate brick pools for both bits and store them in 3D textures with 32 bits per texel. As introduced in chapter 6, we will refer to groups of 32 bits as a single *sub-brick*.

8.4.2 Allocation

Each time a new frame is acquired by the sensor, allocation of all required data structures is performed instantly. Depth samples are inserted in parallel using one thread per pixel of the image. Each sample is projected into 3D space using the currently estimated pose. We then determine the index of the chunk containing the sample and find the corresponding entry in the hash map, or we insert it and create the root node of a new tree. The appropriate tree level is determined based on the sample's extent on the image plane, and the chunk's tree is traversed top to bottom to reach the leaf node where the sample is to be inserted. All nodes along the way are created, and a slot in the brick pool is allocated for the leaf node if necessary.

To perform slot allocation and deallocation, we keep a list of available slot pointers in a separate GPU buffer. Upon allocation, an atomic counter is increased and the slot at the previous counter position is used. For deallocation, the counter is decreased and the slot is added back into the list.

Special care has to be taken to assure that data allocation for a single pointer is performed by only one thread, even if multiple threads require allocation. This is done by storing a *lock flag* along with all data pointers. Whenever a thread needs to allocate data, brick nodes or world chunks, it first atomically locks the pointer and proceeds only if the lock succeeded. This may result in missing samples if a thread cannot acquire a lock. We address this problem by performing multiple insertion passes and storing all samples yet to be inserted in a temporary buffer, similar to the way proposed by Crassin et al. [CNS*11].



8.4.3 Integration

During scene capture, we store each depth and color map of the caches in two 2D texture arrays along with the corresponding transformation matrices. Before integration starts, we collect all bricks that were visible for at least one frame of the current cache. For each of these bricks, a single thread projects all voxels onto all of the depth maps (denoted with index i in the following). Computed distances are weighted by a weight W_i and accumulated in registers. For a voxel of position p, the distance $D(p)$ is

$$D(p) = \frac{\Psi(p)\Phi(p) + \sum_{i=0}^{n} W_i(p)D(p)}{\Psi(p) + \sum_{i=0}^{n} W_i(p)}, \tag{8.1}$$

where D_i denotes the signed distance from the voxel to the surface in frame i, which is positive if the voxel is in front of the surface. With respect to the uncertainty σ of a depth sample, the weights W_i are set to

$$W_i = \begin{cases} 1 & \text{iff } 0 < D_i(p) < \sigma \\ 1 - \frac{\sigma + D_i(p)}{\sigma} & \text{iff } -\sigma < D_i(p) <= 0 \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

If the according voxel has already been flagged as confident, we determine the initial distance $\Phi(p)$ by a fast sweep through the volume of a few voxels along the three principal axis. If a sign change at a confident voxel is detected along any of these sweeps, the distance between the voxel centers is used as $\Phi(p)$ with weight $\Psi(p)$. $\Psi(p)$ grows as the brick's average error decreases, or it is set to 0 if no surface is found. After a cache has been processed, all empty bricks are collected and removed.

As cache integration needs to be performed in regular intervals of $\frac{n}{2}$ frames, processing all bricks at once will lead to peaks in computation times during scene capture. For reliable pose estimation, however, it is crucial to maintain a high and stable frame-rate. We therefore distribute the integration of each cache over the subsequent $\frac{n}{2}$ frames, with $\frac{2}{n}$ of the visible bricks being integrated in each frame.

8.4.4 Rendering

The resulting surface is rendered using GPU front-to-back ray-casting. In analogy to [NZIS13], we first generate two z-Buffers containing the start and end points for every ray by rasterizing the bounding boxes of all currently allocated world chunks. This approach has the problem that empty space may not be skipped efficiently, but since we perform it only on the coarse space subdivision into chunks and can then use octree traversal for fine-granular empty-space skipping, no significant performance impact could be observed. We then perform DDA ray-marching through the world chunks along each viewing ray. In each chunk, if the corresponding entry exists in the hash table, the associated octree is traversed

with hierarchical DDA down to the finest existing leaf nodes, skipping all nodes that do not contain data.

Each leaf node, in turn, is again traversed using DDA until a change in the contained sign bits is detected. In this case the ray is intersected with a cube of the size of the voxel. During runtime, we employ deferred shading using finite-differences from the resulting rendered depth image, along with slight screen-space smoothing of our previously presented variant of the ROF filter. High-quality normals, if desired, can be extracted in a post-process from the binary isosurface, either using distance field generation or more sophisticated methods, as e.g. [Lem10].

8.5 Color Management

In principle, acquired colors can be stored in additional bricks comprising RGB voxels instead of binary values. However, this would blow up the memory requirements considerably, since also memory for non-occupied voxels is allocated. To avoid this, a brick's color data is stored at sub-brick granularity in blocks of $4 \times 4 \times 2$ voxels as already demonstrated in chapter 6. In contrast to our previous solution, we need to allocate all resulting and potentially filled 32 slots for each color block in the used 1D buffer. As previously, each sub-brick of the binary volume stores an additional 32 bit pointer into this buffer. This increases the total number of bits per voxel to 3. Colors are only stored for voxels that are flagged as confident and belong to the scanned surface. We define a confident voxel at a 3D position p to be a surface voxel if

$$D_{avg}(p) \leq \frac{1}{\sqrt{2}} \cdot (1.0f - \theta_{avg}(p)) \tag{8.3}$$

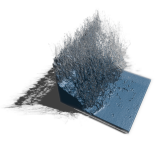
where $D_{avg}(p)$ is the weighted average distance, and

$$\theta_{avg}(p) = \frac{\sum_{i=0}^{n} (0,0,1)^T \cdot N_i(proj_i(p))}{n}. \tag{8.4}$$

Here, $\vec{N_i}$ is the screen-space normal at a 2D position of the i-th depth frame in the current cache, and $proj_i(p)$ projects the point p onto the 2D image plane of this frame.

8.5.1 Color Memory Management

The color buffer is managed by a virtual memory allocator (section 4.2.1). In the current implementation we use 32 bit virtual addressing, with a 19 bit page index and 13 bit offset inside the page. We address blocks of 32 voxels and store each color in 2 byte RGB565 encoding, which yields a maximum page size



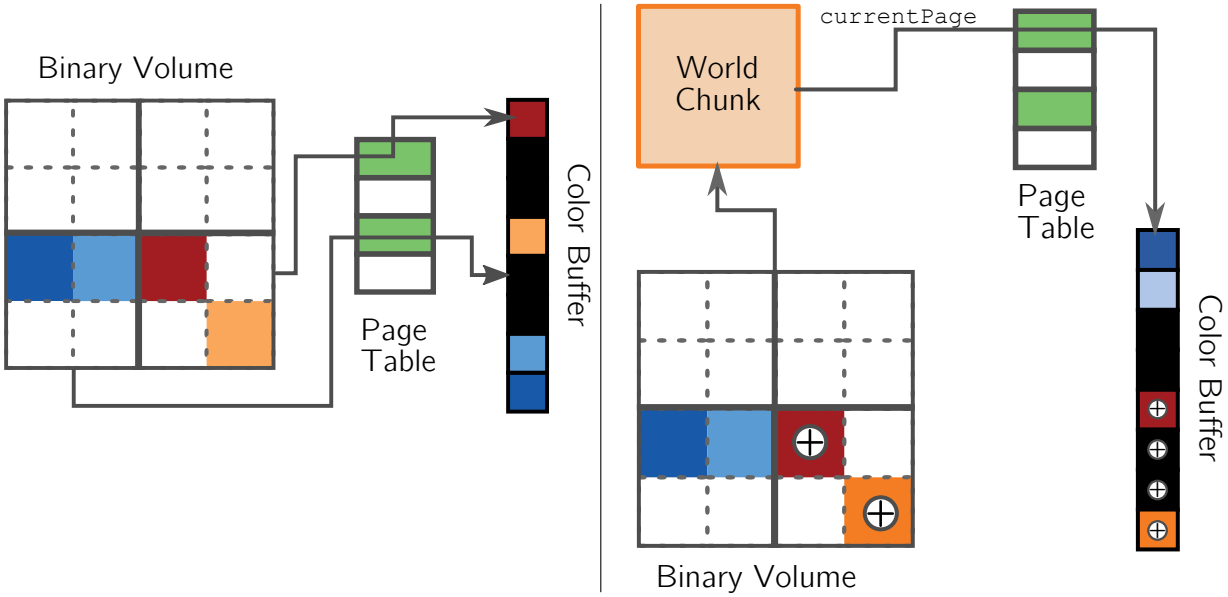


Figure 8.6: Color storage. Left: Each block (solid lines) contains a number of voxels (dotted lines) and a virtual pointer into the color buffer. Virtual pointers are translated into physical addresses using a page table, and combined with the voxel index inside the block to obtain the final address. Right: When new memory is allocated ($\oplus$ denotes the newly added colors), the `currentPage` pointer of the chunk is used and the data is appended to the page.

of 0.5 MB. In total, 256 GB of virtual address space can be managed in this way. A one-level page table is maintained on the GPU, providing space for all possible 2^{19} entries.

