Efficient GPU Usage for Rendering of Volume Data

by

David Ganter, B.A. MCS

Ph.D Thesis

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Doctor of Philosophy in Computer Science

University of Dublin, Trinity College

August 2020

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

David Ganter

August 27, 2020

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

David Ganter

August 27, 2020

Acknowledgments

I would like to begin by thank my supervisor, Michael Manzke, for all his guidance throughout this Ph.D. Without his invaluable input, feedback, and direction this may never have been finished. To my parents, Martin and Eileen who have given me all the support that I could ever have needed, and to whom I will forever be grateful. To all those in GV2 whose insights, talks, and discussions have been invaluable. To Emma, who has has encouraged my growing caffeine addiction whilst simultaneously hearing about all the failed ideas and experiments. Deepest thanks to Mairéad, who warned me in advance of the trials and tribulations of Ph.Ds, but still put up with all the vents and rants regardless, and has always been excellent lunch company. To Ian, Dave, Matt, Mel, and Ray, who provided five kilos of chocolate to help me through my final months, and provided all the "Are you not finished that Ph.D yet?" motivation a friend could ever ask for. Finally, and most importantly, to Bróna, without whom I could never have made this journey, who has been utterly supportive in everything I have done regardless of outcome, who has been through every single high point with me and pulled me out of every low. To you, I cannot ever show enough gratitude for everything that you have done for me over the last 5 years and more.

David Ganter

University of Dublin, Trinity College
August 2020

Efficient GPU Usage for Rendering of Volume Data

Publication No. _____

David Ganter, Ph.D
University of Dublin, Trinity College, 2020

Supervisor: Dr. Michael Manzke

Visualising medical images or scientific data that can be shown in 3D poses some interesting challenges. This thesis investigates two distinct areas of Direct Volume Rendering (DVR); time-varying datasets on emerging light-field displays, and acceleration data-structures for large, static datasets. Firstly, a novel method to efficiently display volume data that changes over time (time-varying) on emerging display technologies is shown to improve upon traditional techniques in the field. The second main focus is acceleration data-structures in the context of volume rendering. The attention is on how particular acceleration data-structures are used in current DVR approaches and why Bounding Volume Hierarchies (BVHs) haven't been the method of choice for large-scale volume rendering. This thesis provides an insightful look at how new hardware implementations in consumer products impact on DVR, and how this may

change the adoption of BVHs in this area. Results show that BVHs coupled with a pre-clustering step are a viable alternative to standard octree Empty-Space-Skipping (ESS) techniques.

Contents

List of Tables

List of Figures

Acronyms

AABB Axis-Aligned Bounding Box. xv, 84, 91, 93, 99, 107, 108, 110

AMR Adaptive Mesh Refinement. xv

AR Augmented Reality. xv, 3

BVH Bounding Volume Hierarchy. vi, vii, xv, 1, 5, 6, 8, 26, 63, 91–96, 98–100, 102, 103, 107–113

CFD Computational Fluid Dynamics. xv, 14, 21, 22

CGI Computer Generated Imagary. xv, 14

CNN Convolutional Neural Network. xv, 48

CPU Central Processing Unit. xv, 1, 2, 4, 23, 24

DVR Direct Volume Rendering. vi, xiv, xv, 1, 2, 4–6, 8, 9, 12, 14, 15, 17, 18, 21–23, 26, 37, 44, 46, 63, 67, 91–96, 98–100, 102, 103, 107, 109, 111, 112, 118, 121

ERT Early-Ray-Termination. xv, 19, 75, 103, 109, 111, 120

ESS Empty-Space-Skipping. vii, xv, 1, 5, 26, 30, 33, 67, 79, 84, 91, 93, 96, 97, 100, 109, 110, 112, 123

FPGA Field-Programmable Gate Array. xv, 38

GPU Graphics Processing Unit. xv, 1, 2, 4, 5, 23, 24

H3DDDA Hierarchical 3D Differential Analyser. xv, 33

HMD Head-Mounted Display. xv, 3, 38, 40, 41, 44

HVS Human Visual System. xv, 41, 42

IBR Image-Based Rendering. xv, 48

LOD Level-of-Detail. xv, 17, 93

SAH Surface-Area Heuristic. xv, 26

SMM Streaming Multi-Processor. xv, 24

VAC Vergence-Accommodation Conflict. xv, 42, 43

VR Virtual Reality. xv, 3

Chapter 1

Introduction

Visualising medical images or scientific data that can be shown in 3D provides some interesting challenges. This main two works of this thesis investigate two distinct areas of DVR; time-varying datasets on emerging light-field displays, and acceleration data-structures for large, static datasets. Firstly, a novel method to efficiently display volume data that changes over time (time-varying) on emerging display technologies is shown. The results show that this method improves upon traditional techniques in the field [1, 2, 3]. The second major area this thesis investigates is acceleration data-structures in the ever-expanding large static datasets domain, with focus on single computer systems. The attention is on how particular acceleration data-structures are used in current DVR approaches and why specifically BVHs haven't been the method of choice for large-scale volume rendering. Since the focus is on simple Central Processing Unit (CPU) and Graphics Processing Unit (GPU) systems rather than clusters, this thesis therefore also provides an insightful look at how new hardware implementations in commodity products impacts on DVR, and how this may change the adoption of BVHs in this area. It further shows that BVHs are a viable alternative to standard octree ESS techniques [4, 5, 6], especially when coupled with a pre-clustering step.

Although these are the main aspects of this thesis, this work also includes vital investigations that are fundamental principles upon which the main topics build upon. A more in-depth explanation of the core concepts to further outline gaps in the literature will be presented in the following chapter. For now, the remainder of this chapter briefly introduces each of the topics of work that this thesis presents in the hope that

the reader is given an understanding of the justification to these individual modules.

1.1 Investigating the GPU Memory Hierarchy

The entirety of work in this thesis focuses on improving rendering performance for volumetric data on commodity CPU and GPU systems. To effectively utilise this underlying platform the thesis begins by examining the effect of regular-grid, structured data on the GPU memory hierarchy. This data is representative of the types of scenarios presented when rendering volume datasets. Benchmarking this is important to give an understanding of how the GPU reacts to different volumetric data types and sizes. Furthermore, it informs decisions on the ever important sampling patterns exhibited by DVR.

The results show that the opaque storage — storage for which the exact layout is undocumented for commercial reasons — for 3D textures is not quite as simple or straight-forward as one might imagine. There are clear indications of transposed memory to suit spatio-temporal coherence in 3D texture sampling. These are fundamental principles which are then exploited in following work. Observations are also made that deviation from these principles can result in substantial performance penalties. This retroactively reinforced the requirement for this investigation.

1.2 View Dependent Scheduling for DVR

The findings of the GPU memory hierarchy investigations are then utilised to formulate a method to accelerate the rendering of dense grid volumetric data by scheduling appropriate regions of the grid to be utilised efficiently. When studying the results of 3D texture sampling, it becomes clear that efficiently sampling the L2-cache should be a priority for DVR. Thus the shared L2-cache on the GPU was targeted as a source of coherent memory access when sampling the volume. Scheduling L2-cache-sized regions of volume data to be rendered therefore seems like an obvious optimization that could be made. The results — while both positive and negative — were enlightening. Memory performance statistics were considered to be quite good, however the scheduling unfortunately comes at a considerable cost. This meant that the overall render per-

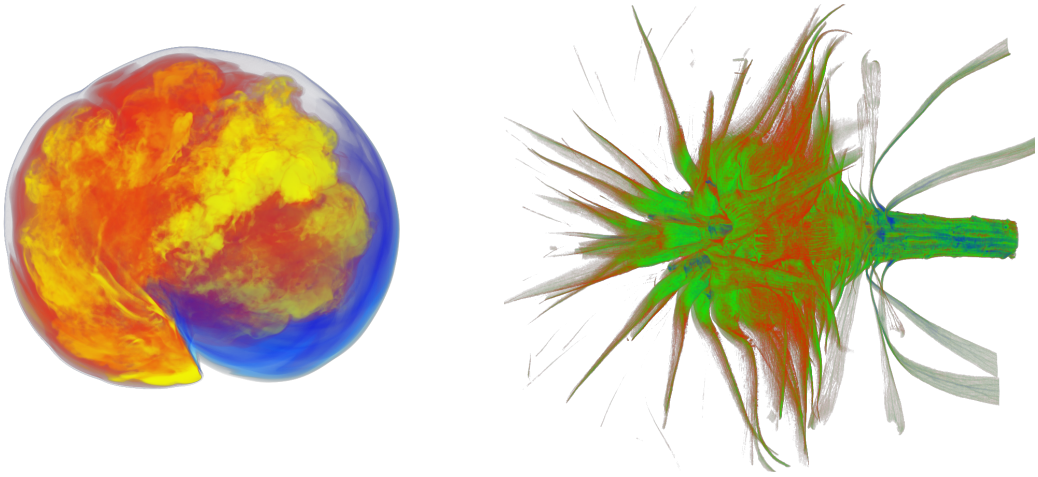


Figure 1.1: Examples of volume rendering in different domains.

formance of this method were not satisfactory when applied in this scenario. These results and drawbacks expanded upon in chapter 4. This particular work should be more seen as groundwork for following evaluations. Since the memory performance was improved, these methods were then extended to multi-view rendering which exhibits a more random access pattern.

1.3 Light-field DVR for Time-Varying Volumes on GPU

Most volume rendering techniques have been focused on single-view, 2D displays but other emerging technologies should also be considered. For example 3D displays and Virtual Reality (VR) / Augmented Reality (AR) Head-Mounted Displays (HMDs) have had huge advances in recent years, becoming more high-fidelity or more responsive in the case of HMDs. There are of course many advantages to come from these displays. For example, when using auto-stereoscopic or light-field displays [7, 8, 9, 10], the additional views can provide users with considerably more perceptual information from parallax [11, 12]. Recent emerging light-field display technologies — like the Looking Glass [7] as shown in figure 1.3 — are making these kind of displays more consumer

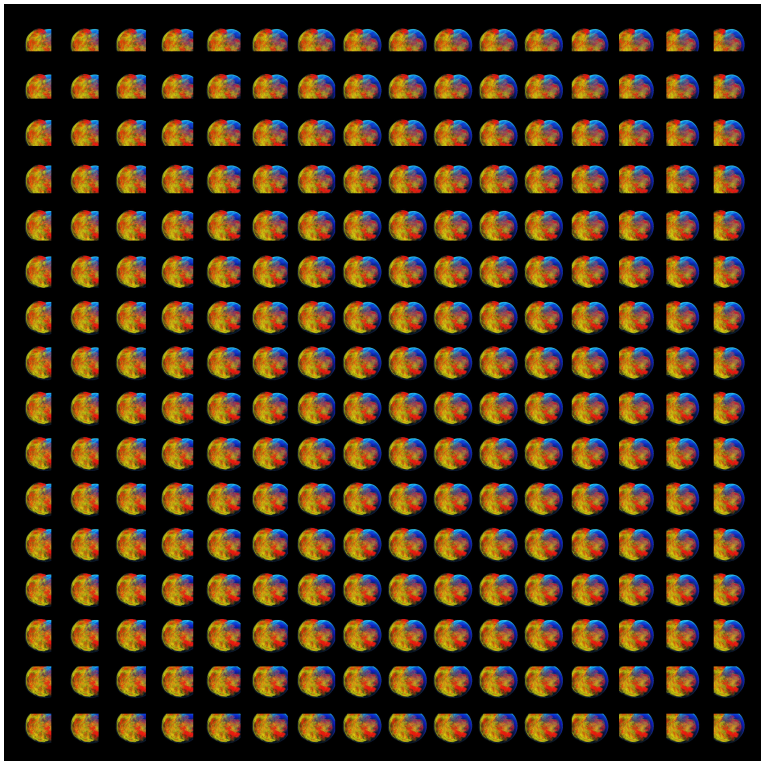


Figure 1.2: Example of a light-field volume render using a 16^2 angular resolution of the supernova dataset.

ready. Since a simple discretised light-field can be represented as a 2D planar grid of camera angles, it can then be considered as a multi-view render target.

These displays generally need to be provided with multiple views at some stage in the pipeline. These views of the same scene are offset from each other in specific ways effectively generating a discretised light-field. For DVR this means rendering the volume not only twice but many times from unique view points, requiring the volume to be traversed in a potentially wasteful manner with regards to the physical memory hierarchy of the CPU or GPU, or indeed both. Unfortunately, the current literature in this area — especially for using relatively low-cost consumer hardware and setups — are extremely lacking.

A core theme of this thesis focuses on the efficient scheduling on time-varying volume data with the intent of targeting on light-field displays. Following on from the previous evaluations, this is achieved by sub-dividing the volume, determining the 'active' regions with respect to the transfer function and creating an over-arching render

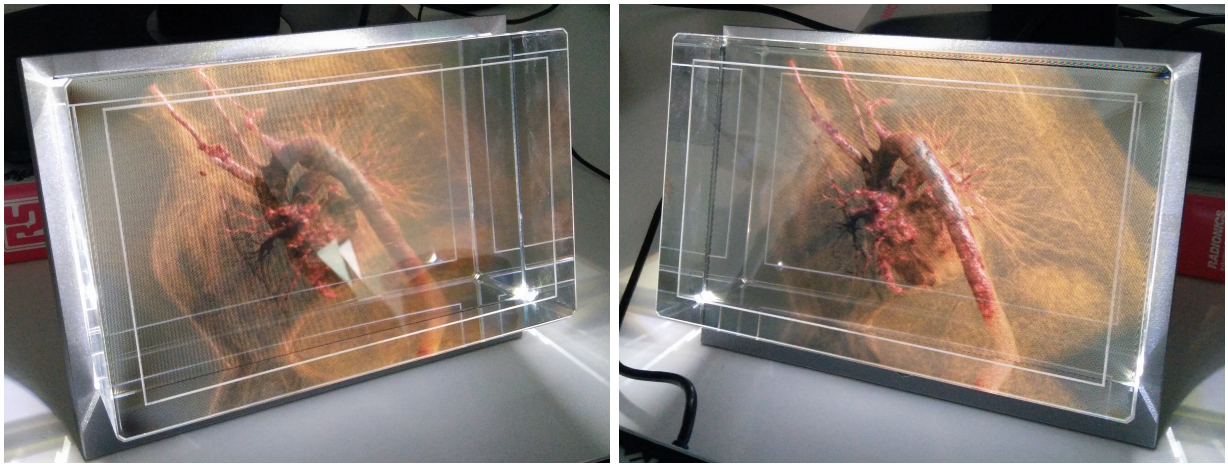


Figure 1.3: Image of the LookingGlass [7], a horizontal light-field display aimed at consumers. Parallax that is clearly visible between the left and right images, as a user moves their head through the horizontal range of projection.

order list. Since light-field synthesis is effectively multi-view rendering, these volume sub-divisions must be rendered to all views. The render order of sub-divisions may not necessarily be the same for all views.

The key result of this work is a method to accommodate the partial out-of-order rendering required to schedule sub-divisions efficiently on the GPU. This method demonstrates significant improvements over traditional time-varying DVR techniques which were not designed for multi-view rendering are shown. An in-depth explanation of this work and its results are presented in chapter 5.

1.4 Acceleration Data-structures and DVR

As a reminder, the overall theme for this thesis is performance of DVR on commodity hardware. It then goes without saying that when a new piece of technology gets implemented into a new line of consumer available graphics cards — and when this technology could possibly be beneficial to DVR — an evaluation must then be performed. ESS for DVR has been a topic of much research. Regular spatially sub-dividing methods like octrees are a common choice for GPU as they can be easily constructed and traversed, and can support trivially sub-sampled regions of volume that do not need to be densely sampled due to its content or distance from the camera. In contrast BVHs

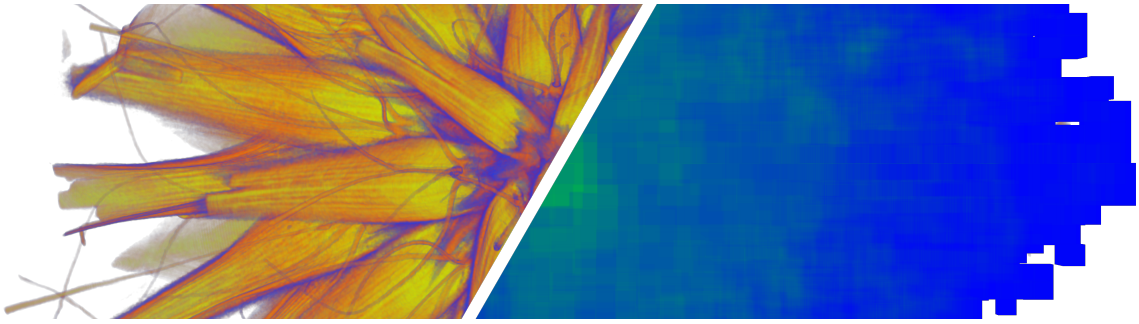


Figure 1.4: Cut-away example showing the heat-map of the leaf-node complexity for a particular data-set with a particular transfer function.

are slightly more complex to traverse and require extra work to support sub-sampled regions. However, BVHs have been the topic of much research in the graphics industry due to their effectiveness as a spatial acceleration data-structure for surface geometry, leading to their prevalence in surface path-tracing. As such, BVH logic has begun to be integrated into graphics hardware, beginning with Nvidia's RTX [13] line of GPUs. Therefore, a re-invigorated evaluation of BVHs in the context of DVR is required. The results show that BVHs are indeed a viable approach to interactive volume exploration with real-time transfer function editing.

One major difference between building acceleration data-structures for surface represented models like meshes and structured volumetric data is the regularity of leaf-nodes. To be more concise, a BVH suits polygon data due to the potentially arbitrary positioning of polygons in space. In contrast, sub-dividing a volume results in many leaf nodes which are adjacent to each other. It then follows that — if a large group of leaf nodes are present and determined to be active by the current transfer function — clustering groups of leaf nodes can be beneficial to BVH DVR by reducing tree and leaf-node complexity. As such, a method is presented to further increase the render performance of volumes that exhibit highly cluster-able regions of sub-division leaves. It is shown that cluster leaf-node complexity can be reduced to anywhere between 60% and 5%, although this massively depends on the particular volume and transfer function. The results of these evaluations are shown in chapter 6.

1.5 Research Questions

The scientific, medical and industrial uses of volume rendering have been previously described. It should be clear that, certainly in some of these applications, time-varying or real-time volumetric data is required to be rendered. It should also be understood that the render targets include auto-stereoscopic, multi-view or light-field displays for their advantages in perceptual information and visual comfort. Combining all of these factors creates a problem that is quite computationally complex and exhaustive on memory systems. It is clear that the memory hierarchy plays a major role in volume rendering, both in terms of source-to-host stages and GPU memory hierarchy. With this in mind, the initial research question of this thesis is as follows:

What ways can view-dependent information be exploited to efficiently schedule volume data for rendering?

In this work, this research question is applied not only to single-view volume rendering, but for multi-view volume rendering in the form of light-fields, where view-dependent scheduling is shown to have a major impact on rendering performance.

As with all long-term projects, over the course of a piece of research technology changes and new features are brought into the consumer domain. One of the most significant changes that emerged during this thesis was fixed-function ray-tracing hardware in consumer GPUs, specifically targeting BVHs. BVHs had been experimented with on CPUs for volume rendering, but the literature for using these data-structures on GPU was virtually non-existent. Therefore, it seemed logical to add another research question during the course of this work:

Are Bounding Volume Hierarchies a viable alternative to other acceleration data-structures to use with empty-space-skipping for volume rendering?

1.6 Contributions

The contributions of this thesis can be summarised as follows: Evaluations of the GPU memory hierarchy with special attention to texture accesses in the context of 3D textures is presented. These evaluations shed some light on the underlying memory organisation used in the used GPU architectures and highlight the importance of adhering to optimised L2 cache coherency for volume rendering. Using this knowledge, a data-scheduling method is devised for single-view volume rendering that aims to make use of the fact that both the CPU and GPU can run in parallel and as such can be pipelined with the CPU doing the scheduling work. The evaluation of this approach showed sub-optimal results, showing a burden of overhead due to scheduling information on the GPU. However, taking this work and applying it to multiple-views could amortize the cost of this overhead. Therefore a method that optimises data-scheduling in DVR for light-field displays is presented and evaluated, showing considerable improvement over prior techniques.

Finally, during the course of this work new ray-tracing hardware became integrated into consumer GPUs. This hardware allowed for accelerated traversal of BVHs, a data-structure not commonly used in volume rendering. Therefore, an evaluation of GPU BVH out-of-core DVR in comparison to current popular methods such as octrees was warranted, with emphasis on render performance for static data volumes and dynamic transfer-function updates. The results show that BVHs are indeed a viable candidate for GPU based DVR, especially when clustering leaf nodes using a clustering method.

1.7 Summary of Chapters

This chapter should hopefully have given the reader a basic understanding of the importance of DVR — especially on consumer hardware – and the potential of light-field technologies and their position relative to DVR. The core theme of this thesis is maximising the efficiency of GPUs when rendering volumes. At this point the main contributions should by now be evident but, of course, the justifications, experiments, and the results need to be presented. The rest of this thesis is set out as follows:

- **Chapter 2 (Indirectly Related)** gives a detailed explanation of the back-

ground for this thesis, and a more in-depth analysis of related work with the aim of pointing out areas in current research that needed to be addressed, justifying the work in this thesis.

- **Chapter 3 (BVH Adoption and Clustering for GPU DVR)** evaluates the memory hierarchy performance of the GPU and provides a platform upon which much of this thesis' work stands.

- **Chapter 4 (Conclusions)** presents an investigation into a view-dependent approach for scheduling volume regions for efficient use of the GPU memory hierarchy.

- **Chapter 5 (Conclusion)** continues on from the work done in chapter 4 and extends it to light-fields — the multiple view-point rendering of volumes in a single pass.

- **Chapter 6 (Conclusion)** takes a detour from focusing on the memory hierarchy as an optimisation target and instead looks at using new GPU hardware for acceleration data structures to speed up empty-space-skipping during DVR.

- **Chapter 7 (Conclusions & Future Work)** finally summarises the main body of this thesis and draws conclusions from the results. The work in this thesis is by no means the stopping point for DVR research, and as such potential future work paths are outlined in this chapter.

The chapters in this thesis are using these key areas to build platforms upon which consecutive chapters can build. The author feels that structuring the work in this packaged way allows the reader to understand the flow of research over the duration of this thesis. In addition to these chapters, the following appendices are provided to give additional background and detail omitted from the main work for sake of brevity and being concise:

- **Appendix A (Summary)** As part of investigatory work in this thesis, a tool was made to visualise the sampling characteristics of DVR. Being able to visualise this information provided valuable information when formulating work in the data-structure domain. This appendix provides information about this tool.

- **Appendix B (Summary)** provides information about the datasets used and from where they were obtained.

1.8 Publications

This thesis resulted in publications that are directly related to the research performed, but also produced methods and algorithms that were used in additional indirectly related works.

Directly Related

The following is a list of publications that were produced by the author during the course of this thesis.

- **Ganter D.**, Alain M., Hardman D., Smolic A., Manzke M.: Light-Field DVR on GPU for Streaming Time-Varying Data. In *Pacific Graphics 2018 Short Papers* (Hong Kong, 2018), The Eurographics Association [14]

- **Ganter D.**, Manzke M.: An Analysis of Region Clustered BVH Volume Rendering on GPU. *Proceedings of High Performance Graphics 2019* (Strasbourg, 2019) ,The Eurographics Association, and published in *Computer Graphics Forum* special issue [15, 16]

Indirectly Related

The indirectly related publications were produced with elements from this thesis' research — specifically, the work performed in chapter 5 (Conclusion) was then used as a platform for rendering datasets used in the following publications.

- Bruton S., **Ganter D.**, Manzke M.: Fast Approximate Light Field Volume Rendering: Using Volume Data to Improve Light Field Synthesis via Convolutional Neural Networks. Pending approval in *Springer Book of IVAPP 2019*, Springer

- Martin S., Bruton S., **Ganter D.**, Manzke M.: Synthesising Light Field Volume Visualisations Using Image Warping in Real-Time. Pending approval in *Springer Book of GRAPP 2019*, Springer

- Bruton S., **Ganter D.**, Manzke M.: Synthesising Light Field Volumetric Visualizations in Real-time Using a Compressed Volume Representation. In *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP* (Prague, 2019), INSTICC [17]

- Martin S., Bruton S., **Ganter D.**, Manzke M.: Using a Depth Heuristic for Light Field Volume Rendering. In *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 1: GRAPP* (Prague, 2019), INSTICC [18]

Chapter 2

Background & Related Work

In this chapter the necessary background is presented on concepts used in this thesis. This should give the reader a broad comprehension on the underlying fundamentals which are used not only by this work, but by related works. The cornerstone topics are DVR, GPUs, acceleration data-structures and light-fields. A brief introduction is given to each of these areas. This background contains an in-depth survey of the work relating to this thesis. This is followed by a summary of the aforementioned background work, and the motivations and rationales for chapters in this thesis.

2.1 Volume Rendering

DVR is a computer graphics method that can be used for visualising three dimensional regular grids of scalar data with applications ranging from scientific simulation, games, and medical analysis. Anything that can calculated, simulated or sampled which can be discretised and represented by a scalar value can be stored and displayed using volumetric data.

In the medical domain, for example, 3D-MRI machines can capture the density of a region of space and transform this into data that can be stored as a 3D grid of numbers. These density values coupled with a function to transform data into colour — known as a transfer function — can then be displayed to a medical practitioner or patient.

Scientific research also has provided an important application for volume ren-

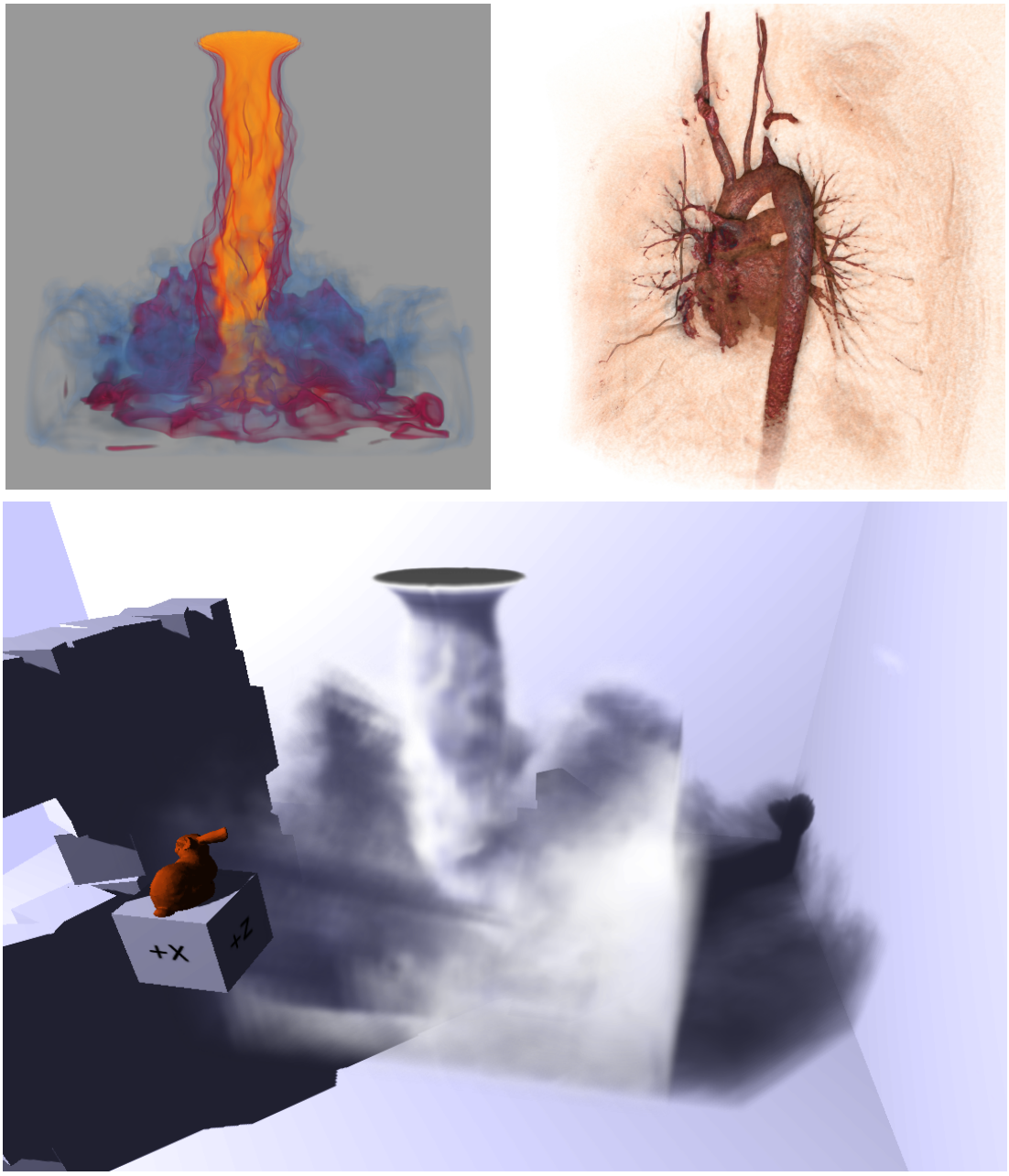


Figure 2.1: Examples of volume rendering applications. Top-left: Fluid simulation of smoke being introduced into a volume. Top-right: 3D MRI of a subclavia. Bottom: The same dataset shown in the top-left but applied to a mixed-media scene using global illumination (shadows).

dering. Computational Fluid Dynamics (CFD) simulations allow researchers and engineers to understand the complexity of fluid mechanics in a myriad of scenarios, quite a large proportion of which are discretised as volumetric data at some point. Due to the intricate nature of these simulations, the resulting data can theoretically be massive dimensionally, or perhaps contain multiple channels of data representing different phenomena.

The entertainment industry too make use of volumetric data. Examples of this can be seen in almost any Computer Generated Imagary (CGI) scene that contains clouds, smoke, or liquids. Although surface approximations are inherently smaller and potentially faster to render, volumetric representations can give a much higher degree of realism; for example the forward and backward scattering of light through clouds, and the varying intensity of shadows they create. This application — while it falls under the umbrella of volume rendering — is more focused on the accurate simulation of multiple scattering light transport. As interesting and complex as this topic is, it is not the main focus of this thesis, although it is worth noting that this field must combat its own set of problems in addition to those of standard DVR.