Whenever colors are accessed, the virtual address is translated into a physical address by performing a look-up in the page table and combining the 13 bit offset of the address with the physical page base address. Each page is exclusively reserved for a single world chunk (see Fig. 8.6).

Page Allocation. Page allocation is performed via a single atomic counter. Each world chunk keeps track of a `currentPage` pointer. Whenever color allocation is performed during scene integration, the size of the referenced page is atomically increased by 1.

Allocation of memory at the end of a page is done by atomically increasing the current page size. Due to parallel allocations performed by multiple threads, this may increase the indicated page size to a value above the maximum allowed size. If this is the case, the corresponding thread interprets the returned address as a failed allocation and instead requests a new page. To avoid multiple threads requesting new pages for a single world chunk, the current page pointer is atomically locked by all threads. Only the thread which successfully acquired the lock is allowed to increment the page and set the new page pointer. This can result in threads being unable to obtain a valid address, forcing the triggered allocations to be executed in subsequent passes. On NVIDIA GPUs, however, we utilize the fact that color allocation is performed only by a single thread per block during integration. With the size of each block matching the GPU warp size, we can safely put threads unable to acquire the lock

into a busy-waiting loop until the allocation has finished. In practice, page allocation occurs only rarely, not introducing any noticeable performance impact.

8.6 Data Streaming

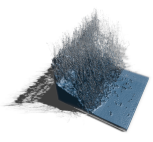
Even with the proposed compact binary scene representation, when scanning large spatial extents the acquired data can quickly become so large that it needs to be streamed from GPU into CPU memory, or even to external disk space. Data streaming is performed on the granularity of world chunks: All data of allocated chunks outside a spherical safety region around the camera can be removed from GPU memory and cached in CPU memory. If the user returns to an already observed region in the scene, required chunks are streamed back onto the GPU. Detection of the binary bricks and color pages is performed similar to the collection of visible bricks by accessing the brick pool and page table in parallel, and writing out a compact buffer of corresponding page and brick pointers.

Whenever a color page is removed, we employ lossless color compression to further reduce the memory for external storage. As color memory is allocated in blocks of fixed size, a large portion of each block will correspond to non-surface voxels and, thus, does not contain any color information. For typical scenes, this applies to about 60% of the data. Moreover, since colors are organized in a locally coherent manner, even non-empty neighboring values may be identical.

Our compression scheme is inspired by Treib et al [TRAW12] and builds upon the freely available *cudaCompress* library. The empty space present in the color data is first removed using a run length encoding, followed by *lossless* compression using parallel GPU Huffman encoding. The required huffman symbol table is computed for each page, allowing an independent (de-)compression of pages during data streaming. On average, this approach compresses color data by a factor of about $3.1 : 1$, with throughputs of 4 GB/s for compression and 8 GB/s for decompression, respectively. Since only comparably small chunks of the volume need to be streamed in each frame, the compression and decompression times are usually negligible.

8.7 Discussion and Results

In this section we discuss the benefits and drawbacks of the proposed extension of 3D online reconstruction. We performed a series of tests including live captures of scenes of various spatial extents to demonstrate the scalability and the quality of our approach. Some of our results are shown in Fig. 8.1 and Fig. 8.7. All tests were performed with a Microsoft Kinect 1.0, a dual quadcore Intel Xeon X5560 with 48 GB main memory, and an NVIDIA GTX Titan with 6 GB video memory. Unless mentioned otherwise, all images are taken from the online reconstruction, with normals extracted from the



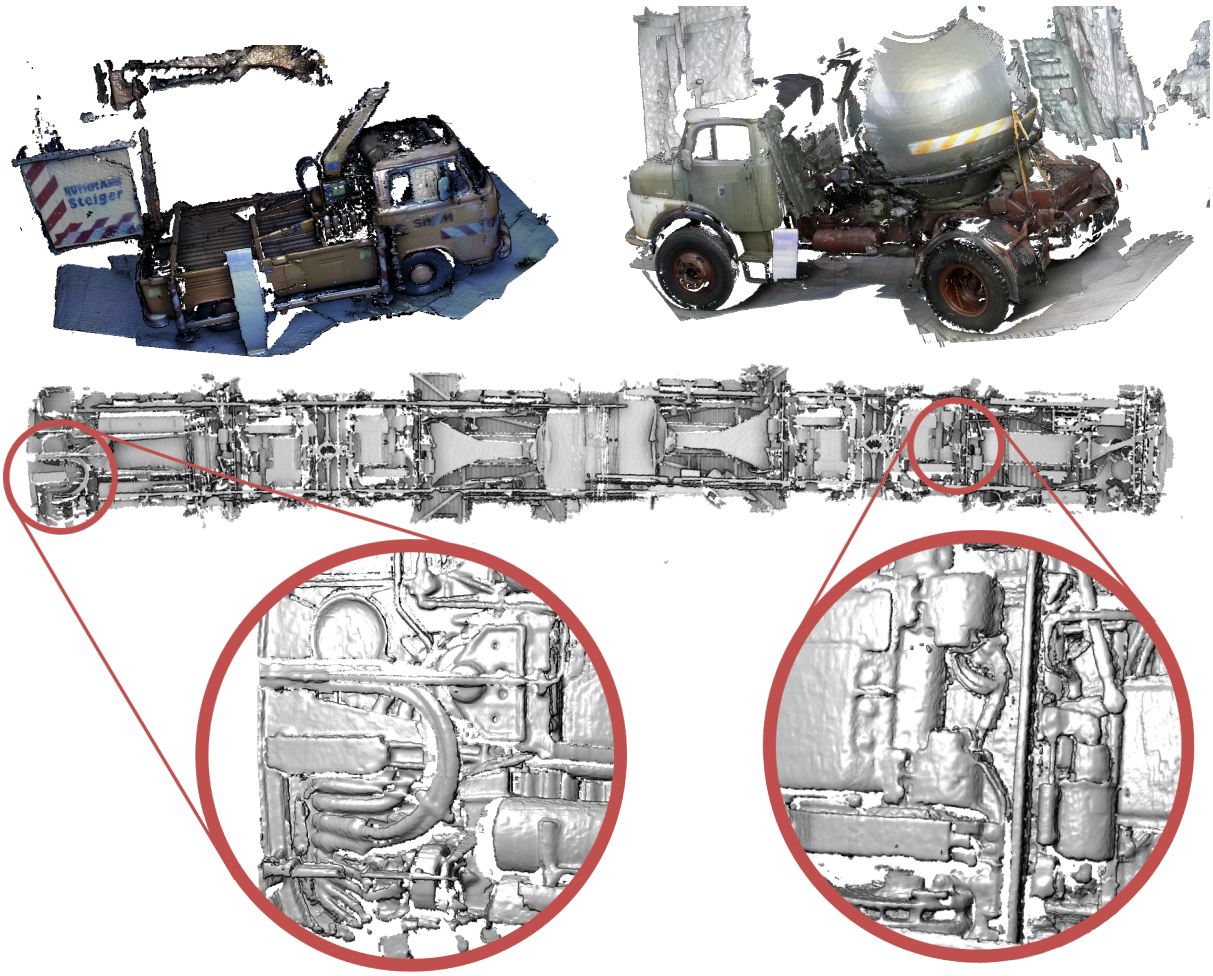


Figure 8.7: Testing data sets `hoist` (top left), `tank` (top right) and `below` (bottom). The last data set depicts the bottom of a tramway and is about 15 meters in length.

depth buffer during deferred shading. To compare our work to existing approaches, we use the available implementation of voxel hashing [NZIS13] (*VH*) as reference.

8.7.1 Memory Requirements

In a first experiment we analyze the effective voxel resolutions determined during online reconstruction. Figure 8.8 shows the number of non-empty bricks of several scans, using a chunk size of 2 meters and a brick size of 8^3 voxels. A voxel-to-pixel ratio of 1 was used to determine the level at which samples where inserted. It can be seen that only a few levels actually contain data at the end of a scan, with bricks mostly containing voxels of side lengths of about 4, 2 and 1 mm. The vast majority of bricks reside at the 2 mm resolution level for all data sets.

These resolutions serve as a baseline for a comparison of memory requirements and reconstruction speeds.