Examples of all the above applications can be seen in figure 2.1. These only represent a handful of the almost endless possibilities for volumetric data and its rendering. It is important to note that — while these applications all share the fundamental theoretical concept of a regular grid of scalar data — the true implementations of data storage and methods of rendering can vary substantially, and this often results in an inverse relationship between the level of interaction achievable to the complexity of the rendering — be that in terms of data size or illumination for example.

2.1.1 Volume Light-Transport Integral

Like the basis for almost all computer graphics techniques, DVR [19] describes the interaction and propagation of light through the volume. Rendering this data can be quite costly in terms of bandwidth, memory footprint and computational resources, especially when the volumetric data is extended into the time domain for time-varying data-sets on disk or real-time streaming data.

While multiple methods for volume rendering exists, they all share the same fundamental principle of accumulation of light. As light is propagated through the

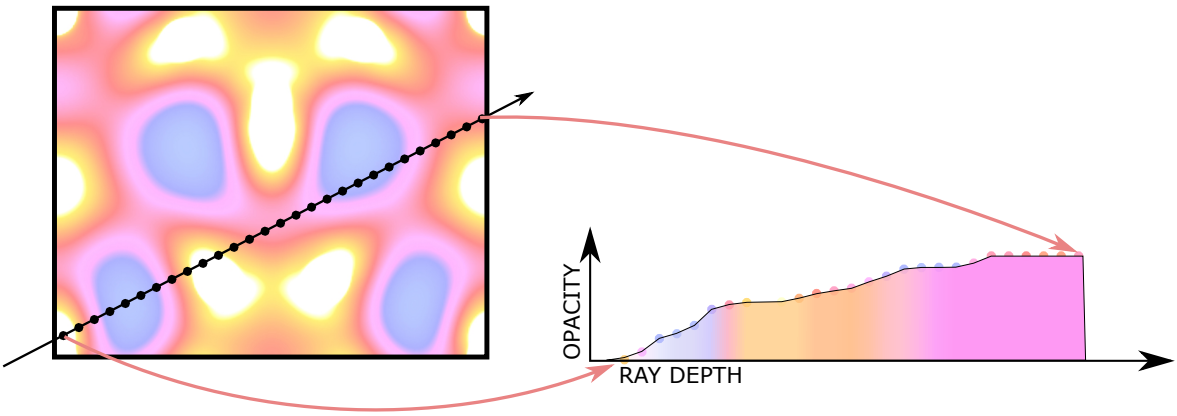


Figure 2.2: Accumulation of colour and opacity while sampling a ray through a volume. Notice that deeper into the ray, the accumulated colour isn't as effected by the sample colours because the accumulated opacity scales the contribution of the colour compositing.

volume towards the eye or the camera, colour and opacity are accumulated. This can be formally represented by the volume rendering integral listed in equation 2.1.

$$I(D_n) = I(D_{n-1})T(D_{n-1}, D_n) + \int_{D_{n-1}}^{D_n} C(s)T(s, D_n)\, ds \tag{2.1}$$

$$T(p_1, p_2) = e^{-\int_{p_1}^{p_2} \alpha(p)\, dp} \tag{2.2}$$

where $\alpha(p)$ represents the absorption coefficient and $C(p)$ represents the colour coefficient. This equation represents the irradiance at a point D_n, integrating along the ray. In equations 2.1 and 2.2, the functions $C(p)$ and $\alpha(p)$ represent the transformation of a point in volume space p to colour and opacity respectively. These are jointly known as the transfer function.

2.1.2 Volume Ray-Casting

The potentially most easy to understand and achieve implementation of volume rendering is volume ray-casting. Throughout this work, when the term DVR is used, it always refers to ray-casting through a volume unless otherwise specified. Looking back at the volume rendering integral in 2.1, it is important to note that this is presented

as a continuous function, however in practice the ray's are sampled at regular intervals. The discretised version of the volume rendering integral can be represented as a Riemann sum shown in equation 2.3.

$$I(D_n) = I(D_{n-1})T(D_{n-1}, D_n) + \sum_{D_{n-1}}^{D_n} C(s)T(s, D_n)\Delta s \tag{2.3}$$

In practice, a ray is sampled at regular intervals. In general, the sampling rate is determined by the voxel spacing. Specifically a sampling rate of half the voxel spacing prevents the aliasing of data along the ray. In the Riemann sum, this spacing is used to determine that Δ_s. To satisfy the Nyquist theorem, the sample rate could be set to twice the density of the voxel grid. The value of these samples is a function of the position in the volume, and different methods can be used. The most common method uses trilinear interpolation, where eight neighbouring voxels are interpolated to give a single scalar value. This scalar value is then transformed to a colour and opacity using a transfer function. This is visualised in figure 2.2. Much like surface geometry alpha blending [20], these samples are composited in either front-to-back or back-to-front order. The compositing methods are outlined in equation 2.4 for compositing the color value and equation 2.5 for alpha.

$$C_{dst} \leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \tag{2.4}$$

$$\alpha_{dst} \leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \tag{2.5}$$

Equation 2.1 is presented as a continuous function, however the volume data used in this thesis is generally not presented in such a form. Most of these data are discrete 3D grids of scalar values, generally with a fixed element data type. The resolution of these discrete volumes varies greatly ranging anywhere from single digit dimensions to grids potentially greater than 1024^3. The data type of the volume also plays an integral part of rendering performance. For example, if each data point is represented by a 64-bit floating point, the resolution allowed for transfer function manipulation and the range of values represented is highly flexible, however the space required to store the volume and the bandwidth needed to transfer the data may be prohibitive. On the other hand, quantizing data to something small like a byte or a char drastically

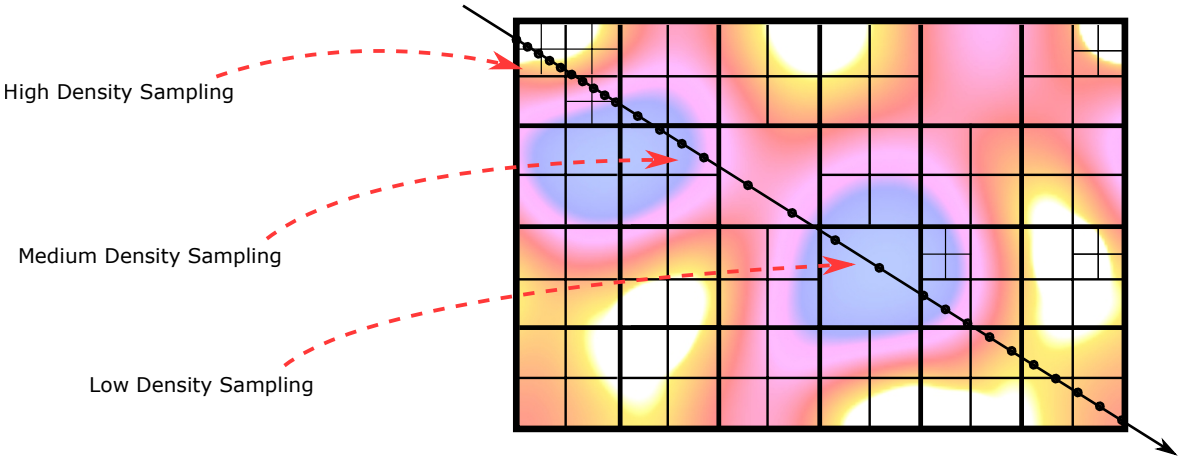


Figure 2.3: Example of adaptive sampling in a volume that has varying degrees of voxel density. As the ray traverses through denser regions of the volume, the sample rate is increased appropriately. The opposite is also true, where the sample rate is decreased in less-dense regions, saving on memory lookups and accelerating rendering.

reduces the memory footprint of the volume, however the range of values visualisable reduces accordingly.

It's worth noting that up to this point, all of this discussion has been assuming a constant voxel density across the entire volume. This is not always the case, and is one of the major first steps in DVR optimisation. A prime example of this is almost directly analogous to Level-of-Detail (LOD) meshes in polygonal rendering. A LOD mesh is a lower fidelity proxy of a full-scale potentially high poly count model. These are generally used when rendering objects further away in the scene. This concept can also be applied to volume rendering, where the volume may have several down-sampled resolutions, and the appropriate resolution is selected depending on how far the sample is from the camera plane, and the sample rate can be adjusted. The same approach can be used for regions of the volume that contain low-frequency data that don't need to be sampled as regularly, regardless of proximity to the camera. These methods are known as adaptive sampling and are a common optimisation [21, 4, 2, 5] for DVR. A visual description of adaptive sampling can be seen in figure 2.3.

Another analogy to this method is mipmapping for textures. Mipmapping also tackles another issue in rendering to a discretised 2D plane — aliasing. Aliasing occurs when textures are sampled at lower frequencies than the data it holds. The same is true

for volume data. When two rays from a perspective projection begin to diverge through the volume, the space between the rays is not sampled, which means important data may be missing from the final image. By using a down-sampled volume representation for further regions of the volume, this aliasing effect can be reduced and important data can be presented to the user [21].

2.1.3 Object Order & Image Order

An important feature of the volume rendering integral to note is the potential to reduced into different ray segments. This allows us to perform out-of-order rendering — specifically, this means integrating over several segments of the ray in parallel and accumulating the result when finished. There is, however, a major caveat; these ray segments must still be composited in the correct order. It then follows that the volume may be considered as smaller sub-volumes, or bricks, and may be rendered individually and composited later [22, 23, 24].

This can have significant advantages and outright suits a specific sub-set of volume rendering; distributed DVR. When displaying massive datasets, it's quite often efficient to distribute the workload by dividing the volume into multiple 'bricks' and load-balancing the rendering of these bricks across a cluster. Each node in the cluster maintains its own version of the frame buffer, and using potentially advanced techniques — expanded upon later — the individual frame buffers are composited in order to reach a final, correct image.

On the smaller scale, it can also be advantageous to sub-divide and render bricks in parallel, even if the whole volume fits neatly on the GPU. Although this can be advantageous on the GPU as it forces all sampling to remain in the same data, thus increasing spatio-temporal cache coherency, each brick is rendered its own separate screen buffer 'tile'. This of course can massively increase not only the memory footprint of the rendering stage, but the complexity of compositing all the tiles to the final image in the correct order. These individual buffers, or tiles, must be composited in front-to-back or back-to-front order. This scheme is represented by the equations 2.4 and 2.5.

While rendering bricks out-of-order seems appealing, there are drawbacks that should be noted from the outset. As mentioned previously, ray-segments must be

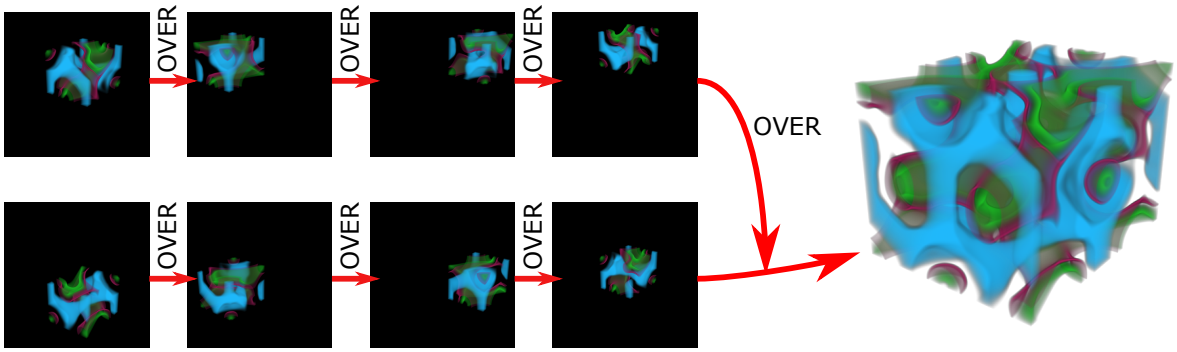


Figure 2.4: Splitting the ray-integral into segments which can be rendered out-of-order and composited at a later stage in-order.

composited in order. This creates an ordered list of how the bricks must be composited. If all bricks are rendered out-of-order, rays cannot take advantage of Early-Ray-Termination (ERT), where the ray hits such an opacity level that any subsequent samples will not contribute to the final image. This means there is a chance that a brick will be rendered out-of-order while perhaps not even contributing to the final image in the end. These drawbacks are covered in later sections.

Hsu [23] presented a sort-last algorithm where individual CPU cores are assigned volume bricks and screen buffer regions. The rendering is performed in three phases: Ray-gathering, integration and accumulation. In the first phase the rays which intersect the brick are determined, storing this information in a buffer. Phase two integrates over these rays using the standard volume rendering integral. The accumulated colour and opacity values are sent to the core in which the ray's pixel resides. The third and final phase accumulates all of these ray-segment values to generate the final image.

Ma et al. [25] presented a sort-last method for volume rendering similar to Hsu[23]. A kd-tree is used to divide the volume into bricks and distribute work. This same tree is then used to allocate work for the compositing phase where sub-images are divided and transferred to nodes in the system such that no node is left inactive

for the duration of the compositing phase. It is acknowledged that this compositing allocation method requires a large amount of communication between nodes. They also draw attention to the need for "data distribution heuristics" to equally distribute workload.

Law and Yagel [24] presented a direct volume ray-casting algorithm variant that stems from both image-order and object-order techniques. As with many other methods their approach divides the volume into smaller bricks. Their aim is to use a brick once and only once if possible. To accomplish this they use an "advancing ray front" which creates a front-to-back list of bricks. A processor steps through all of its ray segments for an individual brick before progressing to the next brick. Unlike Hsu's approach [23] they only need to keep one copy of the screen buffer as the ray segments are inherently ordered front to back.

Palmer et al. [26] build upon the work of Hsu [23] and proposed a distributed, load-balanced rendering solution using a sort-last method. They evaluate that the transfer of buffer tiles is less expensive than that of volume bricks. At the start of run-time, bricks are equally assigned to all nodes. Each brick is assigned a rendering cost after each frame which is then use in subsequent frames to load-balance by assigning relatively equal workloads across all nodes.

Samanta et al. [27] presented a polygonal rendering approach that is a hybrid between sort-first and sort-last for a cluster of PCs, splitting the objects over the clusters and then dividing the compositing work over the clusters.

Grimm et al. [28] presented a brick-addressing method that builds upon Law and Yagel's [24] advancing ray front algorithm. They evaluate the optimum size of brick and find that bricks that fit into the L2 cache perform best. Their approach is tailored for hyper-threaded processors such that when one logical thread is waiting for data from the L2, the other may avail of the otherwise idle execution units. To this extent a distinction can be drawn between this algorithm on CPU and on GPU where 32 threads will be performing in lock-step in parallel.

Mora et al. [29] used a fine-grained object-order rendering technique that takes advantage of template shapes in orthographic projections. Each cell (group of voxels) is projected to the screen using a template shape which, due to orthogonal projection, all have the same projected outline. They note that it is a drawback of their approach that they cannot perform perspective projection.

Hong et al. [30] presented an algorithm that streams bricks, or "cells", of volume data to the GPU while the GPU is simultaneously rendering other bricks. The bricks are ordered front-to-back and "layers" are created so that bricks that do not overlap in screen space are rendered in parallel, thus removing the need for a post ray-cast compositing step. It should be noted that, in their algorithm, brick projection occurs on the GPU after the data has been uploaded to device memory. This implies that bricks that do not contribute to the final image due to being off-screen may be transferred and therefore use bandwidth and computational resources. They make reference to the fact that CPU perspective projection would be too costly and would make ray-casting infeasible. They observe that bricks that posses the same Manhattan distance from the source (entry) brick belong to the same visibility layer, and thus can be rendered in parallel. The layers are generated using a propagation algorithm, starting with the source brick and visiting the neighbour bricks. It is at this point that non-contributing bricks are skipped.

Usher et al. [31] introduced the Distributed Frame Buffer, a framework to allow in-situ visualisation across multiple distributed nodes, building on Wald et al's scientific visualisation framework, OSPRay [32]. This is a prime example of object-order rendering in practice for visualising massive data-sets.

2.1.4 Time Varying Volume Data

Parts of this work study the effect of time-varying volume data [33] on DVR. This is volumetric data that has multiple frames representing discrete time steps of the model. This is especially useful, or even a requirement, in applications such as CFD simulations or 4D ultrasounds, to name but two. These frames may come from a stored sequence on disk, via network storage, or even from a real-time source such as medical scanning devices or in-situ scientific simulations.

The added 4^{th} dimension can add massive amounts of pressure to memory storage or bandwidth systems. To transfer an entire frame of data every single time-step can be needlessly wasteful unless absolutely required. For example, the supernova dataset shown in figure 1.1 (right) has over 100 time-steps. In practice, only about 40% of the data contributes to the final image based on the current transfer function. Determining updated regions of the volume and only transferring those that change is

a very simple optimisation that can be made, however there have been many investigations to optimize this step even more by either using compression [34] or by allowing the GPU to predict the change itself using temporal predictors [35]. Although this work examines time-varying data as a source of frames, the fields of compression and predictors to reduce transfer bandwidth are considered out of scope as they can be seen as an extra layer to the rendering process.

Shekhar et al. [36] use a multi-planar approach to render streaming cardiac data by bricking the volume data. Zhang et al. [37] present a system to visualise 4D cardiac data and synchronise its animation to ECG signals, with the option of displaying on stereoscopic devices. Their approach transfers the entire volume to the GPU rather than a subset.

In some cases, not touched upon in this work, the data may also be multi-variate, meaning there maybe multiple values for a given data point that represent different things. For example, a CFD simulation may have distinction between velocity and pressure values.

2.1.5 Alternative Representations & Rendering Methods

So far, this work has only discussed the use of regular grids of scalar values that represent a volume. Is it important to note that these volumes are referred to as 'dense grids', and are not the only type of volumetric data that exists, although they are the primary focus of this thesis. Unstructured tetrahedral grids are another type of volume representation that exists that — while not as common — can be used in scientific applications, and have been given attention recently in rendering performance literature [38].

In terms of the actual rendering, all of the work in this thesis uses volume ray-casting as the DVR method of choice, however it is useful to highlight that there are other approaches that may be used to display or visualise volumetric data. Before programmable shaders made it possible and accessible to implement ray marching loops on GPU, slice-based rendering was a good choice to make use of the parallelism and texture mapping hardware made available. In essence this approach sliced the volume with discrete planes, which were rasterised and then used to re-sample the volume in the fragment shader. While quite performant, these methods can show visual artefacts

in comparison to ray-casting DVR.

Isosurface rendering allowed the visualisation of specific scalar values by transforming the volumetric data into surface representations, showing the boundaries of volume intensities as a series of discrete shells. One such method to do this is marching cubes. Isosurface rendering can show clear and defined entities in the volume, but quite often visualising low frequency changes like varying temperature in a fire are better represented using semi-transparent transfer-functions and DVR. In addition, generating the isosurfaces can be a time-consuming task, a step which inhibits the streaming of time-varying data.

2.2 Graphics Hardware

This work takes special interest in using hardware designed for use in computer graphics; specifically GPUs. While CPUs are quite adept at complex logic they tend to lack the same amount of parallelism as a GPU. For example, at the time of writing, one of the main GPUs used in this work had 2,944 cores in comparison to the 8 cores available in the CPUs. These GPUs cores are geared towards highly parallelisable tasks — specifically those that will perform the same sequence of operations in large groups. When thought of as ray-marching through a volume sampling at regular intervals, GPUs become an obvious target for parallelising this rendering work. The remainder of this section discusses the advantages and disadvantages of the GPU at a relatively high level so the reader may understand the reasoning behind certain decisions in this work. The two main GPUs used over the duration of this work were the Nvidia Quadro K2200 and the Nvidia RTX 2080, and as such both of these are used as examples of GPU hardware when describing certain traits like core count and memory characteristics. At a more broad level, this means the two graphics architectures explored in this work are the Maxwell architecture (K2200[1]) and the Turing architecture (RTX2080).

[1] Although the K2200 has the 'K' prefix, it was part of the Maxwell rather than Kepler architecture unlike the rest of the Quadro 'K' series.

Figure 2.5: Comparison of the SMMs of Nvidia's Maxwell architecture [39] (left) and Turing architecture [13] (right).

2.2.1 Parallelism

The Quadro K2200 has 640 shader cores and the RTX 2080 has 2,944, which both dramatically outnumber the CPU in terms of computational cores. However, this most certainly does not mean that the GPU is a candidate to replace the CPU in all aspects of work. As mentioned previously, the GPU is excellent at highly repetitive parallelisable tasks, but when it comes to complex, highly branch-divergent code, the CPU is likely a better candidate. This is a result of how the shader cores in these Nvidia architectures are arranged and how they are controlled during execution.

Figure 2.5 shows what Nvidia calls the SMM of both the Maxwell and Turing architectures. Key to understanding how execution occurs on these cores can be seen in the way the individual cores are laid out. This is easier to see in the Maxwell SMM on the left where there are clearly 4 sets of 32 cores, with each set being governed by a warp scheduler. A warp is a group of 32 threads that operate in lockstep — i.e all

32 threads perform the same operation at the same time. It is possible for threads to diverge due to some branch condition, in which case any threads that have diverged from the rest of the warp are scheduled separately. This is an important consideration when programming GPUs, and a core reason that CPUs can handle complex logic better.

Parallelism on these architectures is also limited by register usage. Again, in figure 2.5, each processing block of 32 cores — or 16 integer and 16 floating point cores in the case of the Turing architecture — share a register file of 16,384 32-bit registers. This limits the total amount of threads — and hence warps — that can reside in an SMM at any one time. In general it is key to keep the register count of a GPU thread low so as the scheduler can mask warp stalls due to events like memory latency with another warp that is ready to execute its next instruction.

2.2.2 Fixed Function vs. General Purpose

As requirements of GPUs change over time, their underlying functionality adapts to meet demands. When GPUs first became a popular consumer piece of hardware, they were designed with triangular data in mind that took a set of three coordinates in space, applied a transformation, rasterised them to 2D space, and applied shading and texturing. All of this was done through fixed-function hardware which allowed for faster execution of repetitive tasks. Over time the requirements of GPUs got broader, and more programmability was in demand. To make a long story short, the GPU evolved into an almost completely programmable many-core device, but retained some fixed-function hardware for operations like texture reads and rasterisation. This programmability gave rise to the term General Purpose GPU (GPGPU) programming, and the massively parallel capabilities made GPGPUs a good candidate for the embarrassingly parallel problem of DVR.

What is interesting about recent years is that there is a cyclic trend that is bringing GPUs back to more dedicated hardware, and few more visible than ray-tracing fixed-function units. Specifically, the RTCore introduced into the Nvidia Turing architecture, that supplies ray-AABB intersection, ray-triangle intersection, and BVH traversal hardware. This is as a result of a continuing paradigm shift from polygonal screen-space tricks for real-time global illumination to more realistic light-transport

methods using real-time ray-tracing. This paradigm shift is very visible in recent literature that focuses on hardware support for BVH building [40] and ray-tracing [41].

To elaborate on these fixed-function hardware approaches, Doyle et al. [40] presented one of the first BVH builders implemented in hardware that supported the Surface-Area Heuristic (SAH) method. They showed that this fixed function approach gave drastic improvements over pure CPU-based BVH building. This work was later elaborated on to support BVH construction for mixed-media visualisation, including volumetric data [42]. Lee et al. [41] proposed low-power fixed function hardware that allowed low-power mobile GPUs to complete with desktop GPUs for ray-tracing. These papers can be seen as the precursors to the dawn of fixed-function ray-tracing hardware in modern GPUs, although it is worth noting that at the time of writing, it is unclear if BVH *building* hardware is present in the Nvidia Turing architecture.

Considering that DVR — or more specifically volume ray-casting — can almost be considered an extension of ray-tracing, this new hardware can be potentially be utilised to accelerate the rendering process alongside massively parallel GPGPU methods. To be more direct, BVH hardware can be considered a target for ESS during DVR. In this thesis, in chapter 6, BVHs are evaluated as an ESS contender on GPU and show substantial benefits to be gained by using the RTCores.

2.2.3 Benchmarking

These advancements in GPU architecture are fantastic for the ever-evolving graphics and compute market. However, because the hardware industry can be competitive, companies such as Nvidia, Intel or AMD tend to be quite secretive with their designs and underlying architectures. For example, in the previous section the presence of certain hardware components is acknowledged — and L1 and L2 cache for example. However, the actual operation and therefore — critically — performance characteristics are hidden from the public. There has been some literature detailing how GPU architectures may implement caches, such as Doggett in 2012 [43], but there is no guarantee that these still hold true. This sort of information can be absolutely critical when designing applications and algorithms that are so heavily dependant on the memory hierarchy, such as DVR.

While the ins-and-outs of these components are generally trade secrets, it is

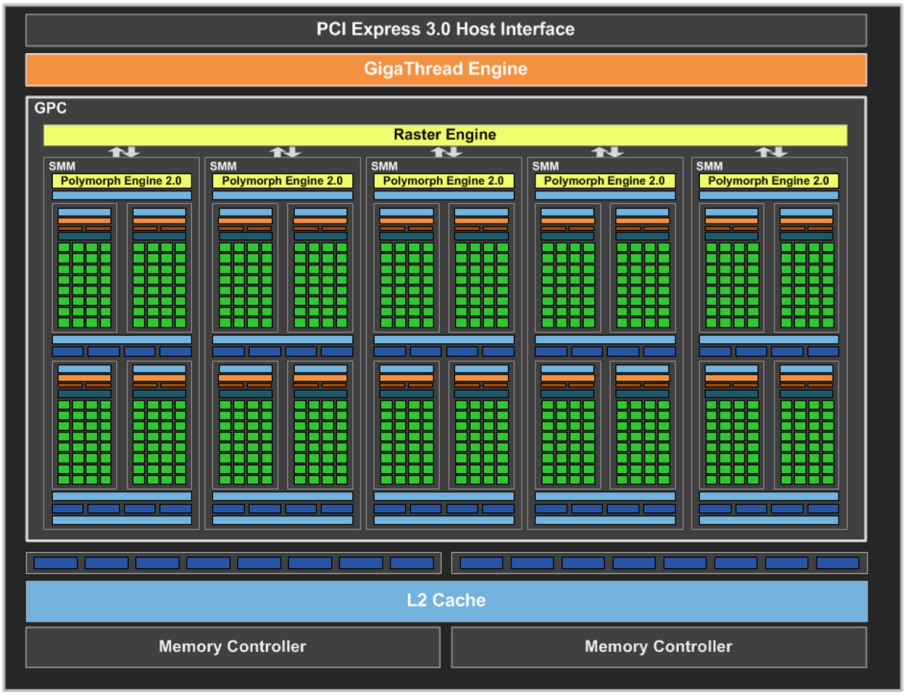


Figure 2.6: Nvidia Maxwell (GM107) architecture [39].

possible in some cases to obtain just enough useful information about their performance characteristics through methods like micro-benchmarking. Mei et al. [44] used this approach to dissect the memory hierarchy by performing small, primitive operations and empirically building upon these to discern the cycles — and the characteristics by proxy — of caches and texturing hardware. This is a very low-level approach to building a dictionary of operations to better understand the underlying hardware. In the case of DVR a mix of low-level and more applied usage characteristics are required to make informed decisions about memory layouts and spatio-temporal coherency.

2.2.4 Memory Efficiency

While the parallelism and hardware capabilities of GPUs seem like an ideal platform for DVR, there are some limitations. The potentially first and foremost of these is the memory hierarchy. When this work began, the memory resources of the GPU were a fraction of host main memory — 4GB of VRAM vs 32GB of main RAM — and while this is fine for smaller, static volumes, keeping hold of important data for larger or

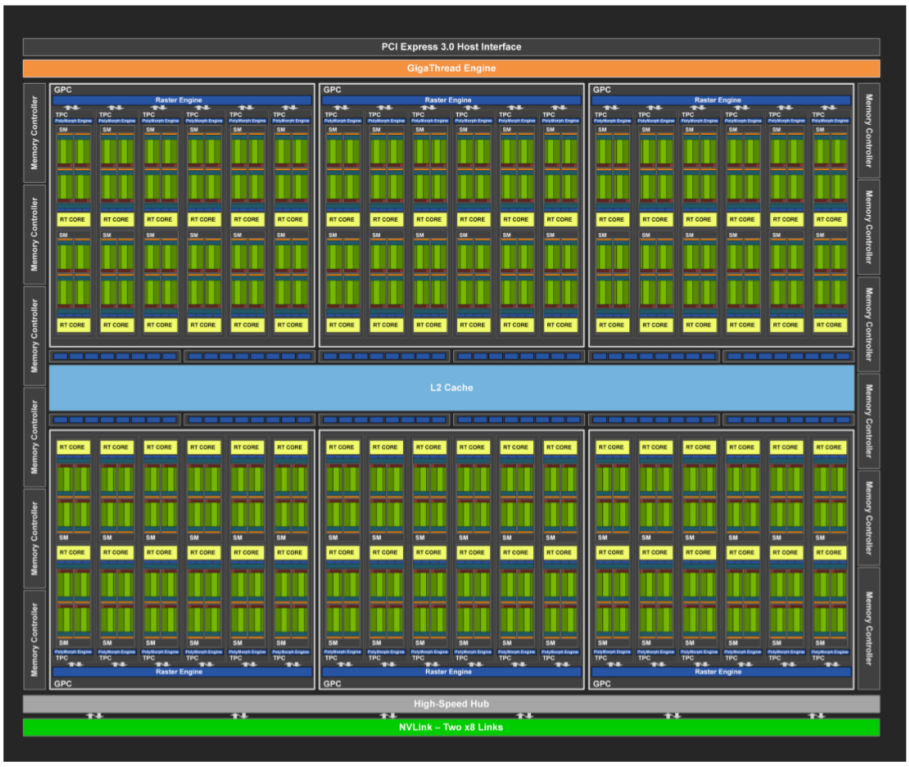


Figure 2.7: Nvidia Turing (TU102) architecture [13].

time-varying data is a complicated task.

Even when a full static volume fits into VRAM, the rest of the memory hierarchy *must* be taken into consideration. If DVR can be imagined as a wave-front of ray-samples traversing through a volume, it follows then that samples are hitting a large range of memory addresses. In figure 2.5 that shows the SMMs of the Maxwell and Turing architecture, there is an L1 cache and a few texture units shared over the whole SMM. Below that, shared by all SMMs in the two architectures is a larger L2 cache, shown in figures 2.6 and 2.7. Even still, this L2 cache is only 2MB on Maxwell and 6MB on Turing. Coalescing memory accesses has always been considered as GPU performance tuning 101, and this is no different when performing DVR.

Knittel [45] presented a complete volume rendering system that focuses on the low-level arrangement of memory, paying particular attention to cache and page structure, while also making use of hand-optimised assembly and SIMD instructions. Knittel employs a spread memory layout that uses parts of multiple pages to reduce the chance of cache trashing and makes use of a cubic-interleaved ordering to keep cubic sets of voxels in the same cache line, thus providing neighbouring sample points with low-latency cache accesses. Knittel's work is implemented and highly optimised for the Intel Pentium-III processor and only makes use of graphics hardware when magnifying the result image from 256x256 to 512x512 pixels. While Knittel's work is useful for single frame volumes, the approach may not be appropriate for a time-varying dataset due to the memory footprint proposed.

Mensmann et al. [46] proposed a GPU volume ray caster which divides the view frustum into slabs, designed for CUDA. Each ray in a slab is sampled with the intention that the sampled data fits into each thread blocks shared memory. Subsequent operations such as gradient calculation for lighting are performed on these samples instead of the actual volume data. They made note that compared to shader approaches for GPU ray casting, their slab based approach underperformed in some examples, namely where early ray termination occurred due to high opaque materials in the volume. In cases where the volume was considerably transparent and voxels would have been multiple times for gradient calculation their approach saw significant speed-ups. They also make note that the more registers available to each thread block, the better their algorithm appeared to perform.

Bethel et al. [47] perform an analysis on the effect of algorithm parameter

variations with respect to overall runtime and L2 cache misses on CPU and GPU. Among other parameters they vary the size of the screen buffer tiles and the memory layout of the volume data. Their results on GPU indicate that there is a sweet-spot in terms of thread-count, but more surprisingly they show that thread counts that have some of the shortest run-times do not necessarily have the lowest amount of cache misses. They conclude from this that there is a trade-off to be made between warp utilisation and good cache performance.

Labshutz et al. [48] alleviate memory transaction bound data structure traversal by just-in-time compiling the traversal code specifically tailored to the data structure. For example, rather than performing pointer and key lookups in a binary-search or kd-tree traversal, the key values are explicitly compiled into the code reducing memory transactions. Furthermore, they allow the combination of multiple types of data structures depending on the homogeneity of volume regions. While their work is novel and most definitely reduces the bandwidth required for traversal lookups, in the case of large volumes that are highly subdivided their approach transferred from memory bound to instruction bound with large amounts of branching in JIT compiled code, along with considerable JIT compilation times.

2.2.5 Out-of-Core Rendering

With so much potential data contained in the volume, reducing the amount of voxels re-sampled is of prime importance. ESS techniques are an obvious optimisation that can be made. Just because a volume may be 1024^3 voxels does not mean that 100% of the volume is assigned a colour as determined by the current transfer-function. This means that — using the transfer-function — a regionisation of the volume can be made containing only those regions that may contribute to the final image, and thus the rest of the volume can then be skipped over. To extend this idea even further, when both the bandwidth between CPU and GPU and the VRAM memory constraints are taken into consideration, only transferring data to the GPU that is necessary for rendering is an obvious optimisation to make. This is referred to as 'out-of-core' rendering, with techniques tending to be 'output sensitive' — that is to say, only data that contributes to the output are considered.

This out-of-core rendering method does tend to have its drawbacks however.

In order to render a single frame, knowledge of the output-sensitive dataset must be known to transfer from main memory to VRAM. In most state-of-the-art approaches, this is achieved using a feedback loop between the CPU and GPU [4, 49, 50, 5]. While these are true output-sensitive solutions, the feedback loop comes at the cost of waiting over multiple frames to have a correct render. Additionally, maintaining the currently paged data and sampling said data requires some form of page table [5] and look-up structure [4], which adds a layer of complexity.

Crassin et al. [4] presented the *Gigavoxels* system for rendering massive scalar volumes on the GPU. Their system maintains an octree representation of the scene with an adaptive loader on the CPU, using a back-and-forth feedback system between the GPU and the CPU for determining the work-set of bricks — based on the current transfer-function and view occlusion — needed to render the volume. Their traversal method is based on the stack-less kd-restart algorithm. At roughly the same time, Gobetti et al. [2] proposed a very similar approach to *Gigavoxels*, differing in the octree attributes. In their method they added pointers to neighbouring bricks allowing traversal directly to the neighbour rather than restarting from the tree root.

Fogal and Krüger [49] present a cross-platform generalised system for out-of-core volume rendering with a bricked data representation. The system is designed to run on GPU systems and, in order to display larger datasets, CPU clusters. In a later paper [50] they build upon their previous work [49]. Their approach bricks the volume and begins rendering, without data, on the GPU. During the rendering stage different level-of-details are determined for each brick and the sampling rate is adjusted accordingly. When a ray encounters a brick that is not present in device memory the absence is recorded and eventually a list of the required bricks is communicated back to the CPU. This back-and-forth communication between the CPU and GPU leads to the volume being incomplete for several frames until the final result is composited. In addition, the CPU is not told which bricks are missing until the GPU completes a rendering pass. This means that the CPU is idling while the GPU is active, and vice-versa.

Hadwiger et al. [5] presented a full volume-rendering system designed to handle large scale data on GPU. They focus on the scenario of an incomplete volume sourced by an Electron Microscope (EM) with partial volume data arriving every 15 seconds, with the intent on displaying the incomplete volume as data arrives. They posit that — while data-structures like octrees are used in previous systems [2] — when dealing

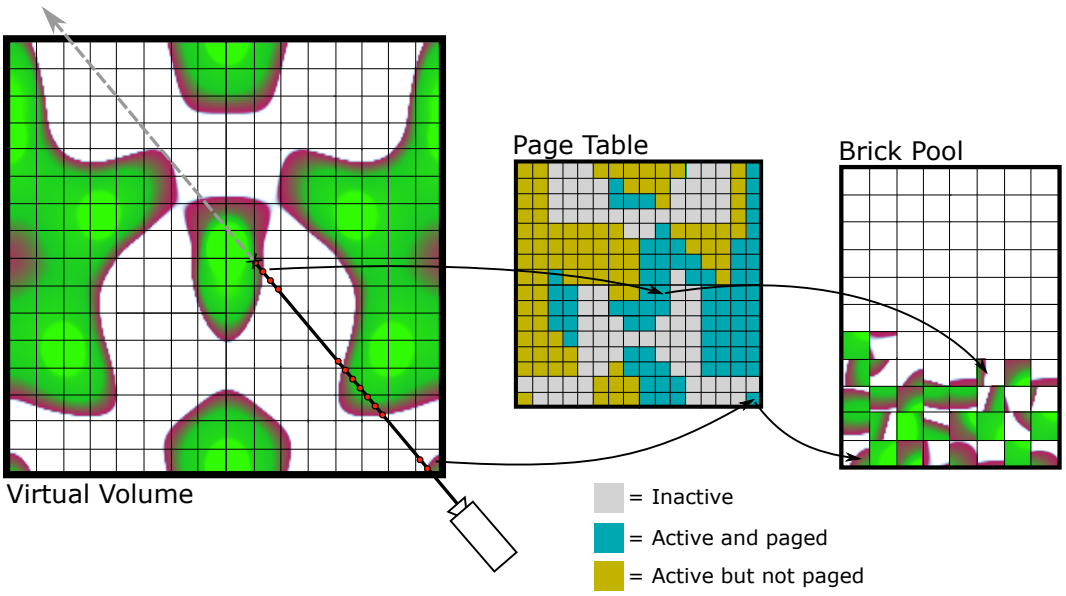


Figure 2.8: Example of a virtual volume with a page table and brick pool. As the virtual volume is sampled, a look-up into the page table is performed, which links to the brick pool if the brick is active and paged into GPU memory. This is the simplified basis for methods like Gigavoxels [4], Tuvok [49, 50], and Hadwiger's works [5, 6].

with streaming partial data, a different approach must be used. Their solution to this problem is to use a virtual paging system on the GPU which conceptually divides the volume into bricks and overlays these with a multi-resolution hierarchy. The general ray-casting approach is basically standard DVR, however each sample point looks up a 3D page table, stored in a 3D texture, with a desired level of detail and a volume position. This look up will either provide the location of the block data in GPU memory [2], or generate a "cache miss" which will invoke the CPU to dynamically create the block data from the partial image data given by the EM. They make note that cache misses, coupled with the corresponding updates, are effectively distributed across multiple frames meaning that, although a smoother frame rate is provided, there will be frames that are not visually correct. It is worth mentioning that their comparisons versus octree traversal methods use quite dense data as determined by the transfer function, in comparison to their later work that deals with sparse volumes [6] discussed later.

Liu et al. [51] presented a volume rendering approach that makes use of an octree and load balancing traversal and rendering on both the CPU and GPU. They propose that the CPU is a better candidate for tree traversal and as such the tree cut is generated on the CPU. Non-empty nodes are then rasterised on the GPU. The paper also makes use of macro-cells, a further subdivision of bricks, to more tightly collect active rays and to perform empty space skipping at a sub-brick level. It should be noted that volume data is stored on a brick scale rather than a macro-cell scale. Liu et al also make use of tri-cubic interpolation as opposed to trilinear interpolation to produce smoother results.

Hoetzlein [52] took the multi-level hierarchy structure [53] that is used in OpenVDB [54] and made it compatible for GPU allowing for the possibility of in-situ sparse volume simulation and rendering, dubbed GVDB. Although a much more configurable hierarchy, the principle of ESS in GVDB is an adoption of VDB [53] and also follows the Hierarchical 3D Differential Analyser (H3DDDA) traversal method [55], which is more similar in essence to Fogal et al's [50] that skips to the next brick and samples at the required resolution rather than determining the resolution at each sample like Hadwiger et al. [5]

[2] All bricks, even different resolutions, are said to be stored in the same 3D texture.

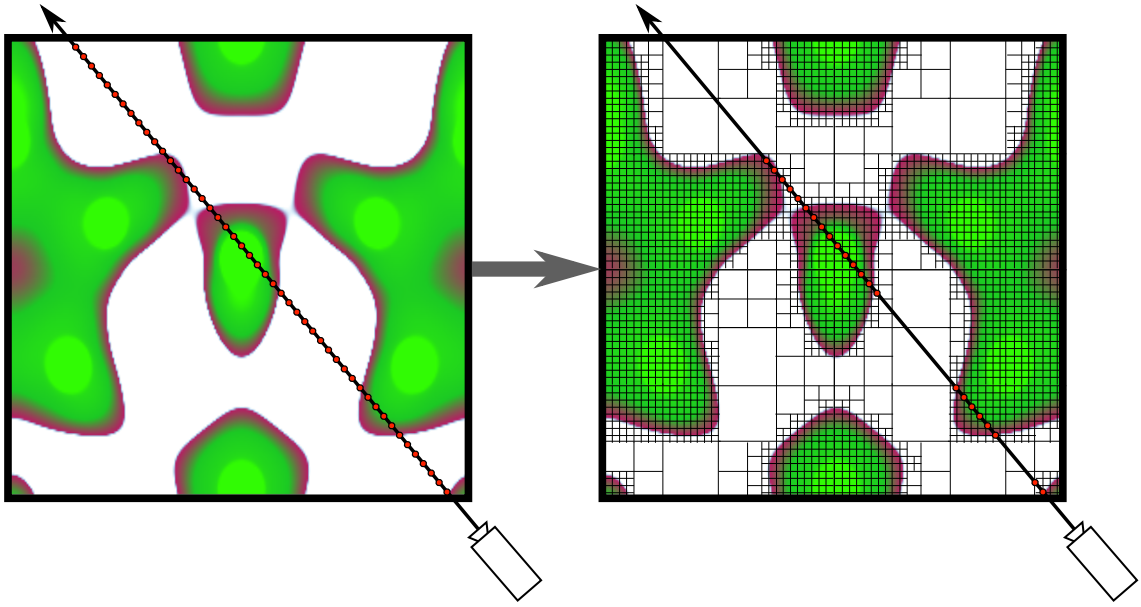


Figure 2.9: Example of an octree that can be used to accelerate empty-space-skipping.

2.2.6 Acceleration Data Structures

Acceleration data-structures are potentially one of the most important aspects in any sort of computer rendering and this is especially true with ray-casting and path-tracing. In essence, a spatial data-structure wraps all geometric primitives in a scene in a hierarchical graph and allows for a fast search for queries, be it ray intersections queries for rendering or geometric intersection queries for collision detection. For DVR the former case is more relevant.

Over the decades there has been a myriad of spatial acceleration data-structures. Among the most popular and long lasting have been Binary Space Partitioning (BSP) trees, octrees, kd-trees, and BVHs. Each of these all have their own trade-offs in terms of build-times, memory footprint, and traversal performance, and the chosen data-structure for a particular use case tends to have parameters that have the same trade-offs again — for example the amount of children per node in a BVH or the amount of leaves in an octree, etc.

The most prevalent data-structure in DVR is — by far — the octree. Hadwiger

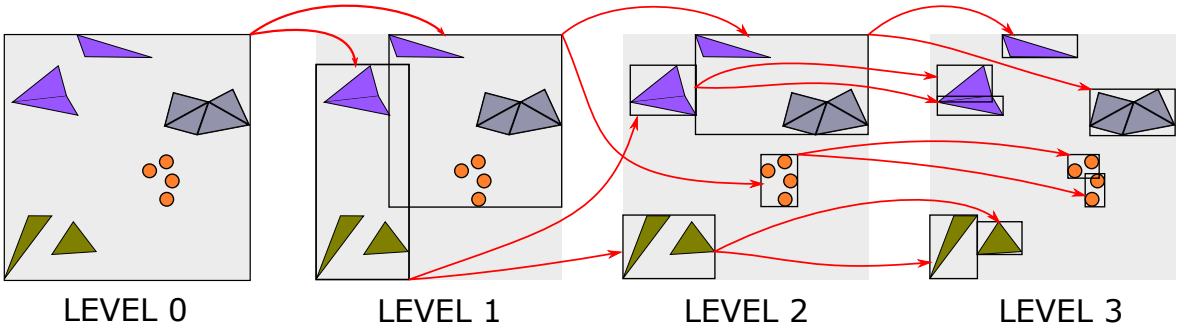


Figure 2.10: Bounding Volume Hierarchies used in the context of surface representations of geometry with axis-aligned bounding boxes (AABB). In this example, each node has two child nodes. An interesting trait of BVHs is that node bounding boxes may overlap, which adds slightly more traversal complexity in a trade-off for bounding arbitrary space.

et al. [5, 6], Gobbetti et al. [2], Laine and Terro [56], and Liu et al. [51], among others, all use octrees as their data-structure basis. Octrees are a regular spatial subdivision that are well suited to the regular grid data seen in volume rendering. An example of this is shown in figure 2.9, where a volume is sub-divided in to bricks and an octree is built based on the sub-divisions. Traversal through an octree is generally trivial, although due to the currently stack-unfriendly nature of GPU programming, specialised algorithms like *kd-restart* and *kd-backtrack* [57] have been developed and are used in many of the octree-based DVR approaches. As well as octrees, other data-structures have been chosen from DVR. Crassin et al. [4] used an N^3 tree, although the format chosen was very similar to an octree. Hoetzlein [52] used a more flexible hierarchical multi-level grid.

Opposed to the many positives, one drawback of octrees is the inherent depth of the tree if thin strands of media are present in the volume. In contrast, BVHs handle thin strands of geometry with fewer inner nodes. However, BVHs were designed with geometry that resides in continuous space in mind, not a regular grid structure. An example of a BVH is shown in figure 2.10. Nested bounding boxes are stored in a hierarchical manner that allows for ray-box intersections to cull searched nodes. In particular, axis-aligned bounding boxes (AABBs) are popular for BVH representations,

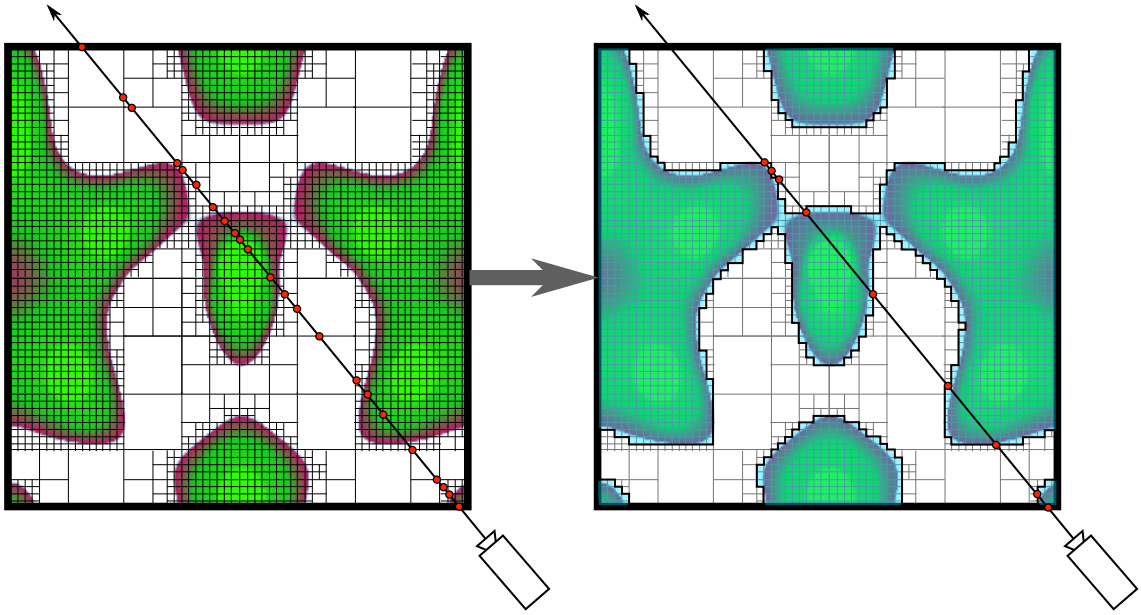


Figure 2.11: Sparseleap [6] accelerates ESS by creating a proxy-geometry around active regions of the volume octree. This figure is over-simplified but sufficient to demonstrate the core concept. The ESS octree traversal points (red-dots, left) have transitions that are redundant. By rasterising a tight proxy geometry, ESS entry-exit points are far reduced.

as the orientation constraint on the bounding box allows for less complex intersection algorithms, and therefore faster BVH queries and traversals.

Unfortunately, there has been a major lack of research into using BVHs in the area of volume rendering. The only novel research that really exists comes from Knoll et al. [58]. They provide a full-resolution DVR solution on the CPU using coherent, using ray-packet traversal on an implicit BVH — a BVH that contains min-max data values that is transfer-function independent. They showed that — at the time — the CPU was a better candidate for large-volume full-resolution rendering than GPUs. This was due to the more branch-capable CPU hardware, and more significantly the larger main memory available.

Hadwiger et al. expanded upon their previous large scale out-of-core work [5] and presented 'Sparseleap' [6]. To re-summarise the base work, empty space skipping was achieved by traversing an octree on the GPU. In this work, however, they note

that octree traversal is costly when jumping from brick to brick in a dense region of the volume. To alleviate this, bounding box geometry is emitted on the CPU for non-empty or 'unknown' bricks. In this case, unknown bricks are bricks which haven't been paged from disk — or another source — into main memory, and are as such unclassified and presumed not needed by the GPU until requested. The emitted geometry is then rasterised in order to create an on-the-fly compressed linked list of ray-segments that allow skipping over large regions of the data structure. This work is very similar in concept to Liu et al's [51] approach. The key differences are as follows: Sparseleap supports not-yet-loaded ('unknown') regions of the volume in true out-of-core DVR fashion. Liu et al's method is a one-shot beginning-to-end rendering of the volume whereas Sparseleap is progressively refined to make it a true output-sensitive renderer. Sparseleap doesn't determine the brick resolution on the CPU like in Liu et al., but rather is refined during feedback. There are of course pros and cons to both approaches. It should be noted that while an octree-like structure was used for the occupancy tree, this is decoupled from the actual underlying volume representation in their implementation.

Concurrent to work presented in chapter 6, Zellmann et al. [59] presented work that used a Summed Volume Table — another name for what this work calls a 3D Summed Area Table — to accelerate the building of kd-tree acceleration structures for DVR.

As stated before, it is clearly desirable to display volume data on relatively simple systems with minimal specialised hardware rather than large clusters of systems working in parallel. These systems are more complex and costly to run in comparison to single-machine setups. Considerable advancements have been made in computer graphics hardware. Notably, certain ray-tracing techniques have now been implemented in hardware allowing for much faster acceleration data-structure traversal, such as Nvidia's RTCores in their RTX line of GPUs, which can accelerate ray-tracing via VulkanRT, DirectX Raytracing (DXR), or their own Nvidia OptiX API [60]. These data-structures are pivotal concepts in volume rendering, allowing for regions of the volume which do not contribute to the final image to be ignored and skipped during rendering.

2.2.7 Volume Rendering Hardware

Up to this point, the only graphics hardware covered has been that designed specifically for surface data, generally in the form of polygons. It is worth noting that volume-rendering specific hardware — while much less common and researched — does indeed exist. Most notably VIZARD II [61, 62] and VIZAR [63] are Field-Programmable Gate Array (FPGA) based systems that were specifically designed for real-time ray-casting and shading of volumetric data that allowed a certain amount of reconfigurability.

Volume-rendering specific hardware solutions are a massively interesting topic, however they are not the focus of this thesis, which is to evaluate efficient DVR algorithms and accelerations on non-specialised consumer hardware.

2.3 Light-Fields

Standard 2-D screens have long been the norm for viewing computer generated images, and have come a long way in fidelity and colour correctness. Both of these factors aid user perception of a scene, but neither particularly improve the sense of spatial depth in an image.

Displays in the form of auto-stereoscopic lenticular-lens flat-screens, active stereoscopic screens that use alternating light-blocking glasses, or dual-display near-eye HMDs dramatically improve the users depth perception into a virtual scene. In these cases depth is determined by the user by the angle of the eyes, called vergence. But these displays still have a drawback. While vergence is the primary sense for depth in the human-visual system, in the physical world it is closely coupled with 'accommodation' which is the changing of focus of the eye's lenses. A mismatch in vergence and accommodation is present in all of the previously mentioned displays, where depth is determined by vergence, but the eyes are focused on a flat plane close to the user. This is called the vergence-accommodation conflict problem and can lead to an uncomfortable viewing experience for users.

Potential solutions around this are the emerging light-field display technologies. A light-field considers the transport of light in a volume of space. The light-field concept was introduced in [64, 65] as an efficient image-based rendering method. A light-field aims to capture all light rays passing through a given volume of space, and is commonly

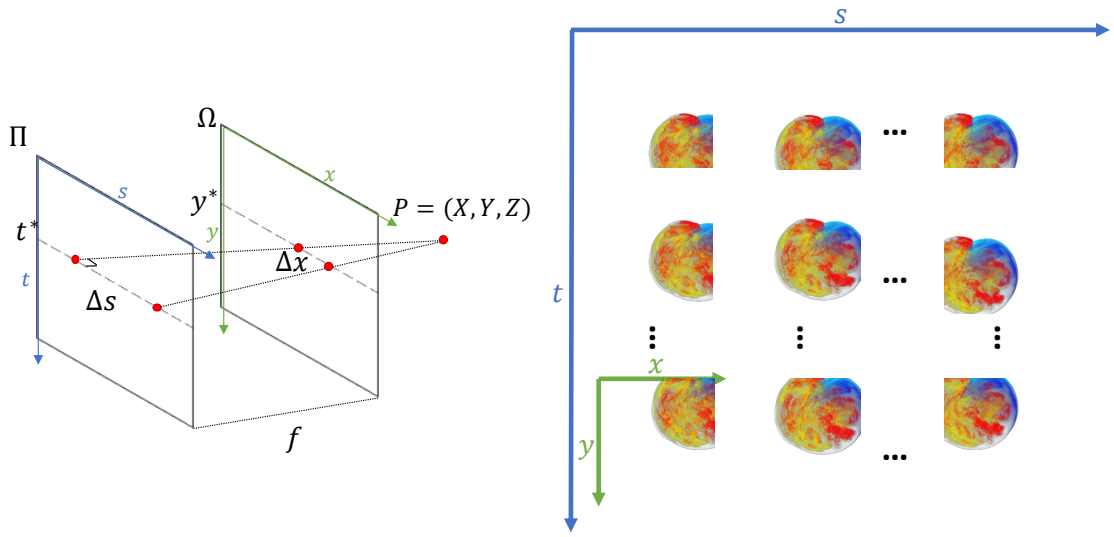


Figure 2.12: Light-field two-plane parametrization (left) and matrix of views representation (right).

parametrized using two parallel planes, as shown on the left in Figure 2.12. It can be formally represented as a 4D function $\Omega \times \Pi \rightarrow \mathbb{R}, (x, y, s, t) \rightarrow L(x, y, s, t)$ in which the plane Ω represents the spatial distribution of light rays, indexed by (x, y), while Π corresponds to their angular distribution, indexed by (s, t). Using this parametrization, the light-field can be considered as a collection of 2D images, called sub-aperture images regularly arranged on a 2D grid (see right of Figure 2.12). As the most common in the literature, two-plane parametrisation can be used to interface multiple rendering methods [64, 66] and light-field displays [67, 68].

2.3.1 Light-Field Capturing

Light-fields have not only been used for display technologies, but also for photography [68]. This allows then for a real-world scene to be almost directly shown on a light-field display or be refocussed at a later point. Both of these applications have huge benefits to users. While the former has been discussed already, the later application of moving the focal plane and camera position post-capture allows for considerable artistic flexibility, although this is not a primary application of this thesis' work.

Capturing a light-field of a real-world scene poses its own problems with regard

to optical hardware and data bandwidth for storing high angular and spatial resolution images. This is almost analogous to the issues presented when capturing a light-field in a virtual scene. In essence, a light-field capture of a virtual scene involves rendering from different view points for the same frame. An example of this can be seen in figure 1.2, where the same volume can be seen from slightly offset positions. This poses the challenge of how to render this data across multiple view points in computationally and data efficient manner. This is one of the primary focusses of this thesis.

2.3.2 Light-Field Displays

While the term 'light-field' is by no means a new one, recent research is surfacing on adapting light-fields for HMD display technologies and computer graphics with companies such as Avegant [8] and Daqri [9] researching light-fields in consumer and industrial use. Light-field TV-screen like products have been available for some time, Holografika [10] is one such long-standing manufacturer. but these displays have been prohibitively expensive for the consumer market. More recently though, small- to mid-scale light-field displays have been coming to fruition. One such example of this is the Looking-Glass display [7] which is effectively an auto-stereoscopic lenticular lens display providing 48 views of horizontal parallax.

Mid-Distance Light-Field Displays

Mid-Distance multi-view or light-field displays tend cover technology that can be multi-user by using scattering media [69], specially designed monitors / TVs or projector systems, to name but a few methods. For example, the Visual Computing Group in CRS4, Italy base a lot of their volume rendering research on a multi-projector light-field display [1, 3, 70, 71]. Their system, as the name suggests, uses multiple projectors behind a "sharply transmissive" holographic screen that maintains narrow horizontal viewing zones, but scatters light correctly in the vertical domain. As such, this is only a horizontal-parallax display, otherwise called an auto-stereoscopic display. Marton et al. [71] use the size of this display to their advantage allowing intuitive exploration of the volume using hand gestures. This, combined with the aforementioned collaborative property of larger, mid-distance displays, is a definite advantage over near-eye displays such as HMDs.

However, one clear disadvantage of this method is the inherent lack in mobility of the fixed viewing zones. In general, there is only the area in front of the screen at a certain distance that provides valid imagery. In contrast to this, methods using scattering or rotational media have been proposed. For example, Yuasa et al. [72] interestingly use a projector system directed at a cloud of fog to create the appearance of floating objects. However, as might be expected with scattering media, the diffusion of light through the fog tends to result in slightly blurry objects. Another approach is to use rotational equipment, such as a helix as implemented by Geng [73], with a high-speed projector system projecting slices of a scene on to the geometry. Mora et al. [74] use an isotopically emitting display to visualise 3D datasets by creating an intermediate light-field they dub a "lumi-volume". Unlike the previously mentioned displays with have fixed, discrete viewing zones, these methods allow for continuous movement around the imagery.

Near-Eye Light-Fields

When the Human Visual System (HVS) is developing people learn to recognise depth from two features of our eyes. Firstly, convergence, where eyes rotate in their sockets to aim at a point in space. It is this depth cue upon which stereoscopic HMDs build upon. The second cue comes from our eye's lens changing shape to focus a point in space onto our retina. This is referred to the accommodation focus cue. With stereoscopic HMDs, the eye's lenses are always focused on the same plane while the HVS gains depth knowledge from convergence cues, outlined in figure 2.13. It is this disparity that can lead to eye strain and nausea in some users [75, 76, 77]. This issue particularly occurs in a certain range of distance in which convergence and accommodation more strongly work in tandem. Cutting and Vishton [78] performed a wide study on the HVS in the context of depth perception and found that vergence and accommodation combined have a useful range of up to 3 meters.

As a side note, a common term in HMD papers that focus on vergence-accommodation is "dioptre" which is a measurement of distance. In this case it's used as the distance from camera/eye and is defined as $1/d$ where d is the distance in meters. It follows that objects close to the eye have a large dioptre value (20cm would be 5.0D) and as the distance increases the value decreases towards zero. As such, objects with what

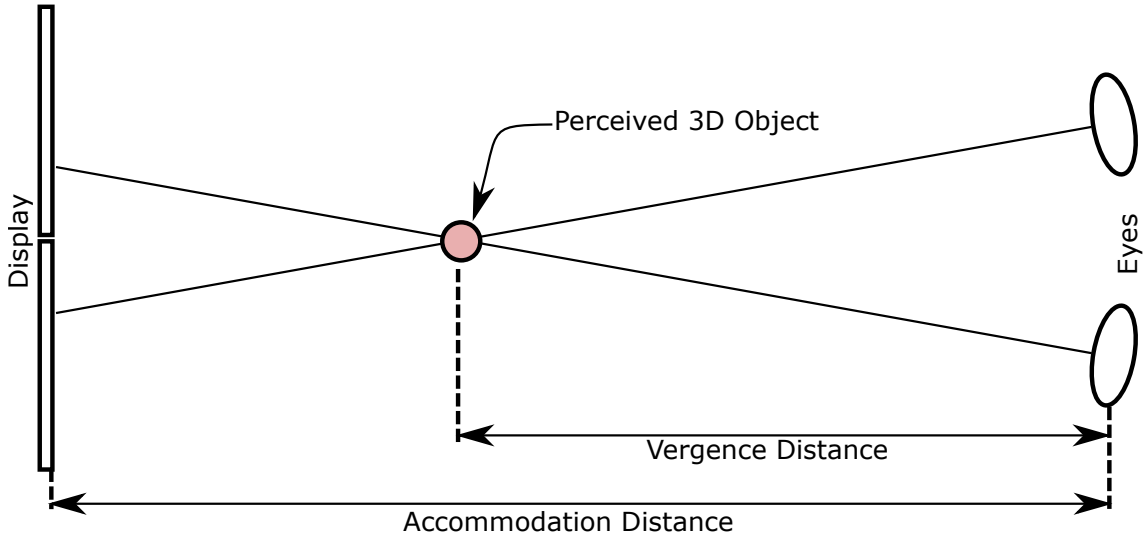


Figure 2.13: While the focal plane of a HMD is located at a fixed distance, virtual objects appear away from that plane. In this case, there is a disparity between the vergence and accommodation distances.

could be considered at "optical infinity" have a dioptre value extremely close to zero.