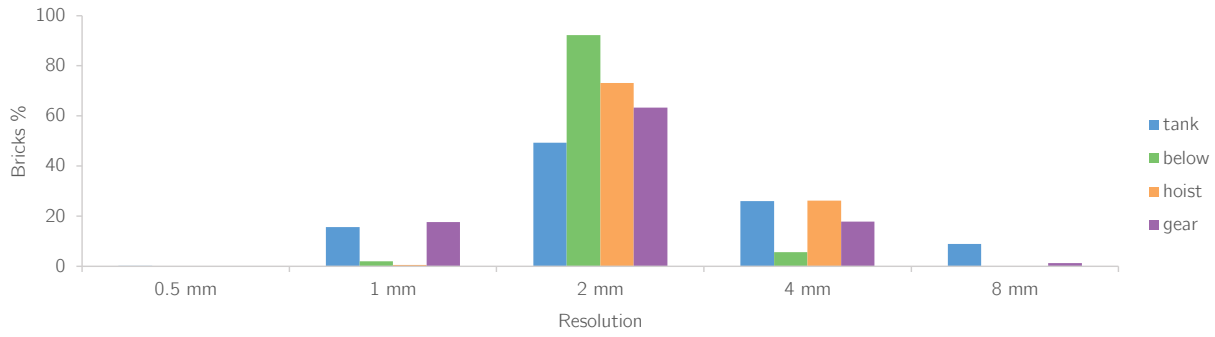


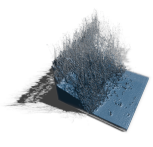
Figure 8.8: Distribution of non-empty leaf bricks across octree levels for various data sets. Given is the extent of a single voxel of the corresponding bricks in each dimension.

For a high quality comparison, we perform the same scans with VH using the resolution required by the majority of voxels (termed VH_{HQ}). To accommodate for the non-adaptivity of the VH-approach, we set a rather aggressive maximum depth cut-off of 2 meters. In addition, we performed a scan of lower quality (VH_{LQ}) using twice the world-space voxel size with a depth cut-off of 5 meters. Results from our approach are measured for adaptive voxels up to the finest required resolution and a depth cut-off of 5 meters, as more distant data exhibits high amounts of noise and is unusable in practice. Table 8.1 clearly shows the vast reduction in memory requirements we are able to achieve, even in comparison to a non-adaptive approach of significantly coarser voxel resolution.

Table 8.1: Dataset memory- and performance statistics. We compare our system to voxel hashing (storing distances and weights in bricks of 8^3) including colors. t gives the average time required to generate a new image (including fusion, pose estimation and rendering). Our the total memory is given as surface-memory Mem_s (3 bits per voxel, including the color pointer) and color memory (compressed / uncompressed as Mem_{c_0} and Mem_{c_1}). The last two columns give to the GPU memory reduction compared to VH_{HQ}, without (2 bpv) and with colors (3 bpv + uncompressed color data). All data is given in MB (memory) or ms (times).

	VH_{HQ}		VH_{LQ}		Ours					
	Mem	t	Mem	t	Mem_s	Mem_{c_0}	Mem_{c_1}	t	ΔSurf.	ΔTotal
below	1496	34	416	32	**64**	**178**	57	27	94 %	83 %
tank	887	42	837	40	**54**	**171**	55	36	91 %	74 %
gear	1290	41	637	36	**56**	**173**	55	32	94 %	82 %
hoist	753	42	644	33	**45**	**137**	44	34	92 %	76 %

If color storage is enabled, our sparse allocation scheme is able to decrease the required memory by 45% on average in comparison to per-voxel color storage, with an additional factor of 2 gained by storing colors in the RGB565 format. Upon data streaming, an average reduction factor of 3.1 is achieved via lossless color compression. In VH, colors increase all memory requirements by 100%.



8.7.2 Performance

During the same set of experiments, the overall system performance was measured. Average timings for each data set are included in Fig. 8.1. Notably, we outperform VH in high quality mode even though our integration shader performs considerably more operations by projecting the respective voxels into a number of depth maps. This is due to the hierarchically reduced number of bricks in more distant regions and the distribution of cache processing over multiple frames.

Detailed performance statistics for an exemplary data set are shown in Fig. 8.9. As expected, cache integration consumes most of the GPU time (11.8 ms on average), with significantly less work performed on the sub-cache (3.1 ms). Color allocation, performed only for the larger cache, impacts performance only marginally, even if we employ per-warp busy waiting for page creation. Ray-casting (8.98 ms) shows to be more expensive than the 5 ms reported by Nießner et al., which we attribute to the higher level of thread divergence due to the increased resolution of our scenes and the additional indirection required for virtual color addressing. Surprisingly, chunk and tree creation as well as brick data allocation is negligible (1.2 ms). The remaining time per frame accounts to data streaming, pose estimation, and shading, yielding a total of 31.3 ms per frame in average. As can be seen, our approach is very well able to match the Kinect's rate of 30 frames per second in a typical scanning procedure. When integration of the first sub-cache is performed, a sudden peak in processing time is observed due to the large number of newly created bricks during the first frames, but the frame rate quickly stabilizes as soon as the cache has been filled once.

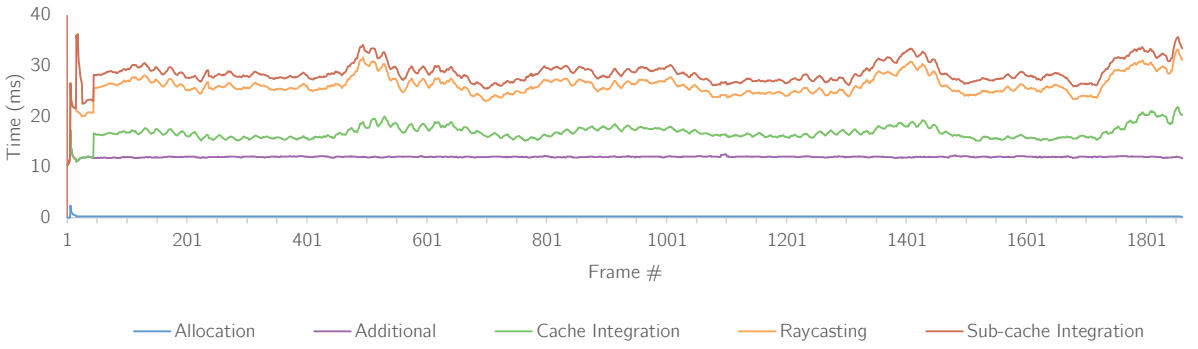


Figure 8.9: Performance analysis of our reconstruction pipeline. *Additional* timings include pose estimation, data streaming, and shading. Timings add up to 31.3 ms avg.

In most cases, the cache size n was set to 40, with a sub-cache of size $\hat{n} = 10$. It is important to note, however, that increasing n does not necessarily decrease performance: Integration of a cache is distributed over all frames of the next cache by processing $\frac{1}{n}$ of the required bricks in each frame. Increasing the cache size thus reduces the number of threads while increasing the workload per thread. Performance-wise, the "correct" size is dependent on the combination of brick size, scanned surface, and speed of movement. In our experiments, caches between 20 and as much as over 100 performed

similarly well on average. Smaller caches are able to include a larger portion of confident voxels for pose estimation, making it less prone to drifts, but may exhibit a slightly increased amount of noise in the final reconstruction.

8.7.3 Surface Quality

The use of a binary voxel representation in combination with DDA ray-marching results in a less smooth, blocky appearance of the surface during capture. However, since the rendering resolution matches the sensor resolution, a pixel-to-voxel ratio of 1:1 in addition to feature-preserving screen-space smoothing can hide these structures and enables smooth normal calculation. High-quality meshes can then be extracted in an offline process as shown in Fig. 8.10. A mesh-based Laplacian filter has been applied after extraction to clean up the artifacts resulting from the discrete marching cubes on the binary grid.

Table 8.2: Deviation (as root mean square error) of trajectories generated from pose estimation in our system to the "ground truth" trajectory generated by VH.

Data Set	Images	Error (RMSE)
below	1797	0.015 m
tank	3677	0.018 m
gear	1861	0.021 m
hoist	2432	0.031 m

As ICP-based pose estimation uses the rendered image to determine reference points, it must be assured that our representation does not affect its accuracy. Again, we employ VH as ground truth for the generated trajectory, and we measure the deviation of the trajectory that is generated by our system by means of the absolute translational root mean square error (RMSE). As shown in table 8.2, the generated paths differ only in the order of very few centimeters for all of our data sets.

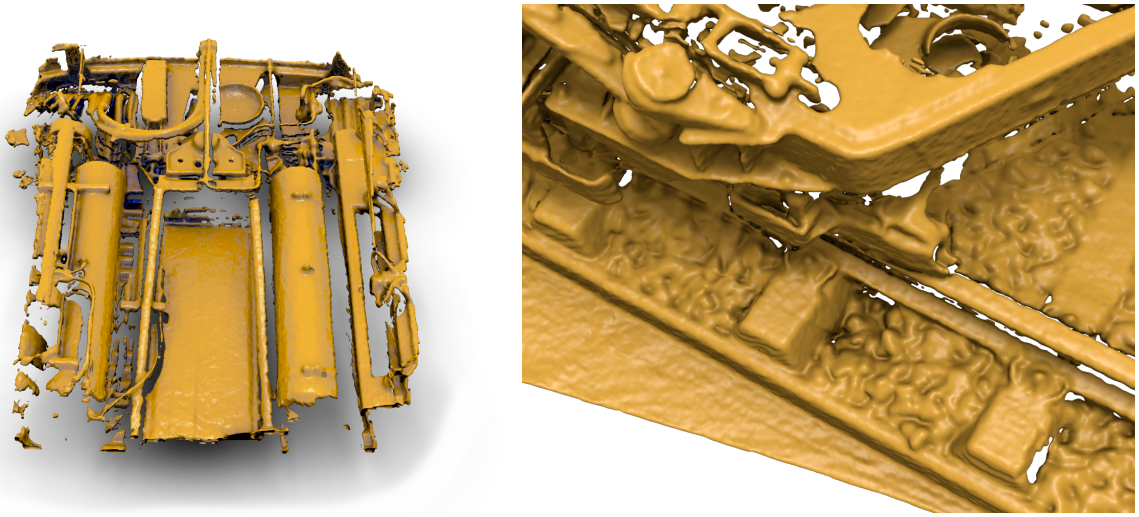
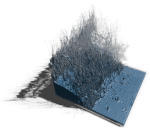


Figure 8.10: High quality meshes extracted from our binary representation.