It should also be noted that the HVS is very complex and can be tricked into seeing a good stereoscopic image with some approximations. Bulbul at al. [79] note that rendering the same scene twice for two camera positions can be an expensive process especially when lighting effects such as shadows are introduced. Their research evaluates the effect of Binocular Suppression theory, where the overall perception of a stereoscopic scene is strongly biased towards the dominant image in the pair. Their results show that for certain modifications made to the second, reduced quality image, the resulting stereo image resulted in a pair that was acceptable to users. Techniques such as up-sampling, specular highlights (or lack thereof), texture re-sampling and mixed shading worked well. However, they note that there are some modifications that result in an unacceptable difference, such as mixed-level AA and simplified meshes.

Light-fields in a near-eye context have a remarkable trait that can be seen as quite important to the future of VR applications. As mentioned before, current market HMDs such as the Oculus Rift and HTC Vive present the user with vergence-based focus cues, but on a fixed focal plane. As such the HVS ends up with Vergence-Accommodation Conflict (VAC). Near-eye light-fields give a solution to this by pre-

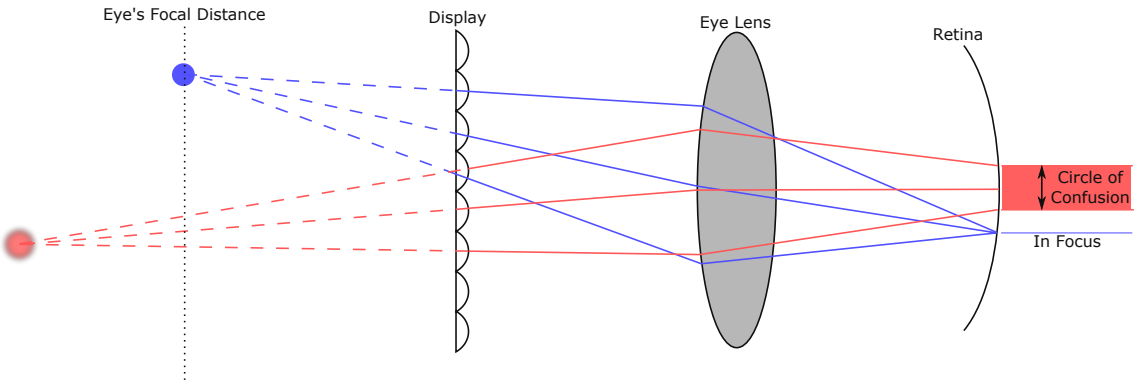


Figure 2.14: The basis behind near-eye light-fields as a solution to VAC. Since the top, blue object lies on the focal plane, rays projected at appropriate angles through the display, in this case utilising a lens-array, are refracted through the eye's lens and to a relatively singular point on the retina, appearing in focus. In contrast, the red object appears out of focus. Note that, when the eye's lens shape changes, the rays are refracted differently, allowing the red object to come into focus and vice-versa for the blue object.

senting the eye with rays from different directions at the same points. This effect is outlined in figure 2.14 where the lower, red object is further from the eye's focal plane and as such is projected with a larger circle of confusion on the retina.

Lanman and Luebke [80] propose a head-mounted light-field display, or rather two separate light-field displays in close proximity to both eyes. Their solution makes use of existing OLED displays from an established stereoscopic HMD overlaid with a micro-lens array. The OLED displays show a specifically arranged group of light-field sub images which, when viewed through the micro-lens arrays at a specific distance, supports continuous accommodation of the eye in a finite depth of field. This is the solution that figure 2.14 more closely is based on. Note that in this system the elemental sub-images are below the lenslets in the lens-array. It then follows that the sub-images are quite small, and as such are much lower resolution than the entire display. This is worth keeping in mind when considering the overall graphical computation requirements.

Huang et al. [68] use dual-layer stacked LCDs combined with non-negative matrix factorisations to emit a rank-1 approximation of a light-field to the user. They note that there is a trade-off between accommodation range and spatial resolution,

and in this case opted to go for spatial resolutions unlike Lanman and Luebke [80]. Regardless of the technology used to display the data, their work does not go into rendering optimisations for their display.

2.3.3 Light-Field Volume Rendering

Rendering volumes to multiple views — as is the case in light-field rendering — is a non-trivial problem if the outcome is to be efficient enough for real-time interaction. While computer graphics has only relatively recently adopted a light-field as a target display system, multi-view rendering has a long history.

Most of the current DVR research focuses mainly on rendering volumetric models for single view render targets and displays. However, in recent years there has been a large push on multi-view renderings targeting stereoscopic displays like a HMD, large format stereoscopic screens, and venturing into auto-stereoscopic (horizontal parallax) and auto-multiscopic (horizontal and vertical parallax) display technologies. All of these targets require presenting the same scene multiple times across few or many displays.

As briefly mentioned before, driving these displays potentially requires rendering the same scene — or volume — multiple times. For example, in the case of the Looking-Glass, the same data needs to be presented from up to 48 different view points. Polygonal methods for multi-view rendering have been researched much more than DVR multi-view techniques. Unfortunately most of the polygonal solutions cannot be applied to volumetric data while expecting the same performance enhancements. To maintain a correct multi-view rendering of a volume, the data must be ray-cast at least once, but potentially up to the amount of views. Basic knowledge of computer architecture should then lead the reader to consider how this data could be accessed, and the temporal-coherency of this re-sampling. What follows is a survey of the current approaches to stereoscopic or multi-view DVR. The simplest form of multi-view rendering is stereoscopic, where two views of the scene or volume are rendered from two points, mimicking the human visual system. These approaches tend to target HMDs.

Stereoscopic

The precursor to multi-view rendering is two-view or stereoscopic rendering. Stereoscopic displays have been around for quite some time, and it follows that scientific visualisation has targeted these almost as soon as they arrived.

Adelson and Hansen [81] presented efficient approach to stereoscopic volume rendering. They consider the regularity in ray-ray intersection that two parallel projected cameras would have. Their method takes advantage of the fact that a simple transformation can relocate an accumulated segment from one view's ray to a ray in the other view. Any gaps or incomplete rays in the second view are then completed as normal. It has been noted that this method can be prone to integer rounding and lead to unwanted artefacts in the final image [82].

He and Kaufman. [82] present an improvement on Adelson and Hansen's stereoscopic rendering method [81]. In their approach they assume that the cameras use parallel-projection and notice that while ray-casting the left-eye, a fixed amount of samples from each ray will always influence a single pixel in the right-eye. To this extent, they create a scheme which composites segments from the left-eye rays onto pixels in the right-eye. While this method performs well, the visual results are passable but do have errors when compared to a ground-truth.

Law and Yagel's work [24] has already been covered earlier while discussing object-order and image-order techniques, but it is worth revisiting their approach here. To re-summarise, they presented an approach that uses volume bricking, with the aim to use a brick once using an "advancing ray front". Unlike Hsu's [23] approach, they only need to keep one copy of the screen buffer as the ray segments are inherently ordered front to back.

Koo et al. [83] extend work done by Yagel and Kaufman [84] by using templated to traverse a volume in object order creating a stereo image in a single pass. Two templates are created for each view and are both applied to a voxel 'cell' allowing for efficient voxel usage. At the time, their experiments showed that it was faster than ordinary ray-tracing twice, once for each eye, with comparable image quality but slower than standard shear-warp algorithms, again run once for each eye, but with much better resulting quality. There are, however, two notable drawbacks to this approach. Firstly the algorithm requires three copies of the volume, each transposed to be aligned with

the three orthogonal axes XYZ. Secondly, perspective projection with this method was, at the time, not supported. Both of these issues were being investigated as part of their future work.

Wan et al. [85] present an approach to rendering volumetric scenes using information generated from the left-eye to effectively splat the pixels from the left-eye on to the right eye and use accelerated ray-casting for holes remaining in the right-eye caused by occlusion. Their algorithm works for perspective projection for two cameras, left and right eyes, that reside on the same horizontal plane meaning that each scan-line in the left image impacts the same scan-line in the right image. The paper mostly focuses on opaque volumes which results in left-eye pixels having the same intensity as their re-projected right-eye counterparts. As such, they note that their algorithm works best in cases where pixel intensity is not view-dependent. However, it can be modified to accompany lighting models and transparency by partly ray-casting re-projected pixels, incurring some performance penalty.

There has been previous work done presenting volumetric data in a medical operating environment on an auto-stereoscopic display [86, 87] in which it is noted that allowing the operator to move freely in space while still presenting them with a parallax-correct display from multiple angles can prevent equipment contamination. Using DVR as the method to generate these different views is a highly computationally and memory expensive task, especially when the data is streaming and may require a full-redraw of the volume.

Multi-view Rendering

Stepping up from two-view stereoscopic volume rendering to N-view multi-view rendering requires additional consideration. These issues are then compounded by the fact that when multi-view targets arrived on the scene, the volume size complexity had already increased to the point of making it difficult for even single-view DVR. Nonetheless, there has been some literature in the area, although not a considerable amount.

Hübner and Pajorola [88] presented a single-pass rendering approach that is based on texture-based view-aligned slice rendering. Their method makes use of knowledge of the wavelength selective pattern that some auto-stereoscopic displays use and

effectively runs a rendering pass based on the component offsets in this pattern mask. In this case however, since slices are parallel to the centre-view camera plane, the more offset a view's camera is from the centre, the lower the sample rate would be through the volume, as the sample offset is applied parallel to the the aforementioned plane.

Agus et al. [1] present a full system for rendering and displaying volume data on a projector-based light-field display. Their paper mostly focuses on the translation between rendering space and projected-pixel space, utilising a relatively standard volume ray-casting algorithm for the actual rendering which is run once per view. In their example the view consists of 96 angles, or views, which each has a resolution of 320x240. It is mentioned in the paper that the scope did not include acceleration techniques such as data-structures but that they may be employed. To accomplish near interactive rates they use relatively low-frequency samples with a mipmapped volume, and batch 16 views at a time. They note this presents artefacts when rotating the volume or changing the transfer-function but also remind us that due to the auto-stereoscopic nature of their display system, once all batches are complete the viewer may move freely independent of refresh-rates.

Guitián et al. [3, 70] adapt stack-less ray-tracing techniques in an out-of-core rendering environment, coupled with a spatio-temporal refinement process, to render large (4096^3) volumes at interactive rates on a projector based light-field display. Their method makes use of maintaining a view-dependent spatial structure on the CPU with feedback from previous frames rendered on the GPU. In addition, not all pixels in a frame are rendered at once. They use a lattice structure to render 1 in every 4 pixels in a square group at a time, thus converging on a full image in 4 frames in the static-view scenario or presenting a blurry but acceptable set of images in the dynamic-view scenario. Their system runs on an cluster of 18 computers with CUDA enabled GPUs generating 4 images each, implementing a modified version of their previous single-pass GPU renderer [2]. Each projector view is rendered separately from the others. While they achieved good interactive rendering times their approach is only applicable to relatively large-scale systems after pre-processing the volume to create the octree structure, which they note takes over 12 hours, making this an unlikely candidate for real-time datasets.

Kwon et al. [89] apply a volume rendering approach to create multi-view elemental images for use with an integral imaging 3D display, although the rendering

techniques used in this paper are relatively standard.

Battin et al. [90] devise a method to determine the view from which a particular lenticular display pixel's chromatic components are derived from. In effect, this allows them to render the entire light-field for the display in a single pass, rather than rendering the data N-times for N-views and compositing the results in a final image, utilising the OptiX engine to perform DVR. In their work they don't investigate time-varying data.

Image-Based Rendering (IBR) techniques have also been used for multi-view rendering of volumes, specifically targeting light-field rendering. Recently Bruton et al. [17] and Martin et al. [18] evaluated — and proposed approaches that use — Convolutional Neural Networks (CNNs) as a method for synthesising novel views from 'anchor' views. These approaches can be extremely fast and are an interesting strain of work, however due to memory constraints on GPU, employing a CNN model that uses the actual volume can greatly limit the remaining space available in VRAM for said volume. It is the opinion of this author that IBR techniques are orthogonal to full volume ray-tracing.

2.4 Motivations

The background has now been presented and it should hopefully be clear where there are gaps in the literature. In this section a very brief summary of the background is given, with explicate outlining of these research gaps.

Performance Evaluation of GPU Texture Memory

While not a particularly novel area of research, the current literature of GPU performance benchmarking is quite sparse. There are high-level architectural GPU white-papers, and there are a few very low-level micro-benchmarking papers, but there is a lack of investigation for specific application oriented benchmarking. In order to have a solid platform upon which the rest of the work in this thesis is built, an analysis of key GPU memory components is required, specifically targeting texture memory in both the 2D and 3D texture domains, presented in chapter 3.

View Dependant Scheduling

One of the most prevalent output-sensitive optimisations that can be implemented for DVR is ray-guided streaming of bricked volumes. This is generally implemented as a feedback loop between the CPU and GPU where the GPU samples along the ray, marking bricks that are not paged into VRAM as needed. The CPU then reads this 'needed' list and pages them onto the GPU, restarting the sampling. This is repeated until a full image is generated. While this is output-sensitive with respect to both view-point and transfer function, the feedback loop makes a fluid pipeline from I/O to CPU to GPU impossible. Therefore, part of this work investigates sharing the view-dependant culling work with the CPU as a straight-through pipeline, presented in chapter 4.

View Dependant Scheduling for Light Field DVR

In addition to reasons mentioned above, view dependent scheduling can have another significant advantage — optimised cache usage. This becomes ever more important when the same regions of the volume are projected to multiple screen 'tiles' simultaneously. This is a very pertinent problem for light-field rendering, where the volume will be rendered the screen or screens at different view-points in a single frame. Therefore, scheduling data to be used in a cache efficient manner to target light-field rendering is an obvious extension of view-dependent scheduling. Research on this topic is presented in chapter 5.

BVH Adoption and Clustering for GPU DVR

With the presence of larger and larger volumes, ESS is a go-to optimisation for DVR. ESS strategies almost always depend on some sort of acceleration data-structure to assist jumping over empty regions or analytically accumulating for constant-space. The most popular data-structure for ESS has generally been octrees for their inherent compatibility with regular-grid dense volumes, while BVHs have been avoided for build times and traversal branch complexity on GPU, with only even little literature for DVR BVHs on CPU. With the emergence of BVH hardware on consumer GPUs in the form of Nvidia's RTCore in the Turing architecture accessible via the OptiX ray-tracing

API, it has come time to re-evaluate BVHs as a form of ESS data-structure for DVR. This research is presented in chapter 6.

Chapter 3

Performance Evaluation of GPU Memory Components

3.1 Goals

The main focus of this thesis is rendering volume data on consumer grade GPUs. One of the major bottlenecks in rendering on GPUs, especially for volume rendering, is memory performance. Getting data from source — hard disk or network storage for example — to screen traverses almost every part of the memory hierarchy on both the GPU and the host machine. Source to host and host to GPU optimisations generally take the form of either output-sensitive filtering [5, 51, 49] or compression methods. The performance of these components is generally well documented and easily benchmarked.

On the other hand, the performance of the memory hierarchy when the data reaches the GPU is a little more opaque and less documented. Understanding how texture memory performs on a relatively low-level but application-oriented basis is vital when formulating new approaches for rendering volume data. The purpose of this chapter is to give a platform upon which decisions are made in later chapters. Clear performance penalties and advantages for texture memory which are not documented in GPU reference manuals are shown. 3D texture benchmarking shows interesting results which imply sophisticated memory layouts that aim to exploit spatio-temporal coherency in shaders.

	Intel Xeon E5-1620 v3	Nvidia Quadro K2200
Cores	8 Logical (4 Physical using HT)	640 Cores (5 SMM with 4 groups of 32)
Shared Memory	-	96KB per SMM
L1 Cache	32KB D + 32KB I per Core	24KB Unified per SMM
L2 Cache	256KB per Core	2MB shared over all SMMs
L3 Cache	10MB shared over all Cores	-

Table 3.1: Comparison between Intel Xeon E5-1620 v3 and Nvidia Quadro K2200.

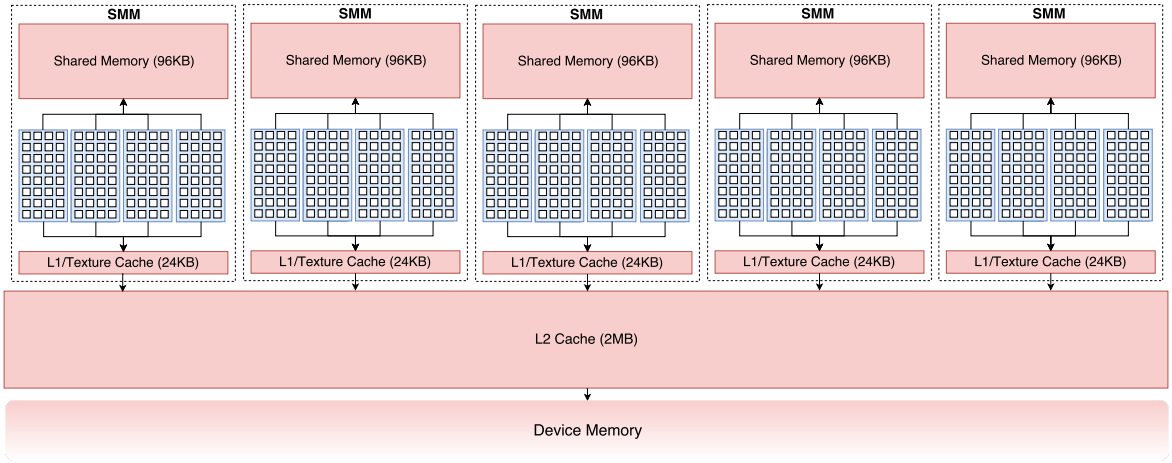


Figure 3.1: Overview of the memory cache hierarchy in the used Nvidia Quadro K2200. Note that shared memory is not included in this image.

3.2 Background & Related Work Recap

In this chapter the focus is on understanding the memory components of the GPU in depth. Specifically this work looks at one of the main GPUs used in the project, the Nvidia Quadro K2200. In general, CPUs and GPUs differ most in terms of their computing capabilities. The CPU used in this project is an Intel Xeon E5-1620 v3. In table 3.1 the difference between the computation and memory capabilities of both the Intel CPU and the K2200 is shown. It is important to note that the Shared Memory of the K2200, and Maxwell architecture in general, is not part of the cache hierarchy and is considered a separate module, much like a scratch pad. This architecture can be better understood by referencing figure 3.1 which visualises the memory placements of the K2200.

Before beginning to come up with a memory-efficient solution it is important to understand the penalties of the memory hierarchy in the Quadro K2200, and in particular how the textures and the texture cache interact with each other. From the outset there are two obvious figures that need to be determined:

- L1 and L2 cache-hit latency
- L1 and L2 cache-miss penalty

However, in addition to these, how the timings relate to actual texture memory should be understood. For example, a 2D texture lookup may only require 4 texels if using bi-linear interpolation. It follows that some texture memory layout could be used to group neighbouring texels close-by in linear memory. This differs greatly from a 3D texture lookup where 8 texels are needed for trilinear interpolation and the layouts required may be more complicated.

These experiments are based on micro-benchmarking techniques [91, 44] to help us understand in greater detail the exact performance characteristics of the K2200. In 2010, Wong [91] presented benchmarking techniques to profile segments of the entire GPU, building an understanding of the hardware from the ground up. This was achieved by running a series of small experiments testing small sections or instructions of the GPU a large amount of times to determine their latency. From initial results they build more complex experiments, using their prior information to more accurately estimate the actual characteristics of the complex routines.

In 2017, Mei [44] built upon the work of Wong [91]. Whereas Wong [91] used averaging as a technique for determining latencies of their micro-benchmarks, Mei [44] timed individual lines of code or operations, taking care to compensate for the overhead that this might present in terms of pipeline latency. It is this method which the following texture memory experiments are based on.

3.3 2D Texture Patterns

3.3.1 Methodology

As a first step this chapter looks at the behaviour of texture memory and the texture cache when referring to 2D textures. The goal is to determine what data resides in

both the L1 and the L2 when a texture reference is made. To evaluate the lookup latency of a point in the texture, an experiment was carried out as followed:

1. Random data was created to fill two textures, each larger than the L2 capacity in terms of raw data.
2. Texture A is transferred to the GPU.
3. Texture B is transferred to the GPU, flushing texture A from the L2.
4. An initial texture lookup at point I occurs on texture A.
5. A warp-timer is started
6. A second texture lookup occurs on texture B at a different point J.
7. The warp-timer is stopped and the time recorded.

In order to evaluate which texels are cached when an initial lookup at point I is made, all steps in this experiment are re-run for all points in the texture, changing point J.

3.3.2 Results

This experiment results in the graphs shown in figure 3.2, which shows the lookup latency for each point J in a 2D texture after an initial lookup at a point I. In this case the texture size was set to 1024^2 with a single 32-bit channel. and the initial lookup, point I, was set to [510, 510] in the left graph and [720, 720] in the right graph. It can be seen that there are hard-defined areas in both graphs that have a much lower lookup-latency than the rest of the texture. It is interesting to note that the area of these regions seems to be constant, yet made of smaller sub-regions 512x128 texels making up 256KB of data. Four of these regions together make up exactly 1MB of data, which appears to be brought into the L2, of size 2MB, with the initial texture reference.

In this example, the size of the sub-regions fits rather well with the overall texture size. However, this may not always be the case, and texture sizes may not always be a power-of-two. The same experiment as above was run to determine the

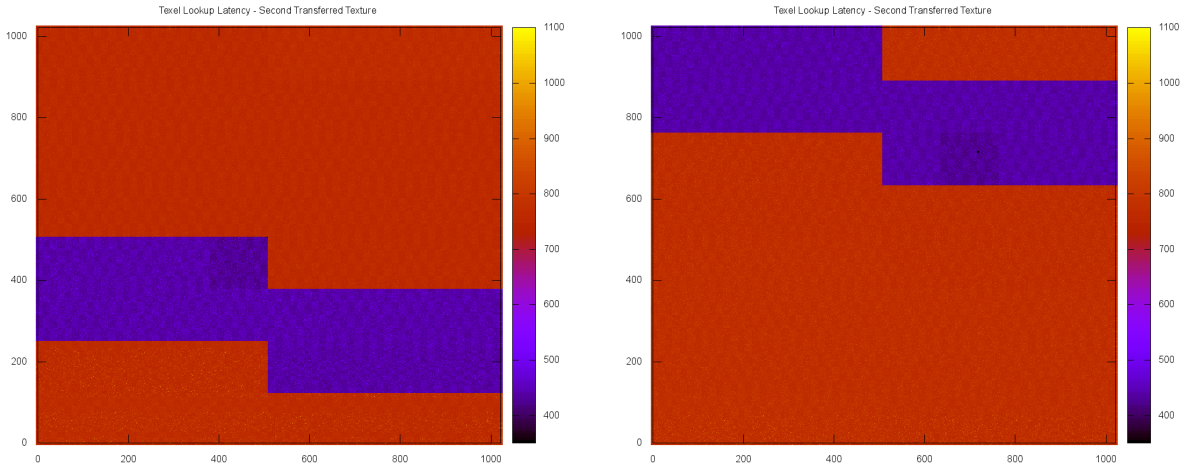


Figure 3.2: Texel lookup latency for a 1024^2 texture with an initial lookup of X:510 Y:510 (left) and X:720, Y:720 (right).

sub-region sizes when a texture of size 725x725 was used. This size was chosen to fulfil the 'flushed texture' nature of the experiment as it just overflows the L2 cache size.

The results of this experiment are shown in figure 3.3. Interestingly, the sub-regions are still apparent. While they can still be considered to be 512x128 texels, they begin to 'wrap' in the texture, overflowing to areas above. In addition to this, as hinted in the lower left image in the previous experiment shown in figure 3.2, the start of textures seem to only consist of the rather odd amount of three sub-regions of 256KB, with following regions being made up of four.

3.4 Exact Latencies

In order to get answers for the two important sets of figures outlined above at the beginning of this section, careful and methodical experiments must be devised, especially when dealing with many extraneous factors such as compilers, drivers, schedulers, etc.

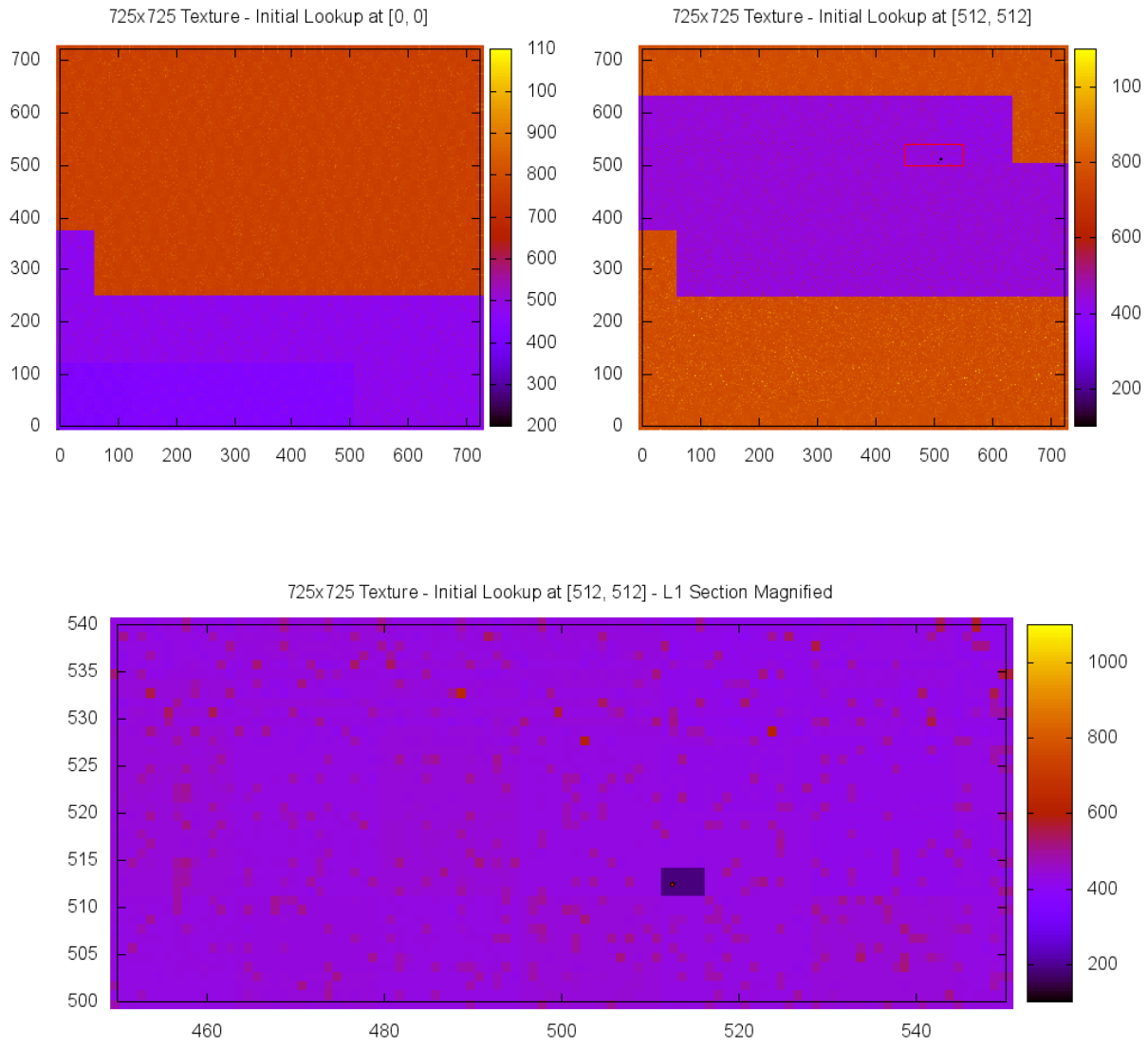


Figure 3.3: Texel lookup latency for a non-power-of-two texture (725^2) with an initial lookup at [0, 0] (top left) and [512, 512] (top right). Note the difference between the low-latency areas in figure 3.2 and in these charts. Also note the "inter-locking" appearance of the low-latency areas in the left and right graphs. Also presented is a magnified region, highlighted in red in top right, showing a much smaller area of substantially lower latency lookups representing L1 hits (below).

3.4.1 Methodology

As an initial probe, experiments testing the latencies of 3D texture lookups were devised to provide us with baseline figures for L1 and L2 hits and misses. In these experiments the aim is to determine the amount of warp-cycles and *the accuracy of this figure* for:

1. **Cycles used in calling clock function.** This is extremely important as this figure will be prevalent in figures obtained for experiments 2-6 and as such must be subtracted from the final results.

2. **First texture lookup.** This figure, along with its standard deviation should give us the initial L1 cache penalty after transferring the referenced texture to the GPU. Note that it is assumed that the texture travels through the GPU's L2 on the way to DRAM.

3. **Second texture lookup in same location.** Making a subsequent texture reference in the exact same location should give us an exact reading on the cycle count for an L1 hit.

4. **Second texture lookup one texel in x-dimension.** As can be seen in figure 3.3 (bottom) there is a small region of texture brought into the L1. This verifies that the same is true for 3D textures.

5. **Second texture lookup one texel in y-dimension.** This verifies much the same as the previous step.

6. **Second texture lookup one texel in z-dimension.** Unlike 2D textures, adding an additional dimension leaves us at a loss of how much data is brought into the L1 in the z-dimension. This figure should help us determine such a number.

In order to get robust numbers for these experiments the following precautions were used:

- All experiments were run 10,000 times to get an average and a standard deviation.
- The GPU was synchronised immediately before and after the kernel was called to prevent accidental overlapping kernels.

	Timer	First Lookup	Same Location	X+1	Y+1	Z+1
Warp Cycles	6	880	190	190	190	420

Table 3.2: Latency times of 3D texture lookups.

- Only one thread ran on the GPU to prevent interference.
- The compiler was given debug and no-optimisation flags.
- The critical sections of the code were written in PTXISA to explicitly specify steps.
- The compiled PTX was then verified to make sure that the lookup alone was surrounded by the start and end timer calls.

3.4.2 Results

The results of these experiments are shown in table 3.2. It can seen that there is a figure for timer cycles, which was step 1. This figure is then used in the rest of the table to adjust the figures in the row "Adjusted". As expected, a relatively large cycle count for the initial texture (experiment 2) lookup at 663 cycles is observed, however this has an interestingly large standard deviation. For experiments 3, 4 and 5 the exact same cycle latency of 158 cycles with a very low standard deviation is seen.

The interesting figure in this table comes from experiment 6, which is a texture lookup one texel in the z-dimension. This equates to 'not-quite-a-hit' but also 'not-quite-a-miss'. This leads us to believe that, since sample interpolation is active, one layer of texels needed for the interpolation were present, but the second later was not, resulting in a 'half-miss'.

3.5 3D Textures and Sampling

It is important to appreciate the effect that using sub-cache sized bricks makes on the GPU memory hierarchy. This is a vital statistic that determines much of the decisions made in later sections of this work.

3.5.1 Methodology

In order to determine these figures an experiment was devised to evaluate the cache statistics of three methods; standard DVR, ray-guided bricking (without sub-sampled level-of-detail), and load-balanced brick-order ray casting. To properly determine the cache efficiency of volume data usage, the output buffer was disabled, but every sample accumulated to a dummy output variable in order to prevent automatic optimisations compiling out important texture loads.

3.5.2 Results

In figure 3.4 it can seen that a 3D texture is effectively cached as a stacked set of 2D textures. This covers how data is cached, but it doesn't indicate how the data is paged. In section 3.4 an experiment was presented to determine the latencies of every texel in a 2D after an initial lookup. It should have been made clear that there are effectively four states in which a texel can be in:

- L1 Cached
- L2 Cached
- Paged
- Not Cached or Paged

It is relatively easy to understand the paging structure for a 2D texture, however when this is extended to the third dimension, what would be the expected impact? Since it appears that the 3D texture is effectively a set of stacked 2D textures, will the first few layers of a volume be paged, or something entirely different?

The results of this experiment, the method of which was the exact same as previous sections with an additional dimension, are seen in figure 3.4. Contrary to the initial assumption, the data are not paged in such a straightforward manner. Instead, there are multiple layers having the same parts paged. It is also interesting to note that this page size appears to be consistent with the 2D texture page size.

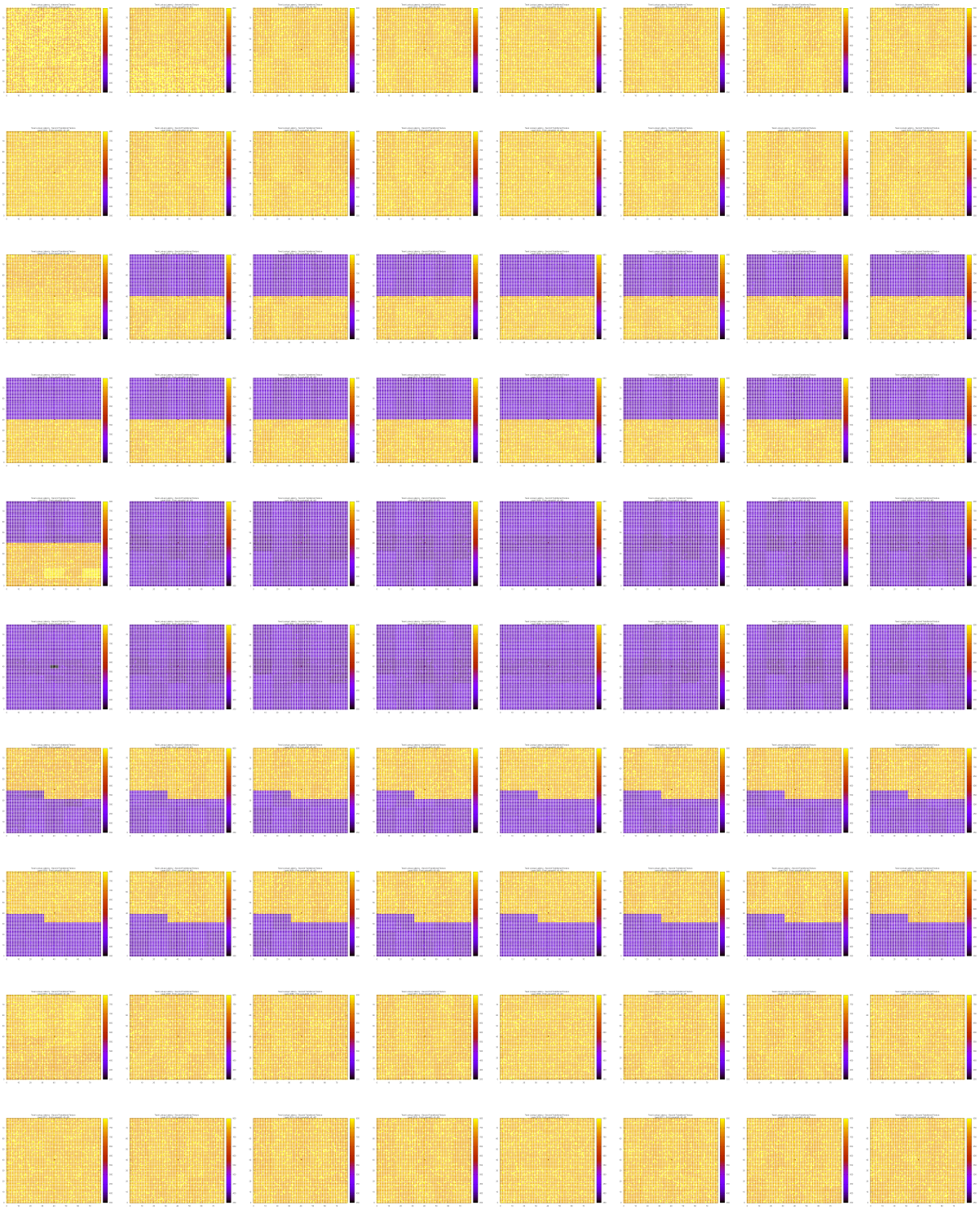


Figure 3.4: Latencies for each layer of a 3D texture given an initial lookup at [40, 40, 40], which is in layer 40, visible in the first column, sixth row. Note the interesting paging pattern.

3.6 Conclusions

In this section exploratory work on the characteristics of texture sampling on the Nvidia Quadro K2200 was shown. This was done specifically with a view to understanding how 3D textures work on the Maxwell and Turing architectures, with the goal of being able to make better decisions on how to structure volume data for rendering. The results shown that making inefficient accesses to a texture or set of textures can result in large miss penalties. As such, care must be taken to make efficient use of the caching techniques provided.

An obvious method to exploit cache coherence — especially the shared L2 cache on the Maxwell and Turing architectures — is to limit the sampling to a cache-sized region of the volume for an extended period of time. This should be seen as a priority, especially when considering the limited capacity for swappable thread groups or warps — in order for more work groups to be instanced on the GPU, older work groups need to finish up. Imagining direct volume rendering as a group of rays moving in a set of tiled ray wave fronts, one can see the potential issue of neighbouring wave fronts ending and beginning temporally far apart, thus the later wave front may have to re-page and re-cache the data needed for its samples. In later chapters this observation is used as the target for better performance via volume bricking and aggressive scheduling.

Chapter 4

View Dependent Scheduling & Load Balancing

This chapter presents work done on attempting to make efficient use of the GPU memory hierarchy when rendering an non pre-processed volume by sub-dividing it into bricks, generating an ordered list of the contributing bricks in front-to-back order, rendering the bricks in any order and compositing the bricks in the previously determined order. It can be shown that by using this method the bricks are used in an efficient manner in terms of memory, however results also determine that there are hindering drawbacks which nullify the advantages.

4.1 Goals

Following on from evaluations performed in chapter 3 it is clear that optimising a renderer to make better use of spatio-temporal cache coherence on the GPU should be a priority. The different levels of the memory hierarchy all have various performance traits, but area with potentially the best trade-off in terms of size-available and performance benefit is the L2 cache.

Additionally, when considering the volume rendering pipeline on a whole, the entire system should be used to the best of its ability. To be more specific, when data is being rendered on the GPU, the CPU should be doing work, and vice-versa. The obvious solution to enable this is pipelining DVR so the CPU queues up work for the

GPU to do.

Taking both of these priorities into account, a potential solution appears where the CPU performs the output-sensitive filtering and scheduling of data that is optimised for the GPU's L2 cache, and GPU just consumes and renders said data. More importantly, the GPU should be presented with all the data and information it needs to render the frame without re-communicating with the CPU. This then allows for a pipelineable output-sensitive approach for DVR.

4.2 Background & Related Work Recap

Volume rendering in not a new field and has been broadly researched. This section gives the reader an overview of work relating to the specific aspects of volume rendering presented in this paper. For a relatively recent review of a large section of volume rendering research the reader is directed to a 2015 State-of-the-Art paper by Beyer et al. [92]

Knittel [45] presented a complete DVR system that focused on the low-level arrangement of memory, while also making use of hand-optimised assembly and SIMD instructions. While Knittel's work is useful for single frame volumes, the approach may not be appropriate for a time-varying dataset due to the memory footprint proposed.

In some volumes it may be the case that there are relatively large sections that are identical or very similar which, when the transfer function is applied, will have no impact on the final image. While Knoll et al. [58] follow in the steps of ULTRAVIS by observing that with highly optimised code the CPU can outperform the GPU when rendering large volume datasets, they also make use of a BVH for empty space skipping. However, whilst this approach can be justified for static datasets the overhead in terms of pre-computing the BVH would be considered a limiting factor.

Fogal et al. [50] bricks the volume and begins rendering — without data — on the GPU. During the rendering stage different level-of-details are determined for each brick and the sampling rate is adjusted accordingly. When a ray encounters a brick that is not present in device memory the absence is recorded and eventually a list of the required bricks is communicated back to the CPU, resulting in a feedback loop and an idling CPU or GPU.

Bricking [4, 45, 21, 49, 50, 58] involves breaking the large volume into smaller

sub-volumes and only transferring and rendering the data of the bricks that rays interact with. While there would still be un-used data in some of the bricks (for example, only a single ray hitting the edge of a brick or rays dispersing the deeper they go into the volume) it can still give a coarse representation of the camera frustum.

However, when dealing with rendering algorithms that require interpolation between voxels to get an accurate sample, the renderer must either use multiple bricks for a single sample or pad the brick with duplicated voxels.

In this approach the intention that an SMM only work on a single brick at a time, implying that the voxels at the edge of a brick are duplicated. It should be noted and well understood that this duplication can be quite drastic when the brick size decreases. As seen in figure 4.1 when bricking even a 512^3 volume the memory overhead for the entire volume increases dramatically.

This overhead in particularly emphasised when the padding not only has to accommodate voxels for interpolation but also gradient calculation for lighting techniques.

There has been work done to mitigate the effects of these ghost voxels by using previous brick data to generate this duplication on the fly rather than transferring it, saving on transfer bandwidth [93].

4.3 View Dependent Scheduling & Load Balancing

The aim of this approach is to load balance across the CPU and GPU. To achieve this a pipeline is created that allows for the CPU to prepare a frame while the GPU is rendering the previous frame.

This section outlines the different sections of the pipeline starting with the brick determination on the CPU, brick rasterisation on the GPU and load-balanced rendering on GPU. These overlapping processes are visualised in figure 4.2.

4.3.1 Brick Determination

In order to load-balance the CPU and GPU, the first step performed is the determination of which bricks the GPU required to render the volume on the CPU. The algorithm is designed to be run in-parallel to rendering, meaning that once the algorithm has de-

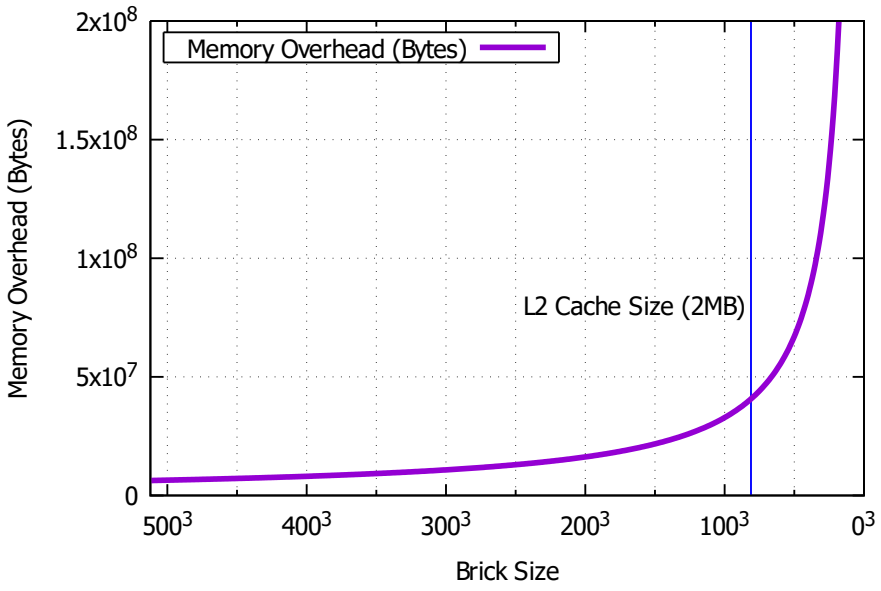


Figure 4.1: When determining the ideal brick size, care must be taken that redundant data used for ghost voxels (if used) does not become overwhelming.

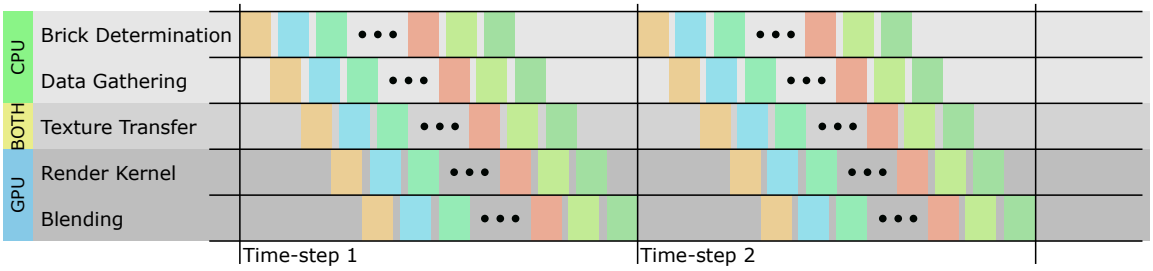


Figure 4.2: Overlapping processes as part of the rendering pipeline on both the CPU and GPU.

termined the first brick to be rendered, the information and data is sent through the pipeline to begin the rendering of said brick.

While complete out-of-order rendering is possible the memory requirements of this would be quite severe as each brick would render to its own buffer. Bricks closer to the camera take up a considerable portion of the screen space, leading to hefty requirements in terms of buffer space.

To address this issue a work tree is created implicitly when sending jobs to the GPU. Bricks are always determined from nearest to the camera working away through the volume.This is accomplished by first finding the closest brick to the camera that is in screen space. the neighbours of this brick are then added to a search stack. The stack head is popped and screen space tested by projecting the eight corner points to a tile region. If this tile incorporates part of the screen buffer, it is added to the render queue. Neighbours of this brick are then added to the search stack. The next brick on the search stack is popped and the loop is repeated. This process is outlined in algorithm 1.

Algorithm 1: Brick determination algorithm. Note that when a brick is pushed to *renderList* rendering of that brick can begin as soon as the brick data has been loaded.

```
1  Brick b = Closest Brick In Frustum;
2  List searchList.push(b.neighbours());
3  List renderList.push(b);
4  while searchList not empty do
5  |  b = searchList.popHead();
6  |  List neighbours = b.neighbours();
7  |  for Brick n in neighbours do
8  |  |  if not n.checked then
9  |  |  |  if screen.willTouch(n) then
10 |  |  |  |  renderList.push(n);
11 |  |  |  |  searchList.push(n);
12 |  |  |  end
13 |  |  |  n.checked = true;
14 |  |  end
15 |  end
16 end
```

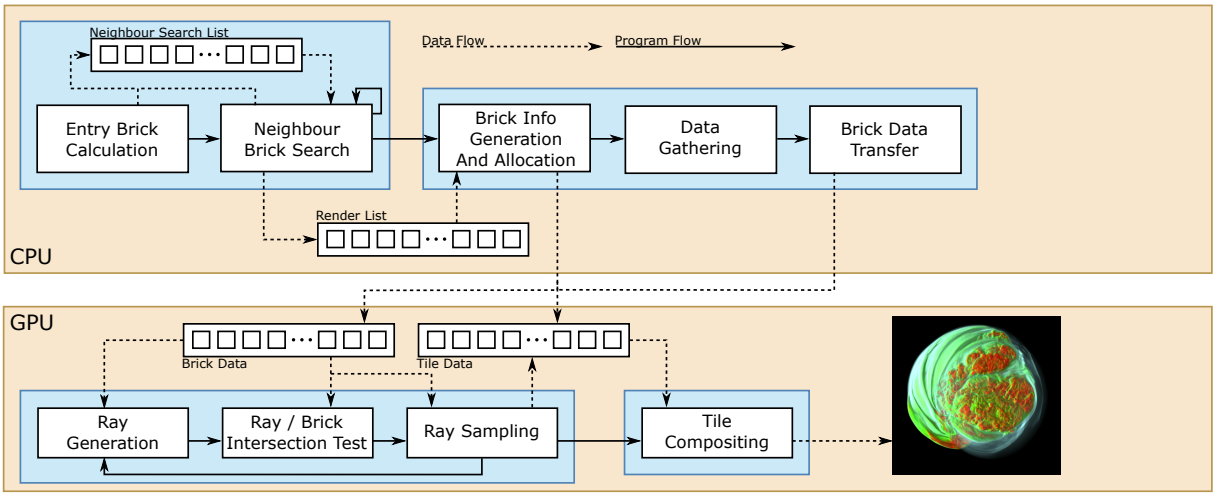


Figure 4.3: Flow of proposed view-dependant algorithm.

4.3.2 Empty Space Skipping

As the proposed approach depends on a previous step that will read all voxels in a brick before parking them to be transferred to the GPU and rendered, it can employ an ESS technique to reduce computational load on the GPU. By reading all voxels in a brick the algorithm will know the minimum and maximum values in the voxel chunk. Much like performing empty space skipping on regular voxel blocks [94], DVR can simply skip rendering the brick if it knows the applied transfer function will have no effect on the resulting image.

4.3.3 Brick Compositing

While the working-set of bricks are determined and rendered has now been covered, this sections show how to make good use of the computational resources to composite the bricks in an efficient manner. In listing 1, the algorithm to generated an ordered list of the working set of bricks is shown. Using this list, final-image screen-space is divided into tiles, which are referred to as screen-tiles to make a distinction from brick-tiles. Each of these screen-tiles are the work item of a CUDA block, or effectively each SMM takes a section of the final image to work on.

Each block traverses the ordered list from front to back, first testing each brick-tile in the list for contribution to the current screen-tile. If there is no contribution to

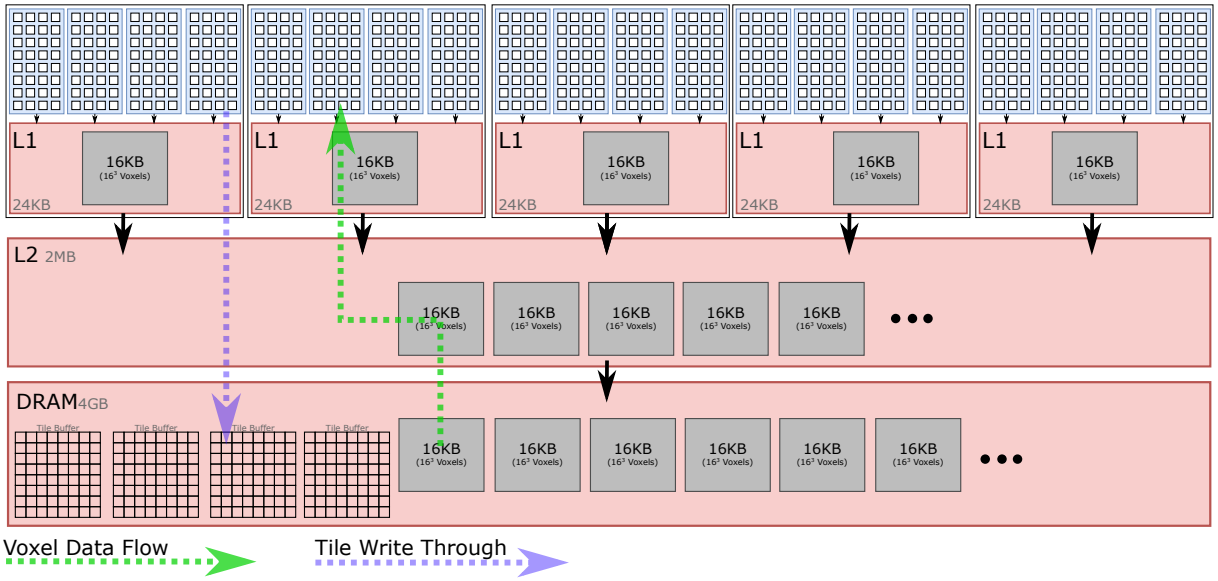


Figure 4.4: Data flow of proposed view-dependant algorithm.

the current screen-tile, the brick-tile is skipped and the CUDA block moves on to the next brick-tile in the list, otherwise the brick-tile is composited onto the final image. A visual representation of this method is provided in figure 4.5. This method ensures that a correct final image is composited.

4.4 Implementation & Results

In this section an outline of the implementation specifics is given and present the results of performance profiling tests.

4.4.1 Implementation

The code for this method was implemented in C++ using CUDA in a cross-platform capable project. While the project itself is cross-platform, most of the results are collected with the project being built on the Linux operating system, specifically Ubuntu 16.04LTS. The system is a Dell Precision Tower 5810 with hardware consisting of an Nvidia Quadro K2200 GPU, an Intel Xeon E5-1620 v3 CPU, 16GB of RAM and a standard 1TB HDD. Bandwidth and cache results were obtained using the Nvidia Vi-

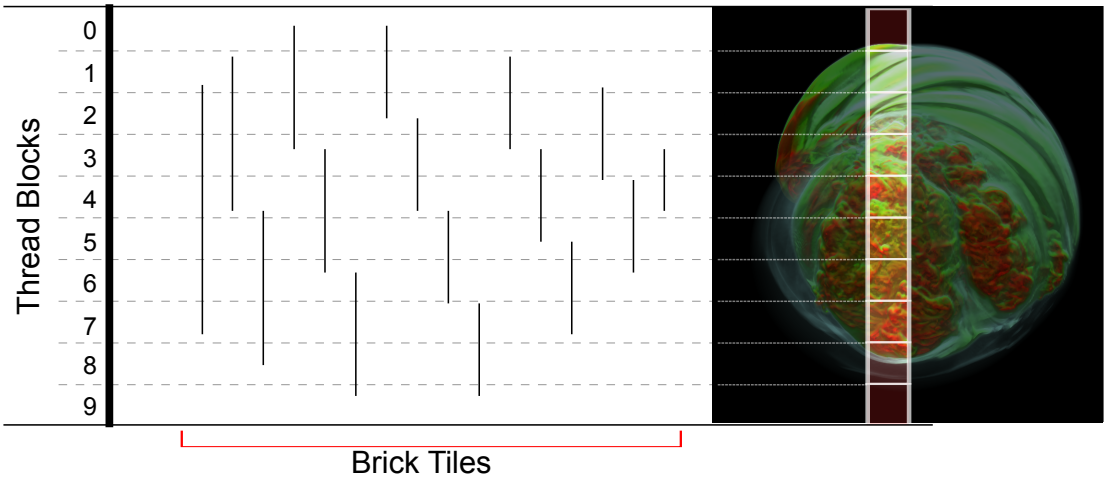


Figure 4.5: Screen-tile based compositing. Kernels are launched in a screen-tile based order to composite the brick render-tiles in front-to-back order. This ensures that each block uses a brick-tile once and only once.

sual Studio plug-in NSight on Windows and the Nvidia command line tool nvprof on Linux.

4.4.2 Memory Bandwidth

Table 4.1 shows the memory performance statistics of the brick-order ray caster versus a standard DVR and ray-guided bricked ray caster, without an output buffer. It is interesting to note that although the L2 hit rate for brick-order ray casting is substantially lower than its counterparts, the data transferred between the L2 and L1

	Standard DVR	**Ray-Guided Bricking**	**Brick-Order**
DRAM to L2	139.73 MB	322.85 MB	135.33 MB
L2 to L1	1.37 GB	2.13 GB	**0.45 GB**
L2 Hit Rate	89.6%	92.3%	84.1%
L1 Hit Rate	70.5%	73.7%	88.9%

Table 4.1: Memory bandwidth statistics for different methods of volume rendering on an Nvidia Quadro K2200 **without an output buffer**. Note the drastic reduction in bandwidth between the L2 and L1 for the Brick-Order ray casting method.

is drastically reduced. Interestingly the ray-guided bricked ray caster can be seen to thrash both the L2 cache considerably, drawing this conclusion from the rise in data between DRAM and L2.

However, when the output buffers are introduced, the memory hierarchy tells a different story. The author found that maintaining the tile buffers for the bricks thrashed the cache substantially. Even using shared memory to save a patch of the tile buffer proved futile, as the available patch area is quite small.

4.4.3 Overhead

In addition to tile buffer drawbacks, the computational overhead for scheduling on both the CPU and GPU nullified any cache performance gains. With small bricks, the cost of brick-determination on the CPU is quite high although in hindsight a parallel method could possibly be devised. Regardless, this time is still less than the GPU render time and, as such, they could be overlapped.

On the GPU — because of the scheduling parameters and brick-tile information needed — the approach began to hit a thread-occupancy issue on the GPU. With more information needed for a kernel to render a brick to a tile, more registers were needed to keep track of this information. As such, the achieved occupancy begins to drop.

4.5 Conclusion

In this chapter, the scheduling of brick data was implemented and investigated for single-view sort-last DVR. Unfortunately the results are not satisfactory in terms of rendering times. While brick re-usage count is low, the scheduling overhead on both the CPU and the GPU nullifies the gains. In addition to this, a sort-last approach is difficult to maintain on the GPU due to the memory footprint needed for the brick tile buffers.

While this approach does not appear to suit single-view rendering systems, there may be room to extend this solution into multi-view rendering with the possibility of spreading the overhead costs over all the views. In section 4.4.3 it is noted that the scheduling costs on the GPU for a single view is quite high. If this method was parallelised and modified to accommodate multiple views, the overhead might not scale

linearly. In addition to this, the drawbacks of tile buffers might be nullified by possibly re-using tiles that provide little or no information to other views.

Chapter 5

Light Field Volume Rendering

Direct Volume Rendering (DVR) of volume data can be quite a memory intensive task in terms of both footprint and cache-coherency. Ray-guided approaches may not be the best option when it comes to rendering the volume to multiple-views or light-fields at interactive rates, as regions of the volume rendered for one view may be overwritten in the on-chip cache only to be needed by another view at a later point. Additionally, many approaches are not suitable to streaming time-varying data due to pre-computation. This chapter presents an approach to schedule volume sub-data while minimising the footprint of sub-buffers needed, and pipe-lining the work load between CPU and GPU, showing that there is significant advantage to using such an approach.

5.1 Goals

For many streaming time-varying volume data applications — i.e 4D MRI, 4D ultrasound or scientific simulations — direct volume rendering (DVR) is the method of choice. It is often desirable for users to view these streaming data-sets in real-time while simultaneously moving the camera or changing the appearance of the volume using a transfer-function. Presenting the user with additional perceptual enhancements like parallax via auto-stereoscopic or light-field displays can give a greater understanding of the data [11, 87, 12]. Light-fields further enhance a user's perception of the scene by allowing the eye to accommodate and focus on any point, as shown with

near-eye light-field displays [80]. This serves the dual purpose of depth perception and preventing eye-strain by circumventing the vergence-accommodation conflict problem.

Much work has been done on rendering single-frame volume data in a ray-guided approach [49, 50, 4], and some work has been done on displaying single-frame volume data to a light-field display [3, 1]. These approaches are a good solution when data is static or updates without a real-time requirement [5] and transfer the exact view-dependent volume data needed by the GPU. However, these ray-guided approaches are not ideally suited for real-time rendering of streaming time-varying data due to the requirement of data-structure construction, level-of-detail sub-sampling and feedback-loops between GPU and CPU. Time-varying DVR has also been well researched [36, 33, 37, 35] but few papers focus on rendering to a light-field display. Approaches that do [86, 87, 90] are either quite simple or resort to down-sampled or image-based techniques to minimise latency.

Finally, when rendering a volume to a single-view, rays are generally closely coupled and sample the data with relatively good spatial locality, taking advantage of the shared L2 cache on some GPUs. This can change, however, when rendering a volume to a multi-view or light-field display when sets of rays from multiple different views can begin to thrash the caches.

Considering this scenario, a pipe-lined approach requiring minimal pre-processing with the ability to run on commodity hardware is desired. In this chapter a novel method is shown to make efficient use of existing GPU architectures to simultaneously project regions of a streaming time-varying volume to all views of a light-field display in one pass, exploiting the on-chip L2 cache, while sharing the scheduling load with the CPU in a pipe-lined fashion.