8.8 Conclusion

In this chapter we have proposed a memory-efficient real-time variant of volumetric fusion using a commodity depth sensor. In contrast to previous approaches, we introduced *deferred integration* to eliminate the need for explicit storage of distance fields and associated weights. In combination with a very compact binary encoding of surface samples this greatly reduces the overall memory requirements. In addition, by inserting samples into an adaptive octree which is grown dynamically during runtime, we are able to represent all parts of the scanned surface at their adequate resolution, avoiding under- or over-sampling the sensor's depth and color data.

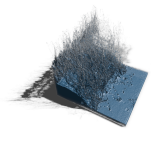
In the future we aim to answer especially the research question whether the computation of a distance field can be avoided entirely. Therefore, we will try to find means to reconstruct the surface during ray traversal from the surface bits in the surrounding of the sampling points along the ray. This approach will be very similar to the one proposed by Keller et al. [KLL*13], yet it will avoid storing point coordinates explicitly and performing costly search operations on the unstructured point set.

9 Conclusion

This thesis presented a system for rendering of high-resolution data sets from various scientific application domains. We were able to demonstrate interactive frame rates of billion-triangle models and simulations comprising over ten billion particles. We furthermore presented the application of our framework to dynamically generated data from surface scans with up to 9 million points acquired per second, fluid simulations of up to 500 million particles in over 200 time steps, and variable isosurfaces from volumetric data sets of resolutions as large as 4096^3.
From observations of challenges commonly present in big data visualization, we specifically derived the design for our framework based on recent state-of-the-art GPU ray-casting systems with streaming- and out-of-core capabilities through various stages of the memory hierarchy. By employing application-specific, highly compact data representations based on a resampling of the input data onto discrete grids, we could significantly reduce the memory requirements in comparison to the input data as well as to existing sample-based rendering systems.

In the future, an extension of the rendering system to additional classes of input data such as point data including colors and normals is an interesting possibility. While our fluid rendering application is already able to render spheres with arbitrary attributes, it cannot truthfully display sharp edges prominent in real-world geometry; our voxel-based approaches, on the other hand, require a denser sampling of the contained points—or a beforehand conversion into triangles from which a voxel representation is extracted, as demonstrated—to avoid holes in the reconstruction.

In addition, the support for more complex rendering effects including full global illumination by voxel path-tracing would greatly improve the visual quality. Especially for fluid rendering applications, a high-quality mode featuring exact refractions and caustics provides interesting challenges, e.g. the missing of smooth normals on all intersection points after the first, which could be obtained by applying larger, object-space smoothing filters on the particle sets. In applications with LoD-support, the required level can then be determined based on perceptual metrics and desired the quality of each individual effect.



Bibliography

[AAIT12] Akinci G., Akinci N., Ihmsen M., Teschner M.: An efficient surface reconstruction pipeline for particle-based fluids. In *VRIPHYS* (2012), Bender J., Kuijper A., Fellner D. W., Guérin E., (Eds.), Eurographics Association, pp. 61–68.

[Áfr12] Áfra A. T.: Interactive ray tracing of large models using voxel hierarchies. *Comput. Graph. Forum 31*, 1 (2012), 75–88.

[AIAT12] Akinci G., Ihmsen M., Akinci N., Teschner M.: Parallel surface reconstruction for particle-based fluids. *Comput. Graph. Forum 31*, 6 (2012), 1797–1809.

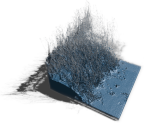
[AKL13] Aila T., Karras T., Laine S.: On quality metrics of bounding volume hierarchies. In *High Performance Graphics* (2013), Fatahalian K., Theobalt C., Lehtinen J., (Eds.), ACM, pp. 101–108.

[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on GPUs. In *High Performance Graphics* (2009), Spencer S. N., McAllister D. K., Pharr M., Wald I., Luebke D. P., Slusallek P., (Eds.), ACM, pp. 145–149.

[ALD06] Adams B., Lenaerts T., Dutre P.: *Particle Splatting: Interactive Rendering of Particle-Based Simulation Data*. Technical report cw 453, Katholieke Universiteit Leuven, 2006.

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: *Real-Time Rendering, Third Edition*. Taylor & Francis, 2008.

[APKG07] Adams B., Pauly M., Keiser R., Guibas L. J.: Adaptively sampled particle fluids. *ACM Trans. Graph. 26*, 3 (2007), 48.



[App68] Appel A.: Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 37–45.

[ATW13] Ando R., Thürey N., Wojtan C.: Highly adaptive liquid simulations on tetrahedral meshes. *ACM Trans. Graph. (Proc. SIGGRAPH 2013)* (July 2013).

[BA02] Bærentzen J. A., Aanæs H.: Generating signed distance fields from triangle meshes. *Informatics and Mathematical Modeling, Technical University of Denmark, DTU 20* (2002).

[BAM14] Barringer R., Akenine-Möller T.: Dynamic ray stream traversal. *ACM Trans. Graph. 33*, 4 (July 2014), 151:1–151:9. URL: `http://doi.acm.org/10.1145/2601097.2601222`.

[BB12] Boyd L., Bridson R.: Multiflip for energetic two-phase fluid simulation. *ACM Trans. Graph. 31*, 2 (2012), 16:1–16:12.

[BBB07] Batty C., Bertails F., Bridson R.: A fast variational framework for accurate solid-fluid coupling. *ACM Trans. Graph. 26*, 3 (2007).

[BD02] Benson D., Davis J.: Octree textures. In *SIGGRAPH* (2002), Appolloni T., (Ed.), ACM, pp. 785–790.

[BD06] Baboud L., Décoret X.: Rendering geometry with relief textures. In *Graphics Interface* (2006), Gutwin C., Mann S., (Eds.), Canadian Human-Computer Communications Society, pp. 195–201.

[Bel66] Belady L. A.: A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal 5*, 2 (1966), 78–101.

[BEL*07] Boulos S., Edwards D., Lacewell J. D., Kniss J., Kautz J., Shirley P., Wald I.: Packet-based whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007* (New York, NY, USA, 2007), GI '07, ACM, pp. 177–184. URL: `http://doi.acm.org/10.1145/1268517.1268547`.

[Ber04] Bertsekas D. P.: *Nonlinear Programming*, 2nd ed. Athena Scientific, 2004.

[BHGS06] Boubekeur T., Heidrich W., Granier X., Schlick C.: Volume-surface trees. *Comput. Graph. Forum 25*, 3 (2006), 399–406.

[BHKW07] Bürger K., Hertel S., Krüger J., Westermann R.: GPU rendering of secondary effects. In *VMV* (2007), Lensch H. P. A., Rosenhahn B., Seidel H.-P., Slusallek P., Weickert J., (Eds.), Aka GmbH, pp. 51–60.

[BHMF08] Beyer J., Hadwiger M., Müller T., Fritz L.: Smooth mixed-resolution GPU volume rendering. In *Volume Graphics* (2008), Hege H.-C., Laidlaw D. H., Pajarola R., Staadt O. G., (Eds.), Eurographics Association, pp. 163–170.

[BHS*11] Beyer J., Hadwiger M., Schneider J., Jeong W.-K., Pfister H.: Distributed terascale volume visualization using distributed shared virtual memory. In *LDAV* (2011), Rogers D., Silva C. T., (Eds.), IEEE, pp. 127–128.

[BKW10] Bürger K., Krüger J., Westermann R.: Sample-based surface coloring. *IEEE Trans. Vis. Comput. Graph. 16*, 5 (2010), 763–776.

[Bli78] Blinn J. F.: Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph. 12*, 3 (Aug. 1978), 286–292.

[BM92] Besl P. J., McKay N. D.: A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell. 14*, 2 (Feb. 1992), 239–256.

[BMW*09] Bittner J., Mattausch O., Wonka P., Havran V., Wimmer M.: Adaptive global visibility sampling. *ACM Trans. Graph. 28*, 3 (2009).

[BN76] Blinn J. F., Newell M. E.: Texture and reflection in computer generated images. *Commun. ACM 19*, 10 (1976), 542–547.

[BN12] Bruneton E., Neyret F.: A survey of nonlinear prefiltering methods for efficient and accurate surface shading. *IEEE Transactions on Visualization and Computer Graphics 18*, 2 (Feb. 2012), 242–260.