The proposed approach focuses on the performance of light-field DVR on current commodity hardware at a practical resolution in terms of both screen and volumetric data dimensions in comparison to an octree empty-space-skipping (ESS) approach. This is done by projecting sub-volumes that fit in the on-chip GPU L2 cache to all views before moving on to the next sub-volume, making better use of the cache, rather than each view sampling from the volume in different locations at different times. The approach exploits volume regions temporally close in render order to minimise the amount of intermediate sub-buffers to which bricks project. All of this is presented as a pipelined approach with the CPU and GPU working in parallel. In summary, the

contributions are the following:

- A hybrid approach for spreading the view-dependent working set determination between the CPU and GPU in a pipe-lined fashion to render reasonably-sized streaming time-varying volumes to a light-field.
- A method for minimising the amount of unique light-field sub-buffers needed to render sections of a volume out-of-order, reducing the complexity and memory-footprint of standard sort-last rendering.
- An evaluation of the proposed method showing significant performance increase in the render-kernel in comparison to an octree-based empty-space-skipping renderer as a result of improved L2 cache utilisation.

5.2 Background & Related Work Recap

This section gives an overview of relevant display technologies, their applications, advantages and disadvantages, with the aim of giving the reader an understanding of why multi-view or light-field displays are an important research area.

Light-Field DVR

Ruijters et al. [87] expand upon previous work [86] and present an application-oriented method to reduce latency between data-retrieval and display in a real-time system displayed on a lenticular auto-stereoscopic display. To achieve this they adjust the output resolution of the volume rendering dynamically and, in certain cases, render fewer views than presented on the auto-stereoscopic display, interpolating between rendered views to fill in the gaps on the display side. While this method works well for latency minimisation, rendering artefacts can be present in the interpolated images. Rezk-Salama et al. [95] convert large volumetric datasets into a light-field representation in an off-line pre-processing step, relying on spherical light-field parametrization and depth information. Because this is an off-line method, streaming time-varying data was not supported. Birklbauer et al.[96] present a combination of light-field and volume rendering to enable real-time interactive explorations of large static volumetric data sets, based on efficient light-field caching[97]. They use a spherical light-field

parametrization with virtual cameras uniformly distributed on a sphere which encloses the volume. Agus et al.[1] display volume data on a projector-based light-field display using fairly standard means of rendering but focusing more on image warping to present a final projected image. Guitian et al.[3] use rendering methods similar to Gobetti et al.[2] but displayed on a projector-based light-field display, targeting large static volume datasets. Battin et al. [90] devise a method to determine the view from which a particular lenticular display pixel's chromatic components are derived from.

Volume Data, Bricking & Data-structures

A common strategy used in volume rendering is to sub-divide the volume into regions or "bricks" and wrapping a data-structure like an octree [50, 49, 4, 3, 56] or a bounding-volume-hierarchy[58] on top. Construction of these data-structures and sub-sampled level-of-detail bricks consume valuable CPU time and are not necessarily possible in a real-time streaming volume environment.

Previous work that uses bricking as an approach for volume rendering has been covered in previous chapters — such as Knoll et al. [58], ULTRAVIS[45], Gobetti et al.[2], Crassin et al. [4], Fogal and Krüger [49], Fogal et al. [50] Hadwiger et al. [6, 5] and Liu et al. [51]. None of these works are designed to work for multi-view or light-field scenarios, requiring off-line pre-processing and sub-sampling steps, and do not take advantage of their bricked-structure for temporal cache-coherency.

Object-Order & Image-Order

There are two paradigms used for the rendering of bricks: 'Sort-first' or 'image-order' rendering refers to systems which render the bricks in-order to the screen-buffer, and 'sort-last' or 'object-order' rendering which renders the bricks out-of order or in parallel to sub-buffers and then compositing these sub-buffers in order to the final image [23, 22, 25]. Samanta et al. [27] presented a hybrid between object-order and image-order rendering, and the proposed approach makes use of this concept but at a much smaller system-scale and proximity, targeting the GPU rather than a full cluster. Since the proposed approach is a hybrid of sort-first and sort-last the aforementioned ERT drawback can be mitigated to some extent.

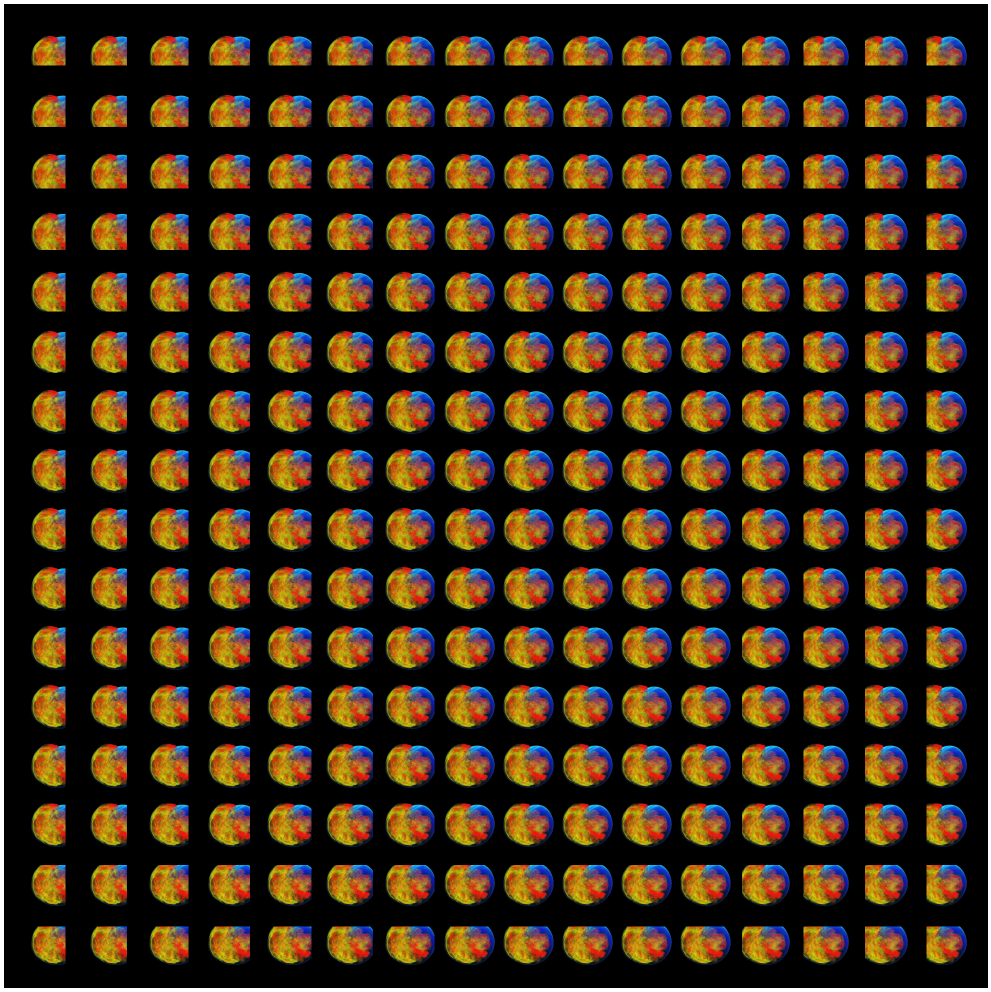


Figure 5.1: 16x16 light-field rendering of super-nova dataset. Note the slight parallax that can be seen across all the images. This parallax introduces complexity when determining the render-order of sub-bricks.

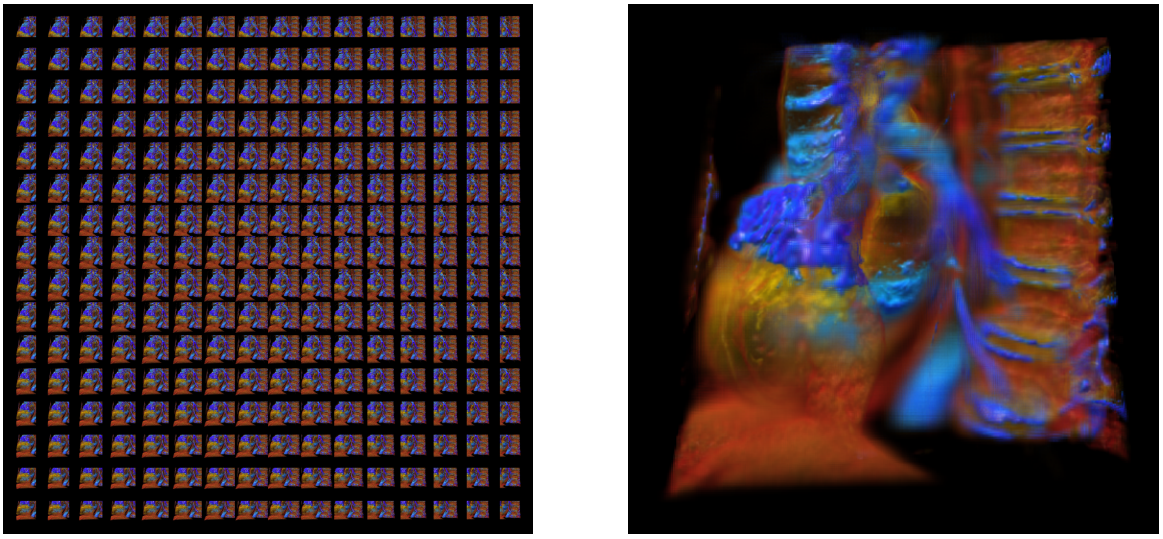


Figure 5.2: A 16x16 planar light-field (left) of a cardiac data-set, refocused (right) using the method outlined in [66], presented as a validation of the presented light-field implementation.

Streaming Time-Varying Data

Time-varying volume data [33] that has multiple frames representing individual time steps. Shekhar et al. [36] use a multi-planar approach to render streaming cardiac data by bricking the volume data. Zhang et al. [37] presented a simplistic 4D cardiac data visualisation synchronised to ECG signals. Noonan et al. [35] use predictive approach to estimate the change in bricks in a volume, cutting down on data transfer between the CPU and GPU, but introducing slight data distortions in doing so.

GPU Memory Hierarchy & Thread Scheduling

The GPU memory hierarchy is of particular interest in this work and it is vitally important to note the placement of the L2 cache in this architecture. For example, with 5 streaming-multiprocessors (SMM), the GM107 architecture provides us with 640 computational cores in total. A small unified L1 cache is provided to *each* SMM, and a larger 2MB L2 cache is shared among these 5 SMMs, followed down to the DRAM on the device referred to as device memory in both the documentation and this work. It is the shared L2 cache which this approach aims to exploit by projecting regions of

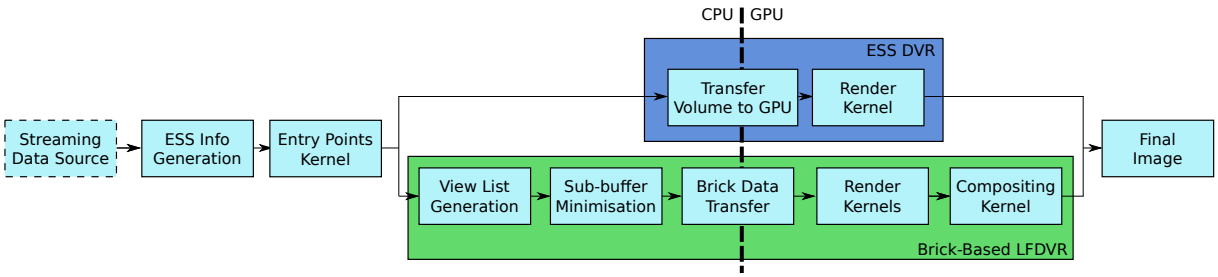


Figure 5.3: Overview of the pipelined approach to profile both the brute-force LFDVR and the bricked LFDVR. Note that the two methods share ESS info generation and entry/exit points step.

the volume to all light-field views before moving on to the next region, making better use of the temporal coherence in the L2.

While the computational capacity of the GPU is quite substantial, the amount of concurrent computation is limited by a few factors such as register or memory usage per thread. It is easy to consider the rendering of volume data as a set of rays being stepped along simultaneously, however due to the limited — albeit large — amount of threads available to fit concurrently on the GPU, in reality the volume is rendered in tiles, with the possibility of the rays in one tile shooting through the entire volume before the next tile begins. This leads to the issue where the L2 cache may be thrashed along the way, before the next set of rays starts and repeats the process. However, since the render kernel is forced to complete a sub-section of the volume before advancing to the next volume brick, improved cache coherency can be expected.

5.3 Pipelined Brick-Based Light-Field DVR

The core concepts of the method are:

- Projecting one sub-L2-cache-sized sub-volume brick to all light-field sub-aperture images before progressing to the next brick, essentially retaining the brick data in the cache for more efficient sampling access by forcing all GPU thread blocks to remain in the same volume brick .

- Since the render order of bricks may be different for each sub-aperture image viewpoint, each brick should render to an intermediate buffer called a sub-buffer. The

algorithm exploits bricks that can use the same sub-buffers as their neighbours to reduce memory footprint.

- Pipelining the working-set/render-order/sub-buffer determination on the CPU, and asynchronously rendering of bricks followed by sub-buffer compositing to get the final image on the GPU.

The pipeline is therefore structured as the following steps, and visualised in 5.3:

1. **CPU** Empty-space-skipping (ESS) information generation.
2. **CPU** Per-view view-dependent render-order list determination.
3. **CPU** Per-view brick sub-buffer amalgamation (minimisation).
4. **Both** Working-set data transfer.
5. **GPU** Per-brick render kernels.
6. **GPU** Sub-buffer compositing kernels to final image.

ESS data-structure generation has been covered in the literature extensively, and the reader can refer to section 2.2.6 for an overview of this. As such this chapter only expands upon view-list generation (stage 2 above) and the sub-buffer minimisation (stage 3 above) in the following sections.

View-List Generation

The rendering process begins by determining the entry brick for a given view. In order to generate a render list per-view there first needs to be *overall order* determined in which the bricks will be rendered. In this approach they are ordered by distance from the camera plane centre point. Note that bricks that are considered empty are *not* placed on this render list, and as such are now excluded from the working-set. With this information the per-view view-dependent render lists are generated in parallel on the CPU. These lists represent the order in which bricks need to be rendered *for a particular view* to produce the correct result. This method is the same as used in chapter 4, algorithm 1, except that is is now used per-view and executed in parallel.

Algorithm 2: Sub-buffer minimisation algorithm.

```
1  Function MinimiseSubBuffers (View[] views, Brick[] bricks)
2      for View view in views do
3          List brickList = view.brickListInOrder;
4          List subBuffers = {};
5          while brickList not empty do
6              b = brickList.popHead();
7              if not b.HasSubBufferForView(view) then
8                  SubBuffer s = new SubBuffer();
9                  subBuffers.push(s);
10                 RecursiveAddBuffer(b, view, s);
11             end
12         end
13     end
14 Function RecursiveAddBuffer (Brick b, View v, SubBuffer s)
15     if b.HasSubBufferForView(v) then
16         return;
17     end
18     b.SubBufferForView(v) = s;
19     s.AccommodateBrickProjectionBounds(b);
20     for Brick n in b.forwardNeighbours do
21         if n.renderIndex >b.renderIndex then
22             RecursiveAddBuffer(n, v, s);
23         end
24     end
```

Sub Buffer Minimisation

An important facet of this work is reducing the amount of sub-buffers necessary for storing brick accumulation values. Reducing the amount of sub-buffers is required to satisfy both storage requirements and final-image compositing complexity. Algorithm 2 outlines the method used to attempt to minimise the amount of unique sub-buffers required. The goal is to exploit the fact that — especially for a planar-camera array — neighbouring bricks will be close enough to each other in the overall render list that they may share a buffer without breaking the front-to-back compositing order required for a correct final image, in which case they may share a sub-buffer, expanded to fit both bricks' projection regions.

The algorithm begins with the entry brick for each view. A new sub-buffer is created, with dimensions satisfying the projected region of the starting brick. The 'forward neighbours' of each brick (i.e bricks which are further from the view-point by Manhattan distance) are then recursively traversed, testing for an existing assigned sub-buffer for the neighbour. If there is no assigned sub-buffer *and* it is later in the *overall* render order, it is assigned the current sub-buffer, and the sub-buffer is 'expanded' (only in terms of min/max bounds) to accommodate the neighbouring brick. When the recursive set of forward neighbours is exhausted, each brick in the current view's render list is checked to make sure there is a sub-buffer, running the recursive sub-buffer assignment algorithm if a brick has no sub-buffer assigned. With all sub-buffer information generated *now* the space for these sub-buffers is allocated on the GPU.

Empty Space Skipping

At some point the volumetric data must be transferred to the main memory of the system — in this case it is assumed that the data streams in from an external data source. During this process the method can generate ESS information on the fly, by creating a bit-mask of voxel intensities (not opacities) for a region of the volume. This approach was chosen to allow a finer granularity of ESS over a min/max approach. In the case of the brute-force LFDVR algorithm this information is used to generate the ESS data-structure. In the case of the brick-based LFDVR this information is used to determine if brick-data should be transferred to the GPU and whether or not a render kernel should be launched for a brick.

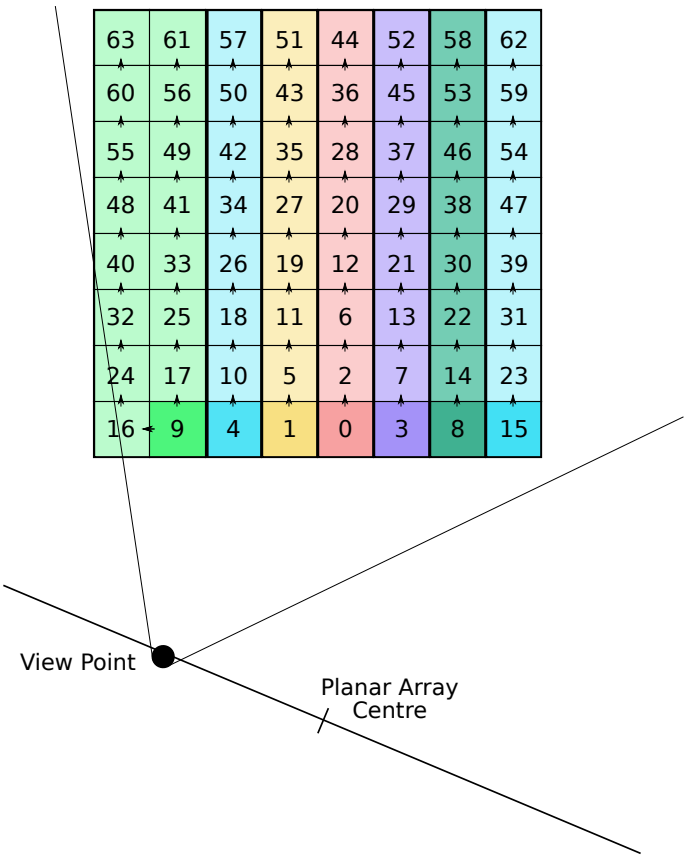


Figure 5.4: Graphic overview of the result of sub-buffer minimisation. Note that bricks with a different luminance are those which begin the recursive function outlined in algorithm 2. In this instance, 7 buffers are needed for the associated view point, rather than 64 unique sub-buffers for each brick.

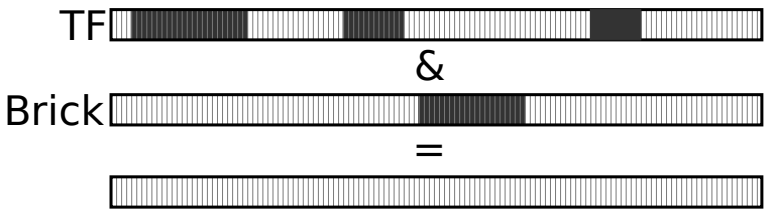


Figure 5.5: By comparing the voxel intensity bitmask of a brick to the intensity contribution bitmask of the working transfer-function's look-up-table the algorithm can decide whether or not a brick is considered empty space.

Compositing Kernel

In algorithm 2 a list sub-buffers was generated, which can be trivially ordered from front-to-back. Using this list the final-image's screen-space is divided into tiles, which are referred to as screen-tiles. Each of these screen-tiles are the work item of a CUDA thread block. In essence each SMM takes a section of the final image to work on.

Each thread block traverses the ordered list from front to back, first testing each sub-buffer in the ordered list for contribution to its target screen-tile. If there is no contribution to the current screen-tile the brick-tile is skipped and the CUDA block moves on to the next brick-tile in the list, otherwise the brick-tile is composited onto the final image. This method ensures that a correct final image is composited.

5.4 Implementation & Evaluation

The proposed approach was implemented as a module — or set of modules rather — for the open source data visualization tool Inviwo, demonstrating that this approach is viable as a real-world application. The main system used an Nvidia Quadro K2200 GPU and Intel Xeon E5-1620 v3 CPU. In most cases, the experiments involved a light-field camera plane rotating a full 360 degrees in 50 steps around a volume. Results presented were obtained from using a 512^3 floating-point volume of a supernova[98], and similar results were also obtained using other datasets whose size exceed the L2 cache size, excluded for sake of space.

As this method aims to exploit the shared L2 cache on the GPU, it should be expected that bricks that have a data size of sub-L2 size (2MB) to perform best. In the case of floating-point data, this is approximately 80^3 voxels or less. Taking into account other L2 access requirements for other rendering details in the kernel, brick sizes of 72^3 voxels or less are estimated to have better cache performance. Further to this, considering computational overhead and complexity of smaller brick-sizes, the expected performant range of bricks can be narrowed to between than 72^3 to 32^3 voxels. In the following experiments it is proved that this is the case.

Since this application area is relatively unresearched, there is little to compare against. Ray-guided approaches discussed in section 2 require too much off-line pre-processing for streaming volumes, and multi-view streaming dataset DVR approaches

are either quite simple or introduce artefacts via image-based-rendering techniques. A compromise can be reached by implementing both a standard brute-force DVR ray-caster and an empty-space-skipping octree DVR ray-caster as comparisons.

The implementation of this approach was written as a set of plugins for Inviwo [99] using C++ for the CPU code and CUDA for GPU code. The individual plugins where as follows:

- An ESS information generator, used to determine empty regions of the volume which was fed into both the octree-based ray-caster and the LFDVR ray-caster.
- A multi-view entry / exit points plugin which generated front-face entry and back-face exit ray positions of the volume's bounding box.
- An octree-based ray-caster used to compare against.
- An implementation of the main contribution of this approach, the LFDVR ray-caster.
- A light-field swizzler, which re-arranged sub-pixels of the light-field grid output into a format that could be displayed on the LookingGlass.

While the octree-based ray-caster exclusively used the ray entry / exit for rendering, the proposed LFDVR approach only used it for ray-direction, and performed a quick Axis-Aligned Bounding Box (AABB) intersection to find the entry / exit point of the brick it was currently rendering.

View-Count & Brick Size

In this experiment the render kernels of an implementation of an octree ESS ray-caster, and the proposed brick-based ray-caster are isolated, and these sections are timed on their own. The parameters changed in this experiment are the brick-size which are varied from 32^3 up to 512^3, and the light-field view-count which are varied from 1 view to 32^2 views. This view-count is a sub-division of the target screen resolution. Furthermore, displaying the volume with and without lighting is introduced, introducing more computation on the GPU along with more intensity on the cache from gradient sampling.

Figure 5.7 shows the results of this experiment with an overall target resolution of 4096^2 for view-counts of 4×4 upwards on the GTX 1080 respectively. It can be seen that when the brick-size begins to drop below the L2 cache size (2MB) on the GPU the proposed BBDVR algorithm out-performs standard octree ESS as much as $\times 2$ in the cases outlined as the expected performant range in the beginning of this section. This is due to the better temporal cache-coherence gained from only using a brick once for all views, verified in a later experiment 'L2 Cache Statistics'. While the graphs do not display data for view-counts 1×1 and 2×2 for the sake of figure space the results are discussed here. The proposed approach performs well in a many-view small-brick scenario. However, smaller view-counts and larger brick sizes reveal a computational overhead of the proposed approach in comparison to the octree ESS method as it no-longer gains data-reuse advantage.

Target Resolution

This work also examines the performance when changing the target buffer resolution parameter between 1024^2, 2048^2 and 4096^2 and show the average performance gain for all view-counts with varying brick sizes versus the ESS DVR. The results for these experiments are shown in figure 5.9. It can be seen that for a relatively small resolution size of 1024^2 the BBDVR approach slightly under-performs in relation to the ESS approach. It can be deduced that this is due to overhead in the kernel to manage additional parameters. It is observable, however, that with a brick-size of approximately 64^3 the BBDVR out-performs ESS regardless of target resolution. Interestingly, for the 1024^2 target-resolution, the performance plummets with a smaller than 32^3 brick size. This again is due to the kernel overhead of scheduling bricks to be used once and only once, waiting for bricks to be completed before moving to the next one.

Render List & Buffer Minimisation

In figure 5.8 the individual times of the Brick-Based DVR pipeline components is shown. Firstly, it can be seen that — for most cases — the components can be pipelined and overlap quite well. The exception to this is when the brick size drops to about 32^3, in which event the computational overhead on the CPU rises dramatically. This, coupled with the overhead on the GPU is reason to keep brick-sizes in the proposed approach

in the sweet-spot of approximately 64^3.

L2 Cache Statistics

As a final confirmation that this approach has achieved what was intended — better utilisation of the on-chip L2 cache — the application was profiled using the Nvidia tool 'nvprof', querying the metric 'l2_tex_hit_rate' on the Quadro K2200. While it was attempted to profile the application with a 4096^2 buffer size, nvprof reported metric overflow errors and couldn't record results. Instead the application was profiled with a 2048^2 output buffer divided into $32x32$ views, with a brick size of 64^3 floating-point data, testing the standard brute-force ray-caster, the octree ESS ray-caster and finally the brick-based ray-caster. As expected a substantial hit-rate improvement was observed from an ˜50% hit rate for the octree ray-caster, to an ˜87% hit-rate for the brick-based ray-caster.

Scalability

In addition to the primary system, the performance of the proposed approach with a more-capable GPU is also timed. The secondary system performed experiments on used an Nvidia GTX 1080. The difference between render kernel times on the K2200 and the GTX1080 can be seen between figures 5.6 and 5.7. In both figures an expected dip in render kernel time when the brick data size begins to drop below the L2 cache size can be observed. This shows that the brick-based approach scales with additional shader cores.

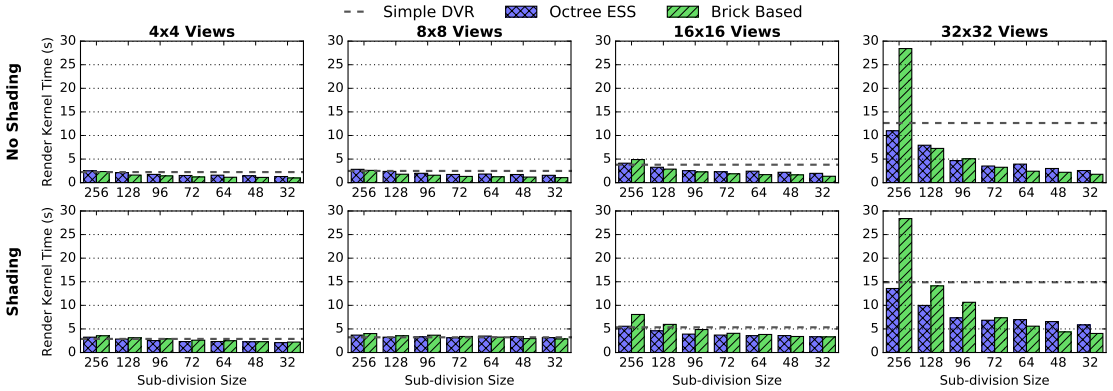


Figure 5.6: Render kernel times for multiple configurations of view count and brick size for an overall buffer size of 4092^2 rendering a 512^3 floating-point volume with and without shading on the Nvidia Quadro K2200. Note the dip in render kernel time with the Brick-Based DVR when the brick size begins to drop beneath the L2 cache size.

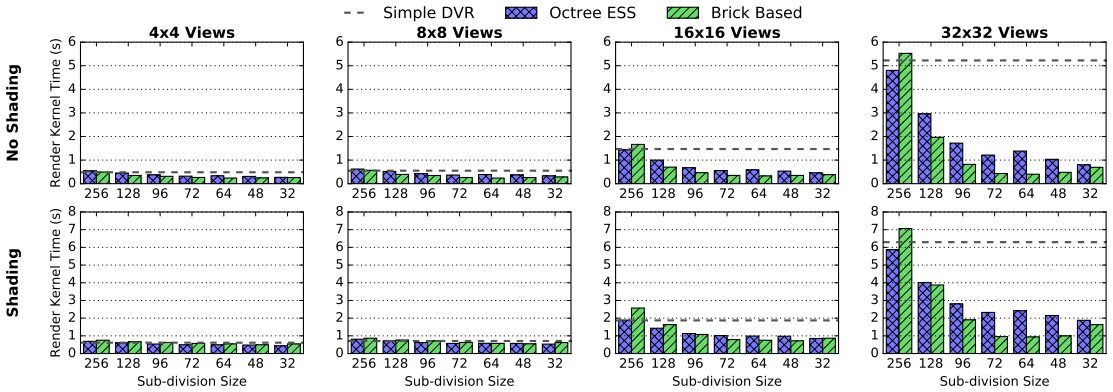


Figure 5.7: Render kernel times for multiple configurations of view count and brick size for an overall buffer size of 4092^2 rendering a 512^3 floating-point volume with and without shading on the Nvidia GTX 1080. A gain in performance is observed when brick size drops below the L2 cache size.

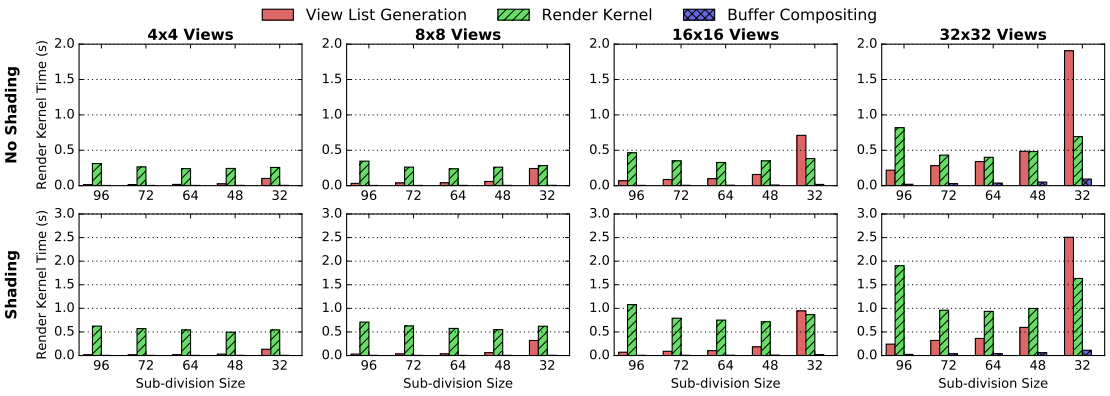


Figure 5.8: Individual times of the Brick-Based DVR pipeline on the GTX 1080 system. Note that when the brick size drops below 32, the view-list generation stage over-runs the render-kernel time, adding another computational-overhead reason to keep brick sizes at approximately 64^3

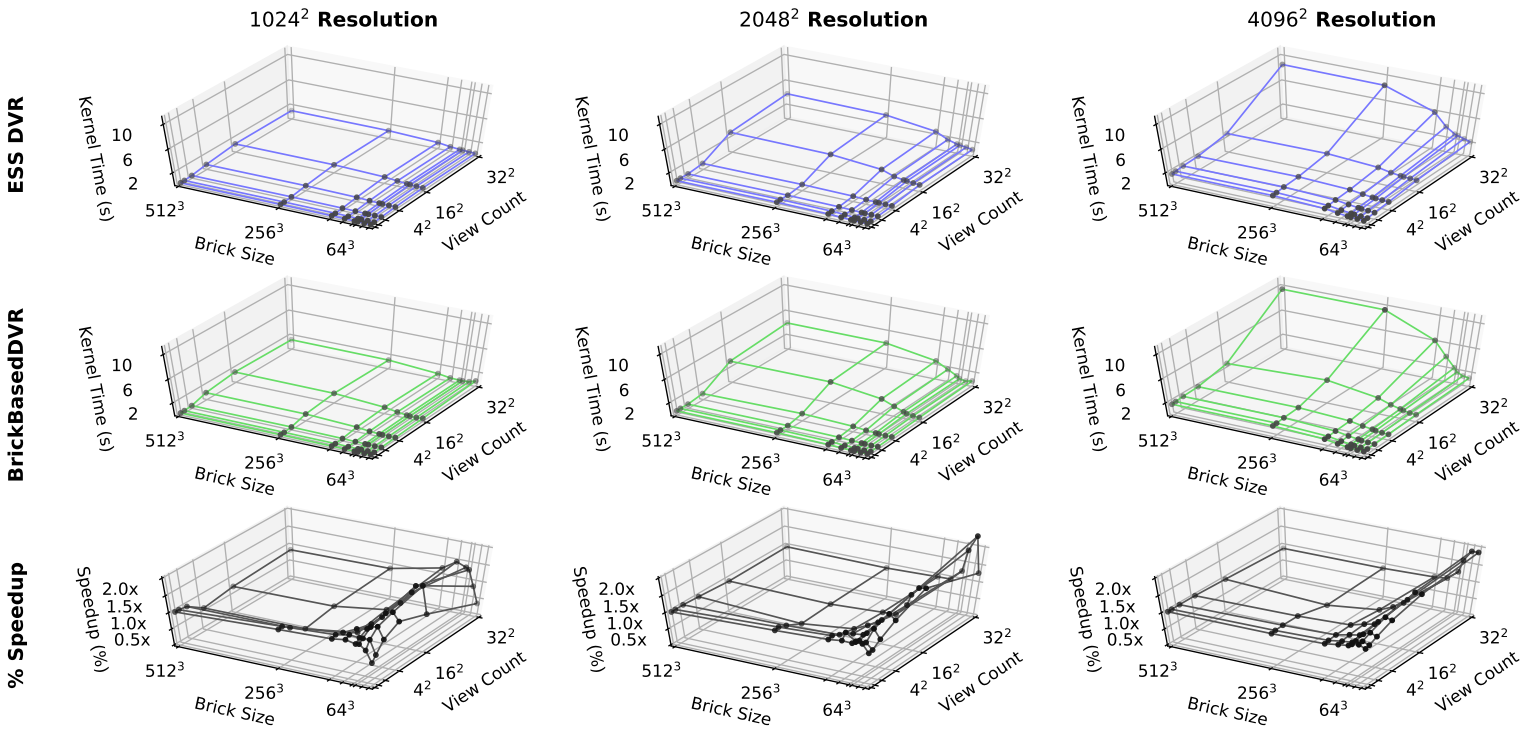


Figure 5.9: Comparison of render kernel times for both the ESS DVR method and the proposed Brick Based DVR method. The bottom row shows the averaged speed-up across all measured view-counts with different brick-sizes. Note the speed-up gained for brick sizes around 64^3 for all resolutions and view-counts.

5.5 Conclusion

This chapter has presented a novel approach in the relatively untapped area of light-field volume rendering of streaming time-varying datasets. It argues that ray-guided approaches require too much pre-processing to handle streaming datasets on-the-fly, and current light-field rendering methods for streaming volumes are either simplistic or image-based, introducing artefacts. Results have shown that by forcing an increased spatio-temporal sampling coherency, DVR can benefit substantially when rendering streaming time-varying volumetric data to a light-field. By limiting regions of the volume to a certain size and using them once and only once, DVR can exploit the temporal-coherency leverage given to us by the shared L2 cache on a GPU, and return a very significant performance increase of up to 2x in contrast to an octree ESS approach when multiple views must be presented. Furthermore, it is shown that this approach scales with additional shader cores. Finally, due to the pipelined nature of this approach with the CPU performing working-set determination and data scheduling, and the GPU performing the rendering and compositing, it is compatible with streaming time-varying volume data without need for extensive pre-processing or sub-sampling.

This chapter was presented as a short-paper in Pacific Graphics 2018 [14] and has been used as a practical platform for other papers — both published [18, 17] and pending publication — from the same research group.

Chapter 6

BVH Direct Volume Rendering on GPU

6.1 Goals

In almost all recent GPU DVR applications octrees in some way or form are the prevalent acceleration data-structures used. There are clear benefits that octrees provide to DVR; easy-to-implement ESS, clear and defined volume paging and caching, and trivially sub-sampled data support are to name but a few. Octrees have long been a natural acceleration data-structure for volumes on GPU due thanks to the clear, defined and easy to implement subdivision pattern, with predictable traversal times and well researched algorithms.

BVHs have also been investigated as a form of acceleration for DVR, however most of this research has been geared towards CPU rather than GPU. BVHs are designed to handle regions of varying size by using AABBs to spatially group surface data like polygons. While AABBs transition nicely to the regular grid structure of volume data, little research has been done on their performance for volume rendering on GPU. This is partly due to the impression that BVH build times impede interactive exploration via transfer function updates. Additionally, it may seem wasteful to create a BVH tree around groups of adjacent active regions that may be considered dense and thus may as well share a leaf node to reduce build and traversal complexity. Recent advancements in GPU technology have provided hardware-based BVH traversal and

ray-primitive intersections which has the potential to make the GPU a more viable candidate for BVH DVR.

In this chapter, an investigation into the characteristics of BVHs on GPU in terms of render-performance, render-complexity, and build times is presented. This is additionally compared against a recent state-of-the-art approach, Sparseleap [6]. A method to cluster acceleration leaves that has a significant impact on render times due to reduced tree- and depth-complexity is presented, exhibiting leaf-count reductions of up to 50% in the average case, improving render times by roughly 10-15%. This work proposes that using BVHs on GPU for DVR is now a viable approach and that tree-build times do not impede interactive exploration and are in fact one of the lowest costing stages of the pipeline. It is finally shown that render times using BVHs can be 20-40% faster than a state-of-the-art implementation with less deviation from the average during exploration.

6.2 Background & Related Work Recap

Volume rendering, and more specifically DVR, has been a well researched topic in the field of computer graphics. Fundamentally it is the accumulation of light through a potentially heterogenous medium represented as a regular grid of data. This data is re-sampled as the DVR algorithm steps along a ray, accumulating colour and opacity [19, 100].

CPU volume rendering has also been well researched, but in general has been focused more on large-scale volume data [101, 102] citing larger memory real-estate in RAM and removing the need to transfer data to GPU. Some CPU-focused work has used BVHs to accelerate data traversal [58, 103, 104]. These approaches all work well for large data and can scale well to clusters of systems, however in this work a single CPU - single GPU system is assumed. In this scenario the GPU currently massively surpasses the CPU in parallelism and is rapidly improving in terms of memory capacity and bandwidth.

GPU volume rendering has come a long way and a comprehensive state of the art report has been presented by Beyer et al[92]. The works that are most relevant are Fogal's Tuvok renderer[49, 50], Liu et al[51] and Hadwiger et al[5, 6]. All of these works share a common theme of some form of hierarchical data-structure. For example

[49, 50, 51] use octrees for ESS and data paging/sampling. Hadwiger et al. [5] instead used a multi-level hierarchical data structure to facilitate paging of peta-scale volumes, and expand upon that work[6] by using an octree-like occupancy histogram tree to generate occupancy geometry for rasterized ESS traversal. It can then be seen that octrees have been the data structure of choice for large scale volume rendering in most relevant works [4, 105, 51, 49, 50, 5], chosen for its logarithmic search times and inherent adaptability for LOD data. However, as recent works make note [48, 6], octrees can be more of a hindrance for dense regions of the volume where overhead is introduced traversing from brick to brick in sparse volumes with potential thin strands of opaque media.

6.2.1 Nvidia OptiX & RTX

In terms of actual BVH research, there has been recent work into hardware acceleration of construction [42], traversal and intersection test [41], and some of these concepts have finally been implemented in consumer hardware. With surface path-tracing being the topic of an immense amount of research, it was only a matter of time before some of these concepts were this happened. While the OptiX SDK by Nvidia [60] has been around for a while now, it has been utilising the massive parallel compute power that was already present in hardware and exposed in CUDA, albeit with insider-knowledge and clever scheduling. Only recently have they accelerated this by implementing some core path-tracing concepts like BVH traversal, AABB and ray-triangle intersection hardware with their new RTX line of GPUs.

While octrees may be a more trivially suitable candidate for DVR, it was time that a proper investigation and evaluation of hardware accelerated BVHs in the context of GPU DVR was warranted. This means analysing the traversal characteristics and performance of both octrees and BVH. For interactive DVR exploration especially, how either data-structure's build time and traversal change with respect to transfer-function updates are vitally important to the end-user scenario.

6.3 Evaluation of BVHs for DVR on GPU

There are many reasons why a BVH approach may be chosen over a more regular acceleration data-structure such as an octree. For example, even if both an octree and a BVH share the same leaf-size in a sub-divided volume, sparsely populated data may require fewer inner tree nodes to be traversed to achieve the same amount of skipped empty space. Secondly, using a BVH allows for the seamless integration into existing path-tracing tools, such as Nvidia's OptiX which can then be used in production renderers for offline path-tracing when volumetric media needs to be used — clouds or smoke for example. Following that, by using OptiX, much of the hard work can be accomplished by the existing library, and offloaded to hardware accelerated implementations.

However, advantages aside, as discussed previously there has been little investigation into performance characteristics of BVH DVR on GPUs. To emphasise again, especially with new hardware acceleration, an evaluation of BVHs needs to be performed on GPUs. In particular, this work is interested in two major parts of the interactive DVR pipeline: transfer function editing and spatial exploration. In the first example, a user may tweak a transfer function making certain data less or more visible. When data that was previously completely transparent becomes visible, this means that the acceleration data-structure needs to be updated. In the case of the BVH, this potentially means that a new leaf-node needs to be added to the scene and the hierarchy needs to be updated. It is this build time that needs to be evaluated if hardware accelerated BVH DVR is to be a viable candidate.

For the second task — spatial exploration — it needs to be determined how BVH traversal handles many potentially transparent regions of volume data that may be redundantly touching (more about that in section 6.4) creating additional depth complexity. In a current state-of-the-art approach [6] rasterization of occupancy geometry — which can loosely be thought of as an octree although the actual geometry skips levels depending on the transfer function — generates a list of ray-segments which can be traversed in order to efficiently skip empty space. Because these ray-segments can be compressed on the fly, continuous regions of active leaves can compressed into a single ray segment. In comparison to BVH ray-traversal, exiting one leaf and entering an adjacent leaf can require a restart of the BVH search. This work evaluates the impact

this has on DVR with both opaque and mostly transparent transfer functions which generate the same amount of leaves, only differing in the amount of early terminated rays. In the next section, it is shown how this effect can be greatly mitigated.

6.4 Region Clustering for BVH

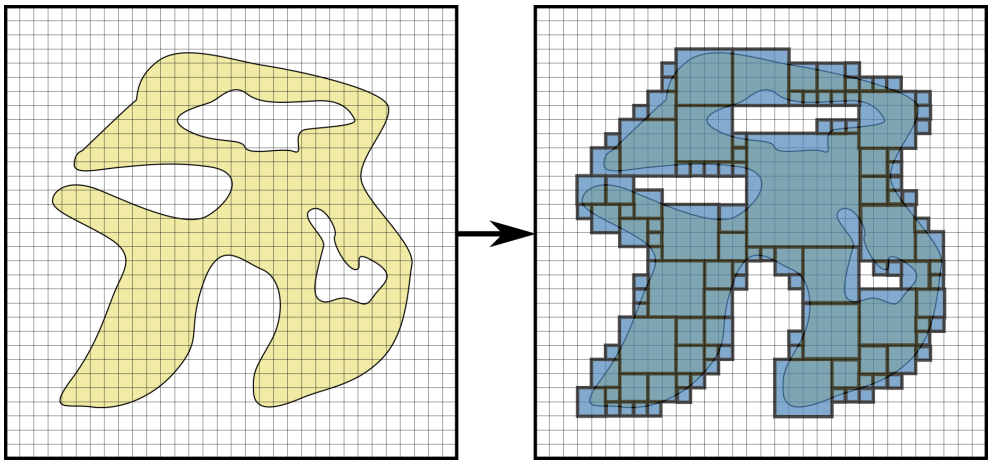


Figure 6.1: In this approach active bricks in a volume are clustered to reduce spatial complexity in the data structure. Note that these clusters are not tied to a regular spatial subdivision like octrees and this can cluster arbitrary groups of active bricks.

A core contribution of this work is reduction of data structure complexity by spatially clustering active subdivision leaves in the volume. There are two major advantages that justify this step: Firstly, BVH construction complexity can be reduced substantially facilitating faster refit or rebuild times when the transfer function changes, although in section 6.6 results show that BVH build times are quite fast even without this step.

Secondly — and perhaps more importantly — BVH traversal complexity can be reduced substantially. By clustering active leaves in the subdivision the amount of BVH leaves can be massively reduced, lessening the complexity of the hierarchy and thus improving ray traversal times. This is demonstrated in section 6.6.2. These benefits do however come at a cost, which is the actual clustering phase which are performed on the CPU. There are two major challenges to face in this task.

The first challenge lies in the complexity and performance of finding ideal or close to perfect clustering that minimises the leaf node count. A simple naïve single-pass approach could begin by traversing the bricks in a scanline fashion, attempting to group as many bricks as possible. While simple and easy to implement, this has the major drawback of increased fragmentation as the scanline proceeds leading to larger clusters at the beginning of the scanline and many smaller clusters towards the end. In the proposed solution a greedy-like algorithm is used to cluster bricks in descending size using a copy of a vector of booleans representing the active leaves in the subdivision, coupled with a 3D version of a summed area table[106] (3DSAT).

The process begins by creating a vector of booleans — or a long bit string — that represents the active leaves in the subdivision. Then a 3DSAT the same dimensions as the subdivision grid is filled. If a leaf is active the sum is increased by 1 and the general process for populating an SAT is used to generate the 3DSAT. When clustering the leaves the 3DSAT is queried to determine if there are the same amount of active leaves as the amount of leaves that are needed to cluster — for example if the algorithm is attempting to cluster an 8^3 group of leaves, the 3DSAT is queried for the sum of all active leaves in an 8^3 window. If the result is exactly 8^3 then is is known that all the leaves in the region are active. This is coupled with checking the 8 corners of the 8^3 window against the active leaf mask. If all 8 corners are active, this region has not been clustered before. When a region is clustered, the active leaf mask is updated so that all leaves grouped by the cluster are marked as 'inactive', and are therefore not considered in following checks. This sliding window can be thought of as a form of convolution kernel, albeit in serial.

An outline of this process is shown in algorithm 3. It's important to note about this algorithm that a copy of the active list of bricks is used. When a cluster is found, the leaves that are covered by that clustered are marked as inactive in the copied list, effectively acting as a mask preventing any of those leaves being added to a different cluster. A visual example of what this would look like is shown in figure 6.2.

The second challenge faced lies with the brick pool: What is the easiest way for the brick pool to handle clustered regions of ESS information? The short answer is to decouple the ESS from the actual DVR sampling. In this solution inspiration is taken from Hadwiger et al's 2012 [5] and 2018 (Sparseleap) [6] works, that separates the ESS information from the actual volume sampling layer. This means that the BVH can use

Algorithm 3: Greedy algorithm to cluster regions of ESS leaf nodes. Note that *clusterSizeList* is in descending order and that *maxClusterSize* is a volume size limited, user defined variable.

```
1  activeList = getActiveLeaves();
2  sat = generateSAT(activeList);
3  clusterSizeList = {maxClusterSize .. 1};
4  clusterList = {};
5  for c in clusterSizeList do
6  |   area = c * c * c;
7  |   for z in numLeaves.z - c do
8  |   |   for y in numLeaves.y - c do
9  |   |   |   for x in numLeaves.x - c do
10 |   |   |   |   vec3 min = {x, y, z};
11 |   |   |   |   vec3 max = min + {c, c, c};
12 |   |   |   |   if !cornersActive(min, max) then
13 |   |   |   |   |   continue;
14 |   |   |   |   end
15 |   |   |   |   if sat.sumBetween(min, max) == area then
16 |   |   |   |   |   clusterList.push(min, max);
17 |   |   |   |   |   activeList.mask(min, max);
18 |   |   |   |   end
19 |   |   |   end
20 |   |   end
21 |   end
22 end
```

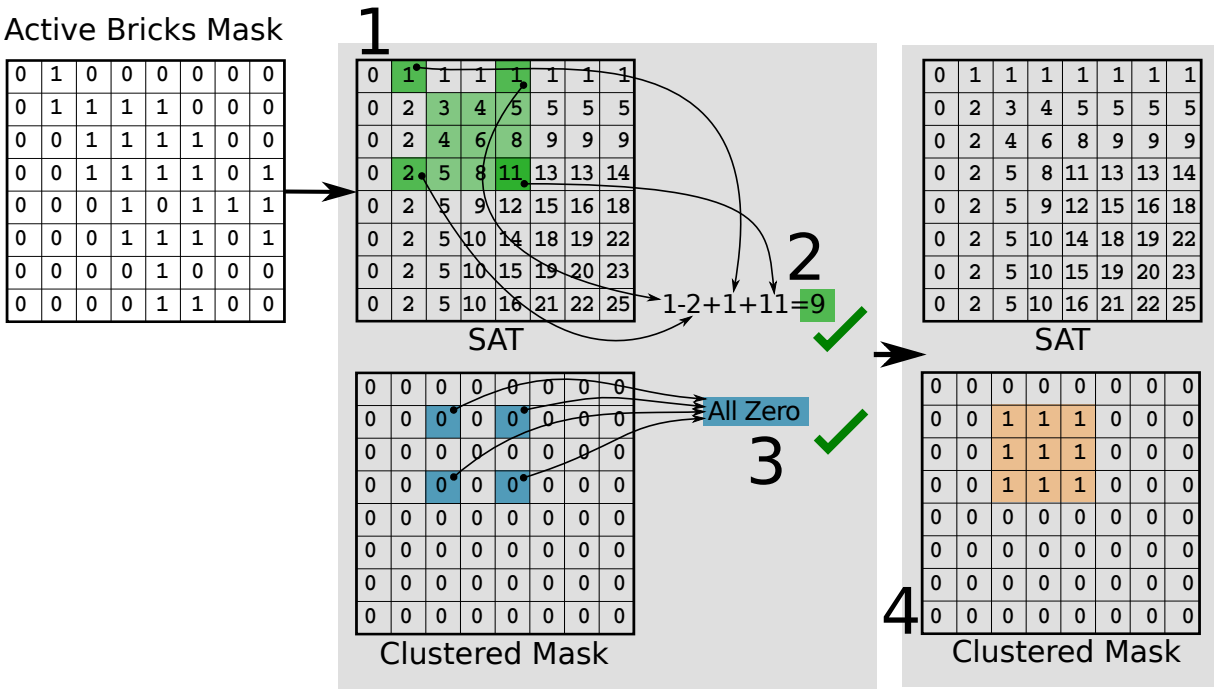


Figure 6.2: Visual outline of the 3DSAT clustering method, simplified to a 2D example.

a fine-grained leaf size and the volume pool can use a brick size that suits I/O needs or page table complexity, a trait that can be important in massive-scale DVR. While Sparseleap use a multi-level page structure, for the intents and purposes of this work, a minimised version is used. The implementation of this brick pool and page table are discussed in more detail in section 6.5.3.

6.5 Implementation

This approach was implemented using Nvidia's OptiX 6.0 SDK which allows for use of the hardware accelerated BVH tools. Their new line of graphics cards that provide hardware acceleration for BVH traversal were targeted as the platform for the proposed solution.

6.5.1 Occupancy Information & OptiX

The first stage of the pipeline is occupancy information. The volume is subdivided into bricks of identical sizes, storing some information about the brick contents; i.e. min/max value or a bitmask of value ranges. In this implementation just min/max values for a brick are used, rather than full bitmasks as in chapter 5. This information will be tested against the transfer function when there is an interaction. This content information is stored in an array in RAM, and can be saved to disk for future re-use of the volume.

When the transfer function is updated the content information array is tested in parallel — using OpenMP in this implementation — and the active/inactive flags are stored in a bitmask array or vector of booleans. At this point, if clustering is not enabled, the active/inactive flags can then be used to generate bounding box information — in this implementation world-space min/max coordinates of each region are used as an AABB. This information is stored in a buffer which effectively represents separate primitives in an OptiX geometry instance.

6.5.2 Clustering

If clustering *is* enabled the method outlined in section 6.4 and listed in algorithm 3 is used to generate a STL vector of min-leaf-index to max-leaf-index bounds which is then used to generate AABB min/max bounds. In the implementation this clustering method is single-threaded and a potentially naïve method of getting the job done, however this should be seen as a proof-of-concept method to accelerate stages later in the BVH DVR pipeline like build times and ray-traversal in both opaque and transparent volumes.

Regardless of if clustering is enabled or not, the AABB information is stored in an OptiX buffer and used as geometry primitives in a *single* geometry instance that has a *single* material assigned to it. During development this was found to be much more performant than a geometry instance per AABB.

6.5.3 Brick Pool, Page Table & Sampling

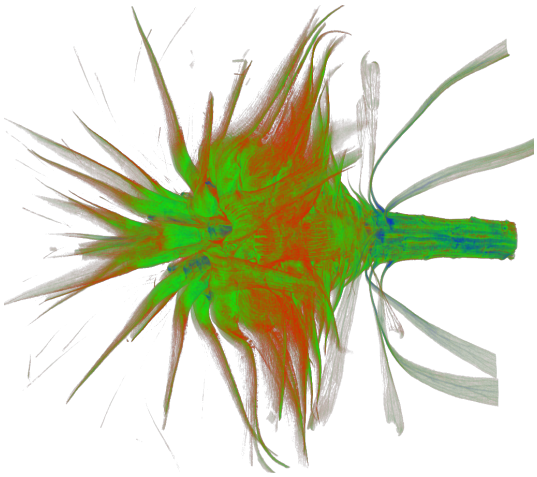
The underlying sampling technique used in both the OptiXDVR approach and the implementation of SparseLeap [6] is a simplified version of Hadwiger et al's earlier work [5]. During sampling perform a look-up into a page table is performed. This look-up determines if the region that the sample resides is active. This look-up also gives an offset into a large brick-pool where the actual volume data resides. As noted in SparseLeap [6], this allows the disconnection of ESS and sampling/paging. A visual example of this was shown in chapter 2 in figure 2.8.

Since the efficient sampling of volumes is not the primary focus of this chapter's work relative to empty-space-skipping, the implementation of this method is simplified by assuming two things. Firstly, it's assumed that the volumes are small enough to not use a multi-level paging technique, and as such just use one page table for the whole volume. This still allows considerably large volumes to be visualised. Secondly, it's assumed that the volume content information is known at run-time and there are no 'unknown' regions of the volume. Regardless of whether this is done offline or in a pre-processing step, streaming incomplete data is not evaluated as part of this work, although the underlying sampling can be updated to facilitate this.

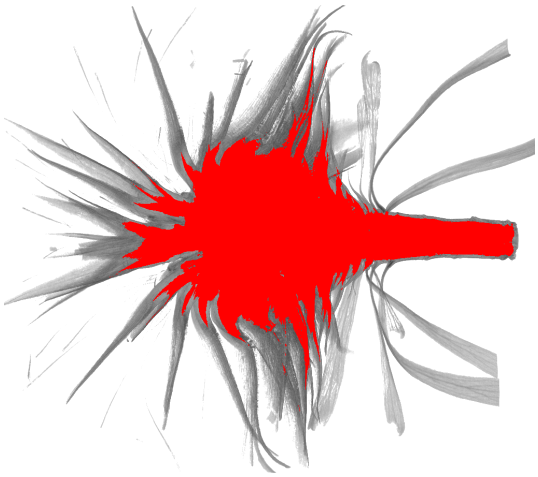
It is important to again stress at this time that the sampling part of this DVR implementation can be made more complex to accommodate these requirements if needed with few changes need to be made to the BVH ESS part of the pipeline.

6.6 Results & Evaluation

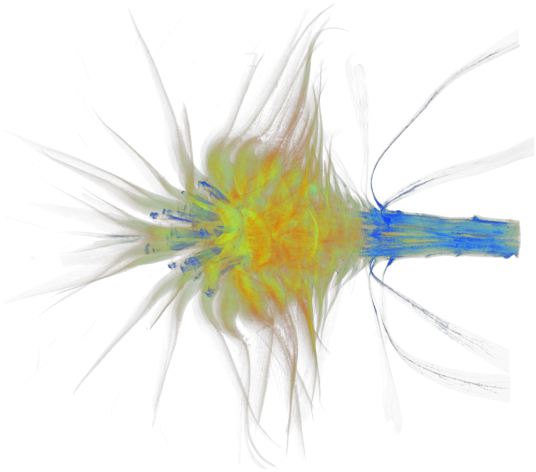
In this section an evaluation of the performance characteristics on the BVH clustering approach is performed when using different parameters, i.e. volume, transfer function, clustering, leaf size, brick size, etc. The approach is additionally compared to an implementation of SparseLeap [6], which is implemented this to the best of the authors ability using the pseudo-code provided in the original work. Much of the same characteristics as shown in their evaluations are observed and verified. The same OpenGL ARB extension *ARB_fragment_shader_interlock* is used to process occupancy geometry in order per-fragment. For linked-list generation inspiration was taken — like Sparseleap — from Yang et al. [107]



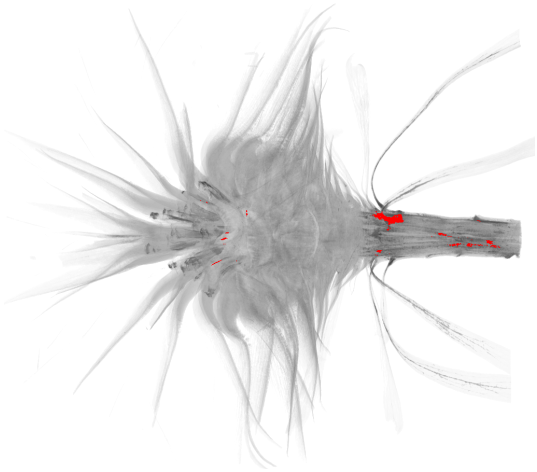
TF1 Colour



TF1 ERT



TF2 Colour



TF2 ERT

Figure 6.3: Examples of the flower dataset used with the two different transfer functions used. Note that the ERT rays are highlighted in red and are substantially less prevalent in the second transfer function. Also note that both transfer functions set the exact same amount of active/inactive regions, thus the only difference is amount of ERT.

While SparseLeap is implemented in OpenGL and the BVH DVR approach is using OptiX, most of the underlying code for occupancy information, paging, etc, are common to both, minimising differences as much as possible and making fair comparisons. The underlying sampling method are briefly discussed in section 6.5.3. Both the OpenGL SparseLeap implementation and the BVH OptiX implementation use the *exact* same underlying implementation and data for the page table and brick-pool, the only differences being that SparseLeap uses OpenGL textures and the BVH DVR uses *rtBuffer* objects and *rtTextureSampler*.

In the following experiments the appropriate sections are timed using the system clock, starting and stopping a timer before and after the evaluated stats. For OpenGL, *glFinish()* is called before starting the timer and before stopping the timer.

The system used for the following experiments was a Dell Precision Tower 5810 with an 8-core Intel Xeon E5-1620 v2 CPU and an Nvidia GeForce RTX2080 using Ubuntu 18.04.2 LTS. The OptiX SDK was version 6.0, OpenGL version was 4.5, and the Nvidia driver used was 418.43.

6.6.1 Datasets

To best maintain a level of fair comparisons a multitude of datasets are used during experimentation. The primary dataset shown in this chapter is a μCT scan of a flower with a resolution 1024^3 8-bit integers obtained from UZH [108]. Examples of this dataset are shown in figure 6.3. The beechnut dataset also obtained from UZH is 16-bit unsigned integer 1024^2x1546, and although not shown in any of the images or figures, exhibited similar results to the Flower dataset. Both of these datasets obtain a decent level of clustering while maintaining regions of thin strands that are difficult to cluster.

In contrast to this, a frame from a Supernova simulation previously obtained from UC Davis [98] is used. This data is upscaled from 432^3 to 2048^3 8-bit unsigned integer data. The Supernova with the tested transfer function exhibited large amounts of clusterable regions. An example of this dataset is shown in figure 6.5. The website for this dataset is no-longer available when last checked, but copies of the data can be supplied on request.