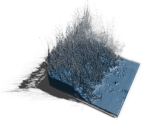
[BNS01] Boada I., Navazo I., Scopigno R.: Multiresolution volume visualization with a texture-based octree. *The Visual Computer 17*, 3 (2001), 185–197.

[BR86] Brackbill J. U., Ruppel H. M.: Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *J. Comput. Phys. 65*, 2 (1986), 314–343.

[Bri03] Bridson R. E.: *Computational Aspects of Dynamic Surfaces*. PhD thesis, Stanford University, 2003.

[BT05] Brawley Z., Tatarchuck N.: *Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing*. Charles River Media Graphics. Charles River Media, 2005, ch. Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing, pp. 135–154.

[BW03] Bittner J., Wonka P.: Visibility in computer graphics. *Environment and Planning B: Planning and Design 30*, 5 (Sept. 2003), 729–756.



[BWPP04] Bittner J., Wimmer M., Piringer H., Purgathofer W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Comput. Graph. Forum 23*, 3 (2004), 615–624.

[Cat74] Catmull E. E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, The University of Utah, 1974. AAI7504786.

[CB04] Christensen P. H., Batali D.: An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques* (2004), Keller A., Jensen H. W., (Eds.), Eurographics Association, pp. 133–142.

[CBI13] Chen J., Bautembach D., Izadi S.: Scalable real-time volumetric surface reconstruction. *ACM Trans. Graph. 32*, 4 (July 2013), 113:1–113:16.

[CCF95] Cabral B., Cam N., Foran J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS* (1995), pp. 91–98.

[CDM06] Childs H., Duchaineau M. A., Ma K.-L.: A scalable, hybrid scheme for volume rendering massive data sets . In *EGPGV* (2006), Heirich A., Raffin B., dos Santos L. P. P., (Eds.), Eurographics Association, pp. 153–161.

[CGG*03] Cignoni P., Ganovelli F., Gobbetti E., Marton F., Ponchio F., Scopigno R.: Bdam - batched dynamic adaptive meshes for high performance terrain visualization. *Comput. Graph. Forum 22*, 3 (2003), 505–514.

[CGG*04] Cignoni P., Ganovelli F., Gobbetti E., Marton F., Ponchio F., Scopigno R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23*, 3 (2004), 796–803.

[Cha04] Chambolle A.: An algorithm for total variation minimization and applications. *J. Math. Imaging Vis. 20*, 1-2 (2004), 89–97.

[CHCH06] Carr N. A., Hoberock J., Crane K., Hart J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Graphics Interface* (2006), Gutwin C., Mann S., (Eds.), Canadian Human-Computer Communications Society, pp. 203–209.

[CIPT14] Cornelis J., Ihmsen M., Peer A., Teschner M.: IISPH-FLIP for incompressible fluids. *Computer Graphics Forum 33*, 2 (May 2014), 255–262.

[CL96] Curless B., Levoy M.: A volumetric method for building complex models from range images. In *Proceedings of SIGGRAPH '96* (New York, NY, USA, 1996), ACM, pp. 303–312.

[CLT07] Crane K., Llamas I., Tariq S.: *GPU Gems*, vol. 3. Addison-Wesley Professional, 2007, ch. Real-Time Simulation and Rendering of 3D Fluids, pp. 633–675.

[CM92] CHEN Y., MEDIONI G.: Object modelling by registration of multiple range images. *Image Vision Comput. 10*, 3 (Apr. 1992), 145–155.

[CN94] CULLIP T. J., NEUMANN U.: *Accelerating Volume Reconstruction With 3D Texture Hardware*. Tech. rep., University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1994.

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *SI3D* (2009), Haines E., McGuire M., Aliaga D. G., Oliveira M. M., Spencer S. N., (Eds.), ACM, pp. 15–22.

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing: a preview. In *SI3D* (2011), Garland M., 0003 R. W., (Eds.), ACM, p. 207.

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C. T., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Trans. Vis. Comput. Graph. 9*, 3 (2003), 412–431.

[COK95] COHEN-OR D., KAUFMAN A. E.: Fundamentals of surface voxelization. *CVGIP: Graphical Model and Image Processing 57*, 6 (1995), 453–461.

[COM98] COHEN J., OLANO M., MANOCHA D.: Appearance-preserving simplification. In *IN PROC. SIGGRAPHâ€™98* (1998), pp. 115–122.

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1984), SIGGRAPH '84, ACM, pp. 137–145.

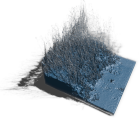
[Cra11] CRASSIN C.: *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. English and web-optimized version.

[CS09] CORDS H., STAADT O. G.: Interactive screen-space surface rendering of dynamic particle clouds. *J. Graphics, GPU, & Game Tools 14*, 3 (2009), 1–19.

[CSI09] CHA D., SON S., IHM I.: Gpu-assisted high quality particle rendering. *Comput. Graph. Forum 28*, 4 (2009), 1247–1255.

[CW14] CHAJDAS M. G., WESTERMANN R.: Quantitative analysis of voxel raytracing acceleration structures. In *Proceedings on the 22nd Pacific Conference on Computer Graphics and Applications (Pacific Graphics)* (2014).

[CZ67] CASE K., ZWEIFEL P.: *Linear transport theory*. Addison-Wesley series in nuclear engineering. Addison-Wesley Pub. Co., 1967.



[DDSD03] Décoret X., Durand F., Sillion F. X., Dorsey J.: Billboard clouds for extreme model simplification. *ACM Trans. Graph. 22*, 3 (2003), 689–696.

[DG12] Derzapf E., Guthe M.: Dependency-free parallel progressive meshes. *Comput. Graph. Forum 31*, 8 (2012), 2288–2302.

[DGPR02] DeBry D., Gibbs J., Petty D. D., Robins N.: Painting and rendering textures on unparameterized models. *ACM Trans. Graph. 21*, 3 (2002), 763–768.

[DH92] Danskin J., Hanrahan P.: Fast algorithms for volume ray tracing. In *Proc. Volume Visualization* (1992), pp. 91–98.

[DHW*11] Djeu P., Hunt W. A., Wang R., Elhassan I., Stoll G., Mark W. R.: Razor: An architecture for dynamic multiresolution ray tracing. *ACM Trans. Graph. 30*, 5 (2011), 115.

[DKW09] Dick C., Krüger J., Westermann R.: GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers* (2009), pp. 43–50.

[DKW10] Dick C., Krüger J., Westermann R.: GPU-aware hybrid terrain rendering. In *Proceedings of IADIS Computer Graphics, Visualization, Computer Vision and Image Processing 2010* (2010), pp. 3–10.

[DN04] Decaudin P., Neyret F.: Rendering forest scenes in real-time. In *Rendering Techniques* (2004), Keller A., Jensen H. W., (Eds.), Eurographics Association, pp. 93–102.

[DN09] Decaudin P., Neyret F.: Volumetric billboards. *Comput. Graph. Forum 28*, 8 (2009), 2079–2089.

[dTWL08] de Toledo R., Wang B., Lévy B.: Geometry textures and applications. *Comput. Graph. Forum 27*, 8 (2008), 2053–2065.

[EKE01] Engel K., Kraus M., Ertl T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2001), pp. 9–16.

[EMI01] Erikson C., Manocha D., III W. V. B.: Hlods for faster display of large static and dynamic environments. In *SI3D* (2001), Hughes J. F., Séquin C. H., (Eds.), ACM, pp. 111–120.

[Eng06] Engel K. D.: *Real-time volume graphics*. Peters, Wellesley, Mass., 2006.

[Eng11] Engel K.: Cera-TVr: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In *LDAV* (2011), Rogers D., Silva C. T., (Eds.), IEEE, pp. 123–124.

[FAW10] Fraedrich R., Auer S., Westermann R.: Efficient high-quality volume rendering of sph data. *IEEE Trans. Vis. Comput. Graph. 16*, 6 (2010), 1533–1540.

[FBH*10] Fatahalian K., Boulos S., Hegarty J., Akeley K., Mark W. R., Moreton H., Hanrahan P.: Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph. 29*, 4 (July 2010), 67:1–67:8.

[FC00] Fang S., Chen H.: Hardware accelerated voxelization. *Computers & Graphics 24*, 3 (2000), 433–442.

[FCS*10] Fogal T., Childs H., Shankar S., Krüger J., Bergeron R. D., Hatcher P.: Large data visualization on distributed memory multi-GPU clusters. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 57–66.

[FK10] Fogal T., Krüger J.: Tuvok, an architecture for large scale volume rendering. In *VMV* (2010), Koch R., Kolb A., Rezk-Salama C., (Eds.), Eurographics Association, pp. 139–146.

[FS05] Foley T., Sugerman J.: Kd-tree acceleration structures for a gpu raytracer. In *Graphics Hardware* (2005), Meißner M., Schneider B.-O., (Eds.), Eurographics Association, pp. 15–22.

[FSK13] Fogal T., Schiewe A., Krüger J.: An analysis of scalable GPU-based ray-guided volume rendering. In *Large Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on* (2013).

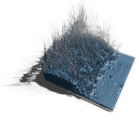
[FSW09] Fraedrich R., Schneider J., Westermann R.: Exploring the millennium run - scalable rendering of large-scale cosmological datasets. *IEEE Trans. Vis. Comput. Graph. 15*, 6 (2009), 1251–1258.

[GBBK04] Guthe M., Borodin P., Balázs Á., Klein R.: Real-time appearance preserving out-of-core rendering with shadows. In *Rendering Techniques* (2004), Keller A., Jensen H. W., (Eds.), Eurographics Association, pp. 69–80.

[GBK06] Guthe M., BalÃ¡zs Ã., Klein R.: Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Rendering Techniques* (2006), Akenine-Möller T., Heidrich W., (Eds.), Eurographics Association, pp. 207–214.

[GEM*13] Goswami P., Erol F., Mukhi R., Pajarola R., Gobbetti E.: An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer 29*, 1 (2013), 69–83.

[GGM10] Guitián J. A. I., Gobbetti E., Marton F.: View-dependent exploration of massive volumetric models on large-scale light field displays. *The Visual Computer 26*, 6-8 (2010), 1037–1047.



[GH97] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *SIGGRAPH* (1997), pp. 209–216.

[GH98] Garland M., Heckbert P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the Conference on Visualization '98* (Los Alamitos, CA, USA, 1998), VIS '98, IEEE Computer Society Press, pp. 263–269. URL: `http://dl.acm.org/citation.cfm?id=288216.288280`.

[GIK*07] Gribble C. P., Ize T., Kensler A., Wald I., Parker S. G.: A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 758–768.

[GKY08] Gobbetti E., Kasik D., Yoon S.-E.: Technical strategies for massive model visualization. In *Symposium on Solid and Physical Modeling* (2008), Haines E., McGuire M., (Eds.), ACM, pp. 405–415.

[GM77] Gingold R., Monaghan J.: Smoothed particle hydrodynamics: theory and application to non-spherical stars. 375–389.

[GM05] Gobbetti E., Marton F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3 (2005), 878–885.

[GMG08] Gobbetti E., Marton F., Guitián J. A. I.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7-9 (2008), 797–806.

[GPSS07] Günther J., Popov S., Seidel H.-P., Slusallek P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (Sept. 2007), pp. 113–118. `http://www.mpi-inf.mpg.de/~guenther/BVHonGPU/index.html`.

[GSSP10] Goswami P., Schlegel P., Solenthaler B., Pajarola R.: Interactive sph simulation and rendering on the gpu. In *Symposium on Computer Animation* (2010), Popovic Z., Otaduy M. A., (Eds.), Eurographics Association, pp. 55–64.

[GW06] Gonzalez R. C., Woods R. E.: *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HBJP12] HADWIGER M., BEYER J., JEONG W.-K., PFISTER H.: Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Trans. Vis. Comput. Graph. 18*, 12 (2012), 2285–2294.

[HDD*93] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1993), SIGGRAPH '93, ACM, pp. 19–26.

[HE03] HOPF M., ERTL T.: Hierarchical splatting of scattered data. In *IEEE Visualization* (2003), Turk G., van Wijk J. J., II R. J. M., (Eds.), IEEE Computer Society, pp. 433–440.

[Hil96] HILTON A.: On reliable surface reconstruction from multiple range images. In *Proceedings of European Conference on Computer Vision* (1996), Springer-Verlag, pp. 117–126.

[HL79] HERMAN G. T., LIU H. K.: Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Images Processing 9*, 1 (1979), 1–21.

[HLE04] HOPF M., LUTTENBERGER M., ERTL T.: Hierarchical splatting of scattered 4d data. *IEEE Computer Graphics and Applications 24*, 4 (2004), 64–72.

[HNB*06] HOUSTON B., NIELSEN M. B., BATTY C., NILSSON O., MUSETH K.: Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM TOG 25*, 1 (2006), 151–175.

[Hop96] HOPPE H.: Progressive meshes. In *SIGGRAPH* (1996), pp. 99–108.

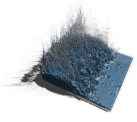
[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. In *SIGGRAPH* (1997), pp. 189–198.

[Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Computers & Graphics 22*, 1 (1998), 27–36.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree gpu raytracing. In *SI3D* (2007), Gooch B., Sloan P.-P. J., (Eds.), ACM, pp. 167–174.

[IAAT12] IHMSEN M., AKINCI N., AKINCI G., TESCHNER M.: Unified spray, foam and air bubbles for particle-based fluids. *The Visual Computer 28*, 6-8 (2012), 1–9.

[IKH*11] IZADI S., KIM D., HILLIGES O., MOLYNEAUX D., NEWCOMBE R., KOHLI P., SHOTTON J., HODGES S., FREEMAN D., DAVISON A., FITZGIBBON A.: Kinectfusion: Real-time 3D reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2011), UIST '11, ACM, pp. 559–568.



[IOS*14] Ihmsen M., Orthmann J., Solenthaler B., Kolb A., Teschner M.: Sph fluids in computer graphics. In *Eurographics 2014 State-of-the-Art Report* (2014), Eurographics Association, pp. 1–22.

[JESG12] Jimenez J., Echevarria J. I., Sousa T., Gutierrez D.: Smaa: Enhanced subpixel morphological antialiasing. *Comput. Graph. Forum 31*, 2 (2012), 355–364.

[JMW07] Jeschke S., Mantler S., Wimmer M.: Interactive smooth and curved shell mapping. In *Rendering Techniques* (2007), Kautz J., Pattanaik S. N., (Eds.), Eurographics Association, pp. 351–360.

[KK86] Kay T. L., Kajiya J. T.: Ray tracing complex scenes. In *SIGGRAPH* (1986), Evans D. C., Athay R. J., (Eds.), ACM, pp. 269–278.

[KLL*13] Keller M., Lefloch D., Lambers M., Izadi S., Weyrich T., Kolb A.: Real-time 3D reconstruction in dynamic scenes using point-based fusion. In *3DV* (2013), IEEE, pp. 1–8.

[Kno65] Knowlton K. C.: A fast storage allocator. *Commun. ACM 8*, 10 (Oct. 1965), 623–624.

[Krü10] Krüger J.: A new sampling scheme for slice based volume rendering. In *Volume Graphics* (2010), Westermann R., Kindlmann G. L., (Eds.), Eurographics Association, pp. 1–4.

[KSA13] Kämpe V., Sintorn E., Assarsson U.: High resolution sparse voxel dags. *ACM Trans. Graph. 32*, 4 (2013), 101.

[KSC13] Kerl C., Sturm J., Cremers D.: Dense visual SLAM for rgb-d cameras. In *Proc. of the Int. Conf. on Intelligent Robot Systems (IROS)* (2013).

[KSN08] Kanamori Y., Szego Z., Nishita T.: Gpu-based fast ray casting for a large number of metaballs. *Computer Graphics Forum 27*, 2 (2008), 351–360.

[KSW05] Krüger J., Schneider J., Westermann R.: Duodecim - a structure for point scan compression and rendering. In *Proceedings of the Symposium on Point-Based Graphics 2005* (2005).

[KTI*01] Kaneko T., Takahei T., Inami M., Kawakami N., Yanagida Y., Maeda T., Tachi S.: Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001* (2001), pp. 205–208.

[KTW*11] Knoll A., Thelen S., Wald I., Hansen C. D., Hagen H., Papka M. E.: Full-resolution interactive CPU volume rendering with coherent bvh traversal. In *PacificVis* (2011), Battista G. D., Fekete J.-D., Qu H., (Eds.), IEEE, pp. 3–10.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *IEEE Visualization* (2003), Turk G., van Wijk J. J., II R. J. M., (Eds.), IEEE Computer Society, pp. 287–292.

[KWPH06] KNOLL A., WALD I., PARKER S., HANSEN C.: Interactive isosurface ray tracing of large octree volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 115–124.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, pp. 163–169.

[LD07] LEFEBVRE S., DACHSBACHER C.: Tiletrees. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 25–31.

[Lem10] LEMPITSKY V. S.: Surface extraction from binary volumes with higher-order smoothness. In *CVPR* (2010), IEEE, pp. 1197–1204.

[Lev90] LEVOY M.: Efficient ray tracing of volume data. *ACM Trans. Graph. 9*, 3 (1990), 245–261.

[Lev02] LEVENBERG J.: Fast view-dependent level-of-detail rendering using cached geometry. In *IEEE Visualization* (2002), pp. 259–265.

[LFH06] LI Y., FU C.-W., HANSON A.: Scalable WIM: Effective exploration in large-scale astrophysical environments. 1005–1012.

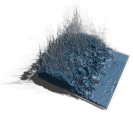
[LFWK03] LI W., FAN Z., WEI X., KAUFMAN A.: *GPU-Based flow simulation with complex boundaries*. Tech. rep., University of Wisconsin-Madison, 2003.

[LH91] LAUR D., HANRAHAN P.: Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *SIGGRAPH* (1991), Thomas J. J., (Ed.), ACM, pp. 285–288.

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. *ACM Trans. Graph. 25*, 3 (2006), 579–588.

[LHJ99] LAMAR E., HAMANN B., JOY K. I.: Multiresolution techniques for interactive texture-based volume visualization. In *IEEE Visualization* (1999), pp. 355–361.

[LHN05a] LEFEBVRE S., HORNUS S., NEYRET F.: Octree textures on the GPU. In *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 37, pp. 595–613.



[LHN05b] Lefebvre S., Hornus S., Neyret F.: Texture sprites: texture elements splatted on surfaces. In *SI3D* (2005), Lastra A., Olano M., Luebke D. P., Pfister H., (Eds.), ACM, pp. 163–170.

[Lin00] Lindstrom P.: Out-of-core simplification of large polygonal models. In *SIGGRAPH* (2000), pp. 259–262.

[Lin03] Lindstrom P.: Out-of-core construction and visualization of multiresolution surfaces. In *Proceedings of the 2003 symposium on Interactive 3D graphics* (2003), ACM, pp. 93–102.

[LK10] Laine S., Karras T.: Efficient sparse voxel octrees. In *SI3D* (2010), Aliaga D. G., Oliveira M. M., Varshney A., Wyman C., (Eds.), ACM, pp. 55–63.

[LL94] Lacroute P., Levoy M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH* (1994), ACM, pp. 451–458.

[LLY06] Ljung P., Lundström C., Ynnerman A.: Multiresolution interblock interpolation in direct volume rendering. In *EuroVis* (2006), Santos B. S., Ertl T., Joy K. I., (Eds.), Eurographics Association, pp. 259–266.

[LLYM04] Ljung P., Lundström C., Ynnerman A., Museth K.: Transfer function based adaptive decompression for volume rendering of large medical data sets. In *VolVis* (2004), IEEE Computer Society, pp. 25–32.

[LMK03] Li W., Mueller K., Kaufman A. E.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *IEEE Visualization* (2003), Turk G., van Wijk J. J., II R. J. M., (Eds.), IEEE Computer Society, pp. 317–324.

[LR98] Lischinski D., Rappoport A.: Image-based rendering for non-diffuse synthetic scenes. In *Rendering Techniques* (1998), Drettakis G., Max N. L., (Eds.), Springer, pp. 301–314.

[Luc77] Lucy L. B.: A numerical approach to the testing of the fission hypothesis. 1013–1024.

[LW90] Levoy M., Whitaker R.: Gaze-directed volume rendering. In *I3D* (1990), Zyda M., (Ed.), ACM, pp. 217–223.

[LYTM08] Lauterbach C., Yoon S.-e., Tang M., Manocha D.: Reducem: Interactive and memory efficient ray tracing of large models. In *Proceedings of the Nineteenth Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2008), EGSR '08, Eurographics Association, pp. 1313–1321.

[Max95] Max N. L.: Optical models for direct volume rendering. *IEEE Trans. Vis. Comput. Graph. 1*, 2 (1995), 99–108.

[MBW08] Mattausch O., Bittner J., Wimmer M.: Chc++: Coherent hierarchical culling revisited. *Comput. Graph. Forum 27*, 2 (2008), 221–230.

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2003), pp. 154–159.

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

[MN88] Mitchell D. P., Netravali A. N.: Reconstruction filters in computer-graphics. In *SIGGRAPH* (1988), Beach R. J., (Ed.), ACM, pp. 221–228.

[Mon92] Monaghan J. J.: Smoothed particle hydrodynamics. 543–574.

[Moo65] Moore G. E.: Cramming more components onto integrated circuits. *Electronics 38*, 8 (Apr. 1965).

[Mor66] Morton G.: *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[MSD07] Müller M., Schirm S., Duthaler S.: Screen space meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2007), pp. 9–15.

[Mus13] Museth K.: Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (2013), 27.

[ND12] Novák J., Dachsbacher C.: Rasterized bounding volume hierarchies. *Comput. Graph. Forum 31*, 2 (2012), 403–412.

[Ney98] Neyret F.: Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Trans. Vis. Comput. Graph. 4*, 1 (1998), 55–70.

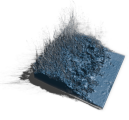
[NIL12] Nguyen C. V., Izadi S., Lovell D.: Modeling kinect sensor noise for improved 3D reconstruction and tracking. In *3DIMPVT* (2012), IEEE, pp. 524–530.

[NJB07] Navratil P., Johnson J., Bromm V.: Visualization of cosmological particle-based datasets. *IEEE Trans. Vis. Comput. Graph. 13*, 6 (2007), 1712–1718.

[NM06] Nielsen M. B., Museth K.: Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *J. Sci. Comput. 26*, 3 (2006), 261–299.

[NNSM07] Nielsen M. B., Nilsson O., Söderström A., Museth K.: Out-of-core and compressed level set methods. *ACM Trans. Graph. 26*, 4 (2007).

[NNT*05] Ng C.-M., Nguyen C.-T., Tran D.-N., Tan T.-S., Yeow S.-W.: Analyzing prefetching in large-scale visual simulation. In *Proc. Computer Graphics International (CGI)* (2005), pp. 100–107.



[Nov05] Novosad J.: Advanced high-quality filtering. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, pp. 417–435.

[NVI12] NVIDIA: Kepler GK110 whitepaper, 2012.

[NZIS13] Niessner M., Zollhöfer M., Izadi S., Stamminger M.: Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)* (2013).

[OBM00] Oliveira M. M., Bishop G., McAllister D. K.: Relief texture mapping. In *SIGGRAPH* (2000), pp. 359–368.

[OHB*13] Orthmann J., Hochstetter H., Bader J., Bayraktar S., Kolb A.: Consistent surface model for sph-based fluid transport. In *Symposium on Computer Animation* (2013), Chai J., Yu Y., Kim T., Sumner R. W., (Eds.), ACM, pp. 95–103.

[OKL06] Oh K., Ki H., Lee C.-H.: Pyramidal displacement mapping: A GPU based artifacts-free ray tracing through an image pyramid. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 2006), VRST '06, ACM, pp. 75–82.

[Pan11] Pantaleoni J.: Voxelpipe: A programmable pipeline for 3D voxelization. In *High Performance Graphics* (2011), Dachsbacher C., Mark W., Pantaleoni J., (Eds.), Eurographics Association, pp. 99–106.

[PBD*10] Parker S. G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D. P., McAllister D. K., McGuire M., Morley R. K., Robison A., Stich M.: Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (2010).

[PBFJ05] Porumbescu S. D., Budge B., Feng L., Joy K. I.: Shell maps. *ACM Trans. Graph. 24*, 3 (2005), 626–633.

[PC12] Peng C., Cao Y.: A GPU-based approach for massive model rendering with frame-to-frame coherence. *Comput. Graph. Forum 31*, 2 (2012), 393–402.

[PdON06] Policarpo F., de Oliveira Neto M. M.: Relief mapping of non-height-field surface details. In *SI3D* (2006), Olano M., SÃfÂ©quin C. H., (Eds.), ACM, pp. 55–62.

[Pea85] Peachey D. R.: Solid texturing of complex surfaces. In *SIGGRAPH* (1985), Cole P., Heilman R., Barsky B. A., (Eds.), ACM, pp. 279–286.

[PFHA10] Pantaleoni J., Fascione L., Hill M., Aila T.: Pantaray: fast ray-traced occlusion caching of massive scenes. *ACM Trans. Graph. 29*, 4 (2010).

[PGDS09] Popov S., Georgiev I., Dimov R., Slusallek P.: Object partitioning considered harmful: space subdivision for bvhs. In *High Performance Graphics* (2009), Spencer S. N., McAllister D. K., Pharr M., Wald I., Luebke D. P., Slusallek P., (Eds.), ACM, pp. 15–22.

[PH89] PERLIN K., HOFFERT E. M.: Hypertexture. In *SIGGRAPH* (1989), Thomas J. J., (Ed.), ACM, pp. 253–262.

[PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering, Second Edition: From Theory To Implementation*, 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[PM90] PERONA P., MALIK J.: Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12* (1990), 629–639.

[PM96] PROAKIS J. G., MANOLAKIS D. G.: *Digital Signal Processing (3rd Ed.): Principles, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D* (2005), Lastra A., Olano M., Luebke D. P., Pfister H., (Eds.), ACM, pp. 155–162.

[PV05] PHAM T. Q., VLIET L. J. V.: Separable bilateral filtering for fast video preprocessing. In *IEEE Intl. Conf. on Multimedia and Expo (ICME)* (2005), pp. 4pp.–.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M. H.: Surfels: surface elements as rendering primitives. In *SIGGRAPH* (2000), pp. 335–342.

[RB08] ROSENBERG I. D., BIRDWELL K.: Real-time particle isosurface extraction. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (2008), pp. 35–43.

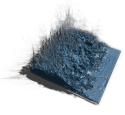
[RCBW12] REICHL F., CHAJDAS M. G., BÜRGER K., WESTERMANN R.: Hybrid sample-based surface rendering. In *Proceedings of Vision, Modeling and Visualization 2012* (2012), pp. 47–54.

[RCSW14] REICHL F., CHAJDAS M., SCHNEIDER J., WESTERMANN R.: Interactive rendering of giga-particle fluid simulations. In *Proceedings of High Performance Graphics 2014* (2014).

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH* (2000), pp. 343–352.

[ROF92] RUDIN L. I., OSHER S., FATEMI E.: Nonlinear total variation based noise removal algorithms. *Phys. D 60*, 1-4 (1992), 259–268.

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (New York, NY, USA, 2000), HWWS '00, ACM, pp. 109–118.



[RTW13] Reichl F., Treib M., Westermann R.: Visualization of big sph simulations via compressed octree grids. In *Proceedings of IEEE Big Data 2013* (2013), pp. 71–78.

[RUL00] Revelles J., UreÃa C., Lastra M.: An efficient parametric algorithm for octree traversal. In *WSCG* (2000).

[RWW] Reichl F., Weiss J., Westermann R.: Memory-efficient scene reconstruction from depth image streams. Submitted to Eurographics 2015.

[Sam84] Samet H.: The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2 (June 1984), 187–260. URL: `http://doi.acm.org/10.1145/356924.356930`.

[Sam06] Samet H.: *Foundations of multidimensional and metric data structures*. Elsevier [u.a.], Amsterdam [u.a.], 2006.

[Say12] Sayood K.: *Introduction to Data Compression*. Morgan Kaufmann series in multimedia information and systems. Morgan Kaufmann, 2012.

[SEE*12] Sturm J., Engelhard N., Endres F., Burgard W., Cremers D.: A benchmark for the evaluation of rgb-d SLAM systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)* (Oct. 2012).

[SG01] Shaffer E., Garland M.: Efficient adaptive simplification of massive meshes. In *IEEE Visualization* (2001), Ertl T., Joy K. I., Varshney A., (Eds.), IEEE Computer Society.

[SKKS12] Steinberger M., Kenzel M., Kainz B., Schmalstieg D.: Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *Innovative Parallel Computing (InPar), 2012* (2012), IEEE, pp. 1–10.

[SKU08] Szirmay-Kalos L., Umenhoffer T.: Displacement mapping on the GPU - state of the art. *Comput. Graph. Forum 27*, 6 (2008), 1567–1592.

[Spr05] Springel V.: The cosmological simulation code GADGET-2. *Mon. Not. Roy. Astron. Soc. 364* (2005), 1105.

[SS96] Schaufler G., Stürzlinger W.: A three dimensional image cache for virtual reality. *Comput. Graph. Forum 15*, 3 (1996), 227–236.

[SS10] Schwarz M., Seidel H.-P.: Fast parallel surface and solid voxelization on GPUs. *ACM Trans. Graph. 29*, 6 (2010), 179.

[SSP07] Solenthaler B., Schläfli J., Pajarola R.: A unified particle model for fluid-solid interactions: Research articles. *Comput. Animat. Virtual Worlds 18*, 1 (2007), 69–82.

[ST90] Saito T., Takahashi T.: Comprehensible rendering of 3-D shapes. In *SIGGRAPH* (1990), Baskett F., (Ed.), ACM, pp. 197–206.

[Swe98] Sweldens W.: The lifting scheme: A construction of second generation wavelets. *SIAM J. Math. Anal. 29*, 2 (Mar. 1998), 511–546.

[SWJ*05] Springel V., White S. D. M., Jenkins A., Frenk C. S., Yoshida N., Gao L., Navarro J., Thacker R., Croton D., Helly J., Peacock J. A., Cole S., Thomas P., Couchman H., Evrard A., Colberg J., Pearce F.: Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature 435* (2005), 629–636.

[SZL92] Schroeder W. J., Zarge J. A., Lorensen W. E.: Decimation of triangle meshes. In *SIGGRAPH* (1992), Thomas J. J., (Ed.), ACM, pp. 65–70.

[TBR*12] Treib M., Bürger K., Reichl F., Meneveau C., Szalay A. S., Westermann R.: Turbulence visualization at the terascale on desktop PCs. *IEEE Trans. Vis. Comput. Graph. 18*, 12 (2012), 2169–2177.

[TIS08] Tevs A., Ihrke I., Seidel H.-P.: Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *SI3D* (2008), Haines E., McGuire M., (Eds.), ACM, pp. 183–190.

[TM98] Tomasi C., Manduchi R.: Bilateral filtering for gray and color images. In *Proc. Intl. Conf. on Computer Vision* (1998), IEEE, pp. 839–846.

[TRAW12] Treib M., Reichl F., Auer S., Westermann R.: Interactive editing of gigasample terrain fields. *Comput. Graph. Forum 31*, 2 (2012), 383–392.

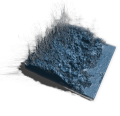
[Tre14] Treib M.: *GPU-Based Compression for Large-Scale Visualization*. PhD thesis, Technische Universität München, 2014.

[TS91] Teller S. J., Séquin C. H.: Visibility preprocessing for interactive walkthroughs. In *SIGGRAPH* (1991), Thomas J. J., (Ed.), ACM, pp. 61–70.

[vdLGS09] van der Laan W. J., Green S., Sainz M.: Screen space fluid rendering with curvature flow. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 91–98.

[VH14] Vinkler M., Havran V.: Register efficient memory allocator for gpus. In *High-Performance Graphics 2014* (Lyon, France, 2014), Wald I., Ragan-Kelley J., (Eds.), Eurographics Association, pp. 19–27.

[WDS05] Wald I., Dietrich A., Slusallek P.: An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.



[WE98] Westermann R., Ertl T.: Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH* (1998), pp. 169–177.

[Wes90] Westover L.: Footprint evaluation for volume rendering. In *SIGGRAPH* (1990), Baskett F., (Ed.), ACM, pp. 367–376.

[Whi80] Whitted T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

[Wik] Wikipedia:. List of Nvidia Graphics Processing Units. Date accessed: 10/19/2014. URL: `http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#Comparison_tables:_Desktop_GPUs`.

[WIK*06] Wald I., Ize T., Kensler A., Knoll A., Parker S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics* (2006), 485–493.

[Wil83] Williams L.: Pyramidal parametrics. *SIGGRAPH Comput. Graph. 17*, 3 (July 1983), 1–11.

[WMG*09] Wald I., Mark W. R., Günther J., Boulos S., Ize T., Hunt W. A., Parker S. G., Shirley P.: State of the art in ray tracing animated scenes. *Comput. Graph. Forum 28*, 6 (2009), 1691–1722.

[Wol90] Wolberg G.: *Digital image warping.* IEEE, 1990.

[WWH*00] Weiler M., Westermann R., Hansen C. D., Zimmerman K., Ertl T.: Level-of-detail volume rendering via 3D textures. In *Volviz* (2000), pp. 7–13.

[WWLG09] Weise T., Wismer T., Leibe B., Gool L. V.: In-hand scanning with online loop closure. In *IEEE International Workshop on 3-D Digital Imaging and Modeling* (2009).

[WWWG13] Widmer S., Wodniok D., Weber N., Goesele M.: Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (New York, NY, USA, 2013), GPGPU-6, ACM, pp. 120–126. URL: `http://doi.acm.org/10.1145/2458523.2458535`.

[XV96] Xia J. C., Varshney A.: Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization* (1996), pp. 327–334.

[YHK09] Yasuda R., Harada T., Kawaguchi Y.: Fast rendering of particle-based fluid by utilizing simulation data. In *Proceedings of Eurographics 2009* (2009), pp. 61–64.

[YLM06] Yoon S.-E., Lauterbach C., Manocha D.: R-lods: fast lod-based ray tracing of massive models. *The Visual Computer 22*, 9-11 (2006), 772–784.

[YSGM04] Yoon S.-E., Salomon B., Gayle R., Manocha D.: Quick-vdr: Interactive view-dependent rendering of massive models. In *IEEE Visualization* (2004), IEEE Computer Society, pp. 131–138.

[YT13] Yu J., Turk G.: Reconstructing surfaces of particle-based fluids using anisotropic kernels. *ACM Trans. Graph. 32*, 1 (2013), 5:1–5:12.

[YWTY12] Yu J., Wojtan C., Turk G., Yap C.: Explicit mesh surfaces for particle based fluids. *Comput. Graph. Forum 31*, 2 (2012), 815–824.

[ZB05] Zhu Y., Bridson R.: Animating sand as a fluid. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (2005), pp. 965–972.

[ZEP07] Zhang L., 0001 W. C., Ebert D. S., Peng Q.: Conservative voxelization. *The Visual Computer 23*, 9-11 (2007), 783–792.

[ZOC10] Zafar F., Olano M., Curtis A.: GPU random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 133–141.

[ZSP08] Zhang Y., Solenthaler B., Pajarola R.: Adaptive sampling and rendering of fluids on the gpu. In *Symposium on Point-Based Graphics* (2008), pp. 137–146.