Both the Flower and the Supernova datasets are chosen to represent varying

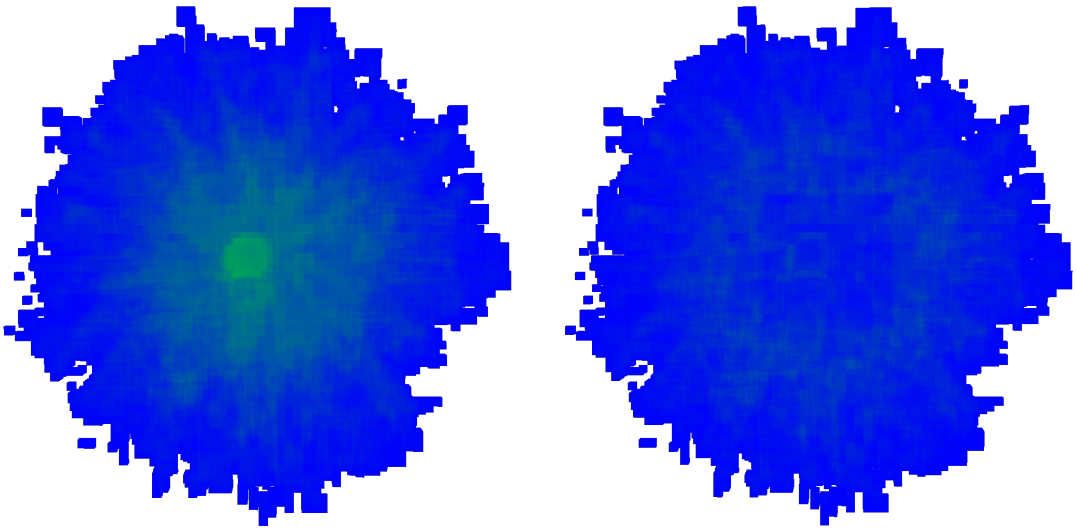


Figure 6.4: Depth complexity comparison of a head-on view of the flower dataset with a leaf size of 16^3 voxels. See section 6.6.2 for explanation of heatmap colour coding, and table 6.1 for statistics on the amount of clusters present.

degrees of clusterability, the percentages of which can be seen in the '% of B_{Active}' column in table 6.1. However, in addition to different datasets, it is important to emulate the interactive nature of DVR applications by using different transfer functions on the same volume. In figure 6.3 two different transfer functions are shown on the same volume. This is necessary to evaluate the impact that large amounts of subdivision leaf nodes has on depth complexity for both the SparseLeap implementation and the BVH OptiX DVR, as such TF1 has a high ratio of ERT relative to rays which actually sample the volume (rays which enter at least one active leaf). TF2 is quite the opposite with very little ERT, allowing rays to traverse through the volume almost entirely. It is important to note that both TF1 and TF2 exhibit the *exact* same amount of active leaves in the subdivision and only differ in opacity accumulation.

6.6.2 Clustering

In table 6.1 comprehensive statistics are shown about both the leaf subdivision and clustering statistics of the clustered BVH method. It can be observed that for the flower dataset using both transfer functions a reduction of between approximately 40% to 50% is achieved. For the Supernova dataset — which has much more contiguous

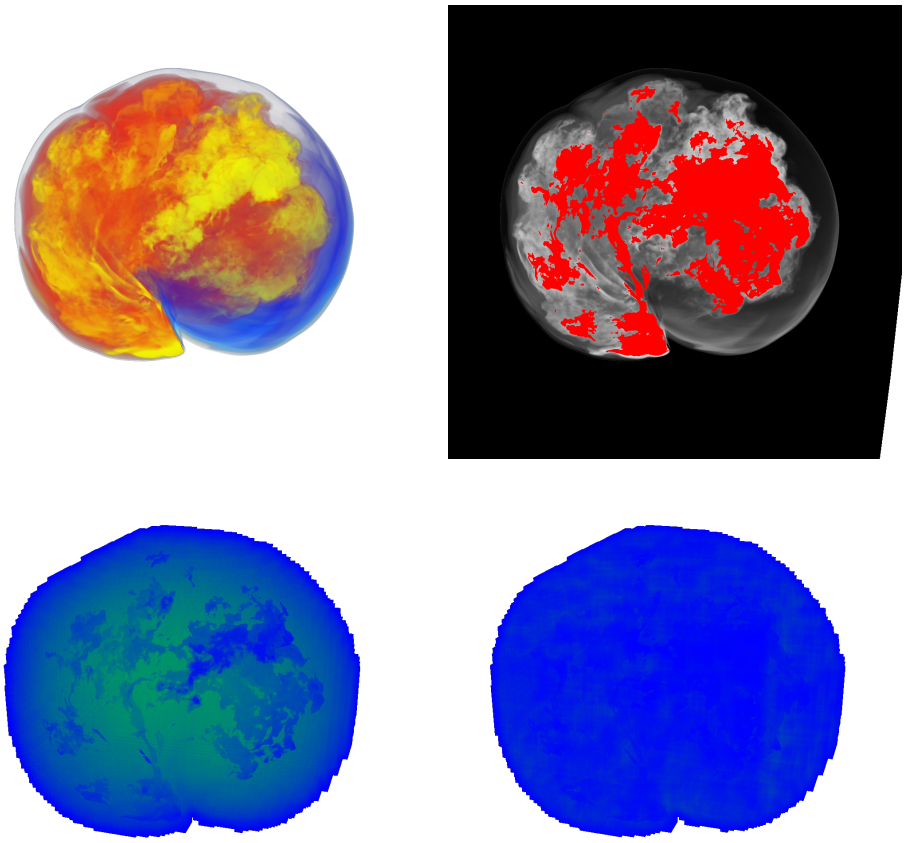


Figure 6.5: Depth complexity comparison of a head-on view of the flower dataset with a leaf size of 32^3 voxels. Top row shows colour image of rendering (left) and the early-terminated rays (right). The bottom row shows the depth complexity with clustering off (left) and on (right). See section 6.6.2 for explanation of heatmap colour coding, and table 6.1 for statistics on the amount of clusters present.

space — a more substantially reduced leaf complexity of 5% to 35% is obtained.

	B_{size}	B_{Total}	B_{Active}	(% of B_{Total})	T_{tf}	$B_{Clusters}$	(% of B_{Active})	$T_{Cluster}$
Flower	128	512	261	50.98%	0.01ms	139	53.26%	0.05ms
	64	4,096	1,126	27.49%	0.08ms	566	50.27%	0.37ms
	32	32,768	4,883	14.90%	0.73ms	2,606	53.37%	1.89ms
	16	262,144	22,058	8.41%	1.58ms	10,605	48.08%	17.63ms
	8	2,097,152	112,139	5.35%	7.48ms	43,603	38.88%	112.46ms
Supernova	128	4,096	811	19.80%	0.03ms	268	33.05%	0.08ms
	64	32,768	5,324	16.25%	0.14ms	1043	19.59%	1.67ms
	32	262,144	4,883	14.63%	0.60ms	5232	13.64%	13.00ms
	16	2,097,152	290,864	13.87%	5.98ms	23947	8.23%	107.33ms
	8	16,777,216	2,262,811	13.49%	27.68ms	112801	4.98%	1090.02ms

Table 6.1: Statistics on subdivision leaf clustering for different leaf-sizes ('B_{size}') showing the total number of subdivision leaves ('B_{Total}'), the number of active leaves ('B_{Active}') for the given transfer function and that number as a percentage of the total number of leaves ('% of B_Total'), the amount of time taken to test all leaves against the transfer function ('T_{tf}'), the amount of clustered leaves ('$B_{Clusters}$'), also represented as a percentage of the amount of active leaves ('% of B_{Active}') and finally the amount of time taken to cluster the leaves ('$T_{Cluster}$').

Visual comparisons of depth complexity both with and without clustering enabled are shown in figures 6.4 and 6.5. The colours range from blue to red representing a depth complexity of 1 to $MaxDC$ which is defined as the Manhattan distance from one corner to the opposite corner of the volumes going by bricks, i.e. $numLeaves.x + numLeaves.y + numLeaves.z$. Note that none of the displayed images ever reach a depth complexity of 100% since there is always a portion of the volume that is skipped and the experiments are mostly run on the horizontal x-plane.

Using the flower dataset results in figure 6.4 show that there is an observable reduction in depth complexity straight through the middle of the volume. There is however a substantial amount of un-clusterable regions — or regions that were already relatively low in depth-complexity — around the fringes of the volume. Looking at figure 6.3 which shows a side-view of the volume, it can be seen that the strands of active regions are thin enough to make it difficult to cluster.

On the other hand, using something like the Supernova dataset shown in figure 6.5 which exhibits large amounts of adjacent subdivision leaves, a considerable reduction of depth complexity throughout the volume is evident. These claims are backed up by statistics in table 6.1 and the render performance benefits can be observed in figure 6.6.

6.6.3 BVH Build Times

BVH build times have been an major factor in the lack of adoption for DVR. To evaluate the actual implications, the transfer function is varied to increase the amount of active leaves in the subdivision. In figure 6.7 build times are shown with a varying active leaf rate of approximately 8% to 100% for a leaf-size of 16^3 using the flower dataset. Note that this time *includes* the time it takes for creating AABB information and uploading to the GPU. Results show that there is a relatively linear increase in time, but remains well below 20ms for this dataset.

This only tells part of the story however. In table 6.1 data relating to clustering time for different configurations is presented. For the most part, the time taken to cluster a volume dwarfs the time taken for the BVH build. It is important to note however, that this clustering implementation should be considered naïve since it is a proof-of-concept for leaf complexity in BVH building and ray traversal, and can

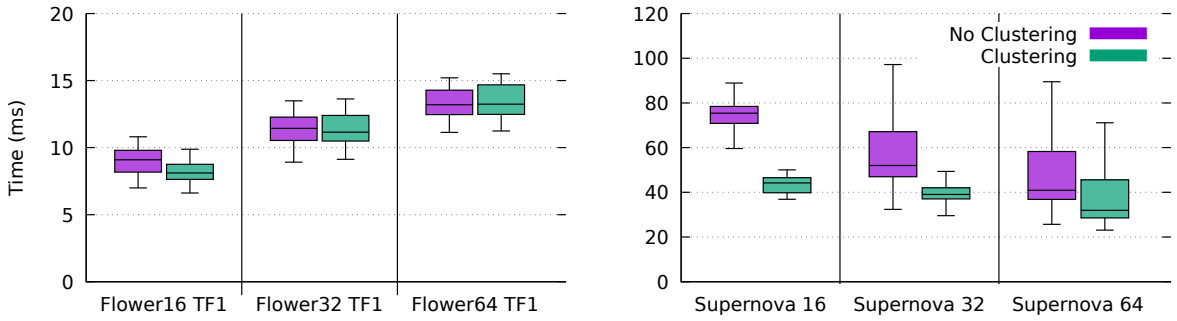


Figure 6.6: Difference in render times with clustering off/on for differing subdivision leaf sizes of 16^3 ('B16'), 32^3 ('B32') and 64^3 ('B64') for the Flower dataset using transfer function 1 ('TF1') shown in figure 6.3 and the Supernova dataset shown in 6.5. Massive stability in render times can be seen by using clustering for the Supernova. In the Flower case, improvements are seen even for mostly opaque transfer functions. This is important so as BVH traversal complexity is reduced in the event many leaf nodes are not needed for rendering, rather than having a deep hierarchy where most of the leaves aren't even touched.

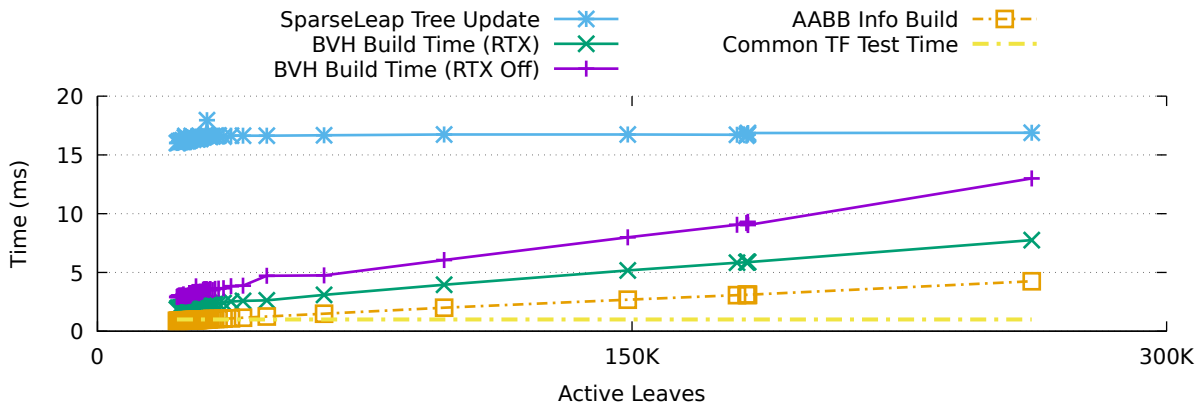


Figure 6.7: Comparison of times for different renderers and configurations with varying amount of active subdivision leaves. Both the SparseLeap and OptiX DVR implementations share the same transfer-function leaf-test code ('Common TF Test Time'). Because the SparseLeap tree size doesn't change, the update time remains almost constant ('SparseLeap Tree Update'). This shows results for the BVH build time with and without RTX enabled ('BVH Build Time (RTX [Off])') which includes the time taken to generate and upload the AABB bounds ('AABB Info Build') and thus should be offset by this value to find the actual build time.

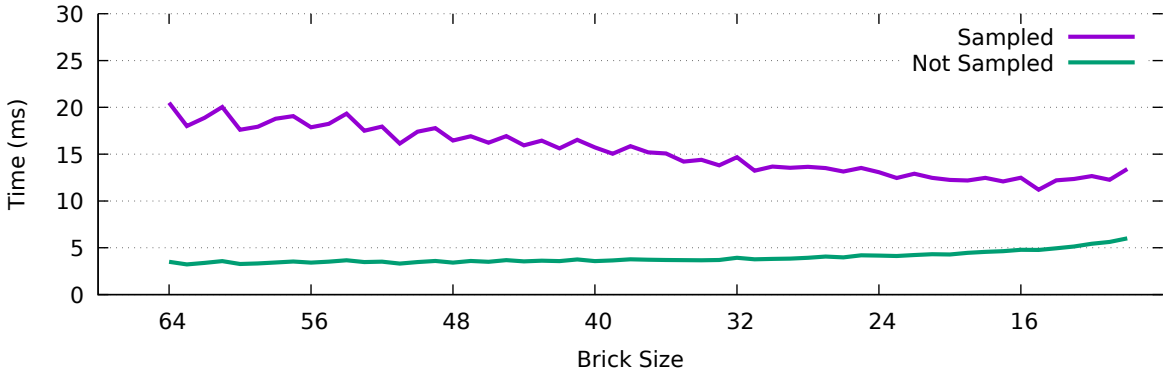


Figure 6.8: Average render times Flower dataset using TF2 (no ERT) with sampling on and off to evaluate the portion of rendering responsible for BVH traversal and scheduling in OptiX.

potentially be improved upon substantially.

Also compared are the BVH build times versus the occupancy geometry generation step of SparseLeap [6]. It is observed that both the occupancy tree update time (which included geometry emission) and the occupancy geometry render order time were relatively consistent at 16ms (shown in figure 6.7) and 2ms respectively.

6.6.4 BVH Traversal Costs

An important part of evaluation is BVH traversal performance for DVR using OptiX. One can roughly estimate the portion that belongs to OptiX based on the difference of rendering a volume *with* and *without* sampling the data while varying the brick size. This was achieved by stubbing the sampling code in the first-hit OptiX program by use of a run-time flag. In figure 6.8 the results of this experiment are shown. As expected, the rising cost of BVH traversal ('Not Sampled') can be observed as the brick size decreases and the subdivision count increases. This clearly has an observable impact on the end render time ('Sampled'). Interestingly, and again as expected, removing the cost of BVH traversal from the overall render time — the difference between 'Sampled' and 'Not Sampled' — reveals the actual cost of sampling the volume, which has a mostly reducing trajectory. This justifies using smaller bricks for tighter ESS granularity, but indicates the requirement for a reduction in data-structure traversal complexity. It is this reason that a clustering approach is proposed.

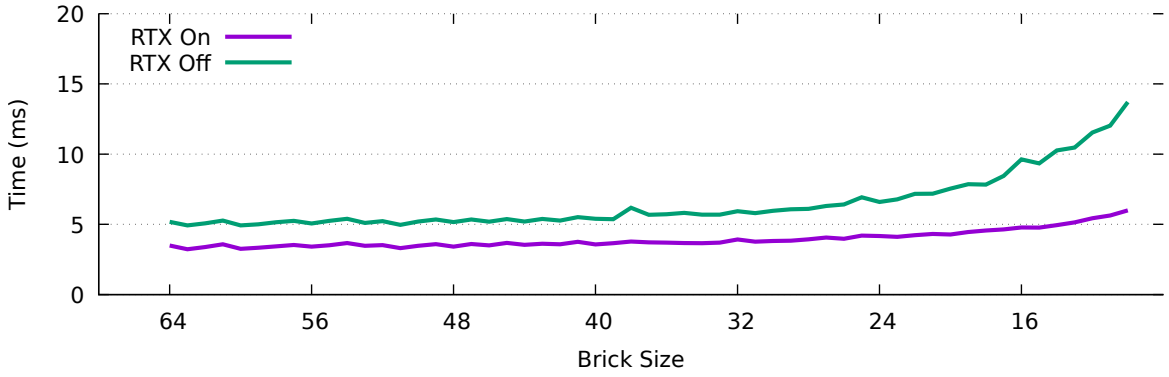


Figure 6.9: Difference in BVH traversal times with RTX off/on for differing subdivision leaf sizes for the Flower dataset. Inner volume sampling loop is stubbed to get an estimation of *just* the BVH performance. Clustering is not enabled for this experiment to force a larger amount of leaf nodes. See table 6.1 for indicative leaf counts, This shows clear benefit for using the RT core hardware when spatially subdividing a volume.

6.6.5 With & Without RTX

A core evaluation performed is the actual benefit of the new RT cores hardware. Starting with the first step in the pipeline, BVH build times are evaluated. In figure 6.7 the build times are shown — including AABB information creation and upload — with and without RTX. Interestingly, although the author is not aware of any advertised hardware for BVH construction, there is a substantial performance increase observed — dropping from approximately 10ms to less than 5ms when removing AABB information build time from the respective BVH build times. This is potentially due to whatever underlying data organisation used by the RTCore hardware. Unfortunately, no public information about this was found.

The major part of the pipeline that RTX and RTCores are expected to show massive performance benefit is during rendering. In figure 6.9 the difference in *just traversal* times is shown to highlight any improvement between configurations. It can be seen substantial benefits to using the RT core hardware acceleration, especially as the leaf size reduces, improving ESS granularity but maintaining a steady level of performance.

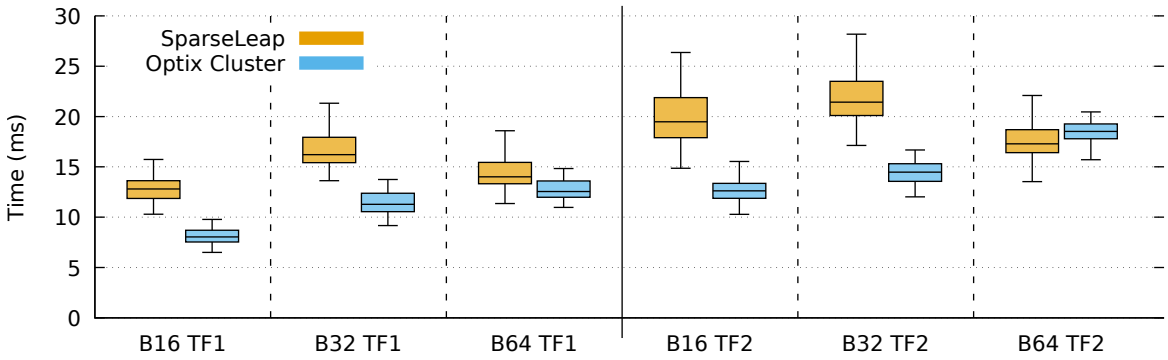


Figure 6.10: Render times comparison of SparseLeap [6] to this chapter's approach using the Flower dataset with different leaf sizes ('Bxx') and the two different transfer functions ('TF1' and 'TF2'). The underlying *sampling* layer used a brick size of 32^3. Note that the SparseLeap times do not include the geometry rasterisation step required for when the camera moves.

6.6.6 Render Times

Finally, and potentially most importantly, the actual performance of DVR for both this clustered BVH approach and the implementation of SparseLeap is evaluated. Figure 6.10 shows render-time results using both TF1 and TF2. The intent being to highlight the differences between performance when ERT is a prevalent feature during ray traversal. In all of these tests, the underlying sampling method used a brick pool with a brick size of 32^3, which was found to be the most performant in this case for both approaches. The results were obtained by performing 50 full rotations in 360 steps with the volume filling as much of the view-port without being cut off, timing the appropriate stages using the method outlined at the beginning of this section. Examples of frame 90 — a side-on view of the volume — can be seen in figure 6.3.

In almost all cases a substantial performance benefit was observed using the clustered BVH approach in terms of average render times. The exception to this rule is when there is little ERT and the leaf size is relatively large. In the case of SparseLeap one can see that a small leaf-size can sometimes be a hinderance during rendering. A smaller leaf-size in general means a finer level of granularity, but for a relatively fragmented volume such as the flower, segment counts can become quite substantial creating extra work for the fragment shader, showing there is a balance to strike.

In comparison, it was observed that the average render performance for clus-

tered BVH DVR is significantly better in most cases than the best-case SparseLeap configuration. For the flower volume using TF1 results showed an almost 40% improvement in render times, a statistic echoed using TF2. Another important quality that is often overlooked is render performance stability. In figure 6.10 it can be seen that while there is deviation of minimum and maximum render times from the average, it is quite stable relative to SparseLeap. This is considered an important quality in DVR when exploring a volume.

6.7 Conclusions & Future Work

This work has shown that BVHs are an extremely viable candidate for DVR on modern GPUs that implement new hardware for ray-tracing, giving us tight-wrapping empty-space-skipping structures with little effort. It has presented a method of leaf-clustering to help ray-traversal performance during rendering and prove the benefits in terms of both depth-complexity and traversal times. It should be noted that — while clustering improves the aforementioned stages — the actual clustering times can become a bottleneck for transfer function update times. This could potentially be alleviated by performing a GPU-accelerated clustering.

Additionally, the 3DSAT added considerable CPU memory pressure when fine grained subdivisions were used. It is necessary to keep in mind that this is a relatively naïve clustering implementation used as a proof-of-concept to demonstrate a method of reducing leaf complexity for BVHs. Their performance may be improved with more efficient algorithms, potentially using fast convolution kernels and using the massive parallelism on the GPU. It is important to re-iterate that one of the main reasons BVHs were previously avoided for interactive DVR was build-times. Results have shown that these build times — including the necessary steps to facilitate the build — are highly interactive and should not be considered a limiting factor.

While all of this work focuses on the ESS stage of DVR, it was observed that the main bottleneck in volume rendering is I/O, both in terms of loading data from disk/network and then uploading to the GPU. While this work does not consider this field relevant to its contributions, it is nonetheless a vital consideration for any large-scale DVR. It is also important to reiterate that in this implementation there is a separation between ESS information and the actual volume data insofar that the

sampling and the space-skipping do not necessarily need to share the same data — a trait also present in Hadwiger's works [5, 6].

In addition to the presented non-illuminated or locally illuminated rendering, the author believes that this approach could augment global-illumination — more specifically soft shadows — in volume rendering by using BVH leaf information to quickly perform any-hit shadow calculations, an approach seen in Lacewell et al's 2008 paper [109] for surface geometry.

This chapter has been presented as a full-length paper co-published in both the High-Performance Graphics conference 2019 [15] and a special issue of Computer Graphics Forum [16].

Chapter 7

Conclusions and Future Work

At this point, in chapters 3 to 6, the investigatory work has been presented, each with their own individual summaries and brief conclusions. In this chapter, a reiteration of these works and their results is presented, with the goal of giving more concrete conclusions with an outlook on how this work can be expanded upon in the future.

GPU Cache Performance

In chapter 3 the thesis' investigations begin with evaluation of volume-rendering relevant components of the GPU memory hierarchy, which helps give a solid platform upon which the later chapters are built. It profiles texture accesses in different configurations, both in 2D and 3D texture space. This investigation highlighted the importance of prioritising cache-coherence when designing a GPU solution. Granted, the current state of GPU architecture is geared towards instruction-level-parallelism style latency hiding for memory references, but this can only go so far, especially with the shader complexity and register pressure needed for DVR. Therefore, targeting efficient cache performance is paramount. Ideally, optimised L1 cache accesses would be the gold standard, however in the context of volume rendering, the L1 doesn't provide a huge amount of practical real-estate for data to be rendered to screen. It follows then that the L2 provides a good balance between available footprint and cache-hit performance. Chapter 3 verifies this, along with evidence of 3D texture rearrangement for better spatio-temporal coherence, which is important to keep in mind for volumetric data accesses. This worked profiled these statistics on an Nvidia Quadro K2200 — the

Maxwell architecture, even though it has the K-prefix — and even though the general memory hierarchy architecture hasn't been totally redesigned, the author would encourage those that take inspiration from this work to perform the same evaluation on later models.

View Dependent Scheduling

In chapter 4 the performance evaluation work in chapter 3 was used to formulate a view-dependent cache-coherent DVR approach. While output-sensitive filtering approaches aren't new in the realm of volume rendering, they generally required some form of feedback loop between the CPU and the GPU. This meant that neither system could be fully utilised in parallel. This work presented an approach that can be pipelined such that the CPU can work independently, feeding view-dependent scheduling information to the GPU. This view-dependent information is targeted towards L2 cache optimisation for sampling the volume, by completing all samples in an L2-sized region of the volume before continuing to the next region. The results of this work showed better GPU cache performance, but came at the cost of both CPU and GPU overhead for generating and consuming the scheduling information respectively. As the brick-size reduced, the complexity of managing determining the view-dependent working set on the CPU in a serial fashion became too much of a burden and became the major bottleneck.

Light Field DVR

Following on from this, this improved cache performance approach while sharing the load with the CPU could be utilised for multi-view volume rendering. In chapter 5, the L2 cache on the GPU is still targeted as a source of performance gain. This time, each region is projected to all views in a light field rendering simultaneously, before moving on to the next region. The scheduling information was again generated on the CPU, but due to the additional work that the GPU had the do, it no longer was a bottle neck. Furthermore, in order to allow each region to render to all views, there must be support for out-of-order rendering and therefore multiple off-screen buffers for each region and view. A greedy algorithm to minimize the amount of off-screen buffers and reduce memory footprint was used to mitigate this issue to good success. Improvements are

shown in the overall render performance of light field volume rendering, resulting from much better cache coherency. Having said this, as the region count grew — by either the volume growing or the regions becoming smaller — the overhead again becomes a bottleneck.

It's worth noting that this investigation used a fairly naïve approach for the CPU-stage scheduling information generation and off-screen buffer minimisation, and could be improved to become less of a bottleneck in future work. Another potential stream of follow-on investigation could look at using this efficient multi-view scheduling approach in global-illumination to generate multiple shadow maps for light-sources in a scene surrounding a volume.

It might also be possible to further exploit frame-to-frame coherency both in terms of view-list generation and memory allocation, or perhaps leverage a light-weight easy-to-update data-structure to accelerate the working-set determination. Fundamentally, more work also needs to be done in the I/O domain, potentially integrating this work with brick-based compression streaming techniques.

Clustered BVH Empty-Space-Skipping on GPU

Chronologically, during work presented in chapter 5, Nvidia announced their RTCore technology, and hardware-accelerated ray-tracing enabled line of GPUs in the form of RTX. This presented an interesting opportunity to study BVHs in the context of ESS for volume rendering on GPUs. Previously, evaluations of BVHs for DVR had only been done on CPUs citing build times and traversal complexity in contrast to relatively simple octree traversal logic. Therefore, in chapter 6 an evaluation of using BVHs on ray-tracing hardware enabled GPUs is presented. This is compared to a state-of-the-art approach — Sparseleap [6] — which rasterizes tight-fitting proxy geometry around active volume regions, using a hierarchical space-cutting approach with an ESS octree. It is shown that BVH build times are actually quite quick which is suitable for user interaction with a transfer-function, and rendering times are substantially better when using the ray-tracing hardware and BVH ESS.

While this was true for relatively sparse geometry like the flower dataset, a densely compacted dataset such as the supernova had a redundant amount of BVH leaf nodes which was found to have dramatically increased rendering times due to ESS

depth complexity. The natural course to follow was to then reduce the leaf-node count in dense regions of the volume by performing a clustering step before building the BVH structure. This was shown to both decrease the BVH build time and increase render performance, further pushing the gain against the state-of-the-art.

In essence, this is the main contribution of this thesis; an analysis that BVHs *can* be very viable alternative to ESS data-structures on GPUs, especially when some clustering approach is applied beforehand. But this is — of course — not the be all and end all for ESS data-structures, or even BVH ESS implementations. For instance, the primary method of building BVHs is generally grounded on the principal of a surface-area-heuristic, which is focused more towards surface representations of geometry. This is not necessarily the best approach for volumetric information at all, however the RTX and OptiX APIs don't (at the time of writing) provide a way of building custom BVH data-structures with arbitrary heuristics for grouping. As such, this would be an interesting topic to investigation.

Additionally, accelerating global-illumination using BVHs has been seen in the realm of soft shadows for surface geometry [109], and using the ESS BVH tree in the same fashion for volumetric soft shadows could be another use, potentially in real-time applications like games that use dynamic smoke or clouds.

Summary

To finally conclude this thesis, evaluations and investigations have been presented on GPU cache performance in the context of volume rendering, a hybrid architecture using both the CPU and GPU for pipelined view-dependent DVR on a single-view target, which was then adapted to light field technologies with success. Then following the theme of efficient use of GPUs for DVR, new ray-tracing hardware was evaluated for empty space skipping to the benefit of final render performance. The implementation from this final piece of work in chapter 6 has been made publicly available at *gitgub.com/ganterd/optixdvr*.

Appendix A

DVR Sampling Visualisation Tool

While the author has covered predictable patterns of texture sampling in chapter 3, this thesis has an application in mind it is worth investigating the actual sampling pattern of DVR. As part of this work, a tool was built to visualise the access pattern of DVR, showing a 3D heat-map of the voxels in the volume and indicating which voxels aren't touched at all.

Images of this program at work are shown in figures A.1 and A.2. In figure A.1, a slice of voxels are shown after rendering the supernova dataset. Since this dataset in particular has a large amount of opaque structure, it can be seen that rays can benefit from early-ray-termination (ERT) as the voxels in the middle of the volume are effectively not sampled at all. Figure A.2 shows the voxels that are not touched by rays at all in green. This layer is the farthest in the volume from the camera. It is interesting to notice the lattice pattern that appears due to rays separating, effectively aliasing data at a distance.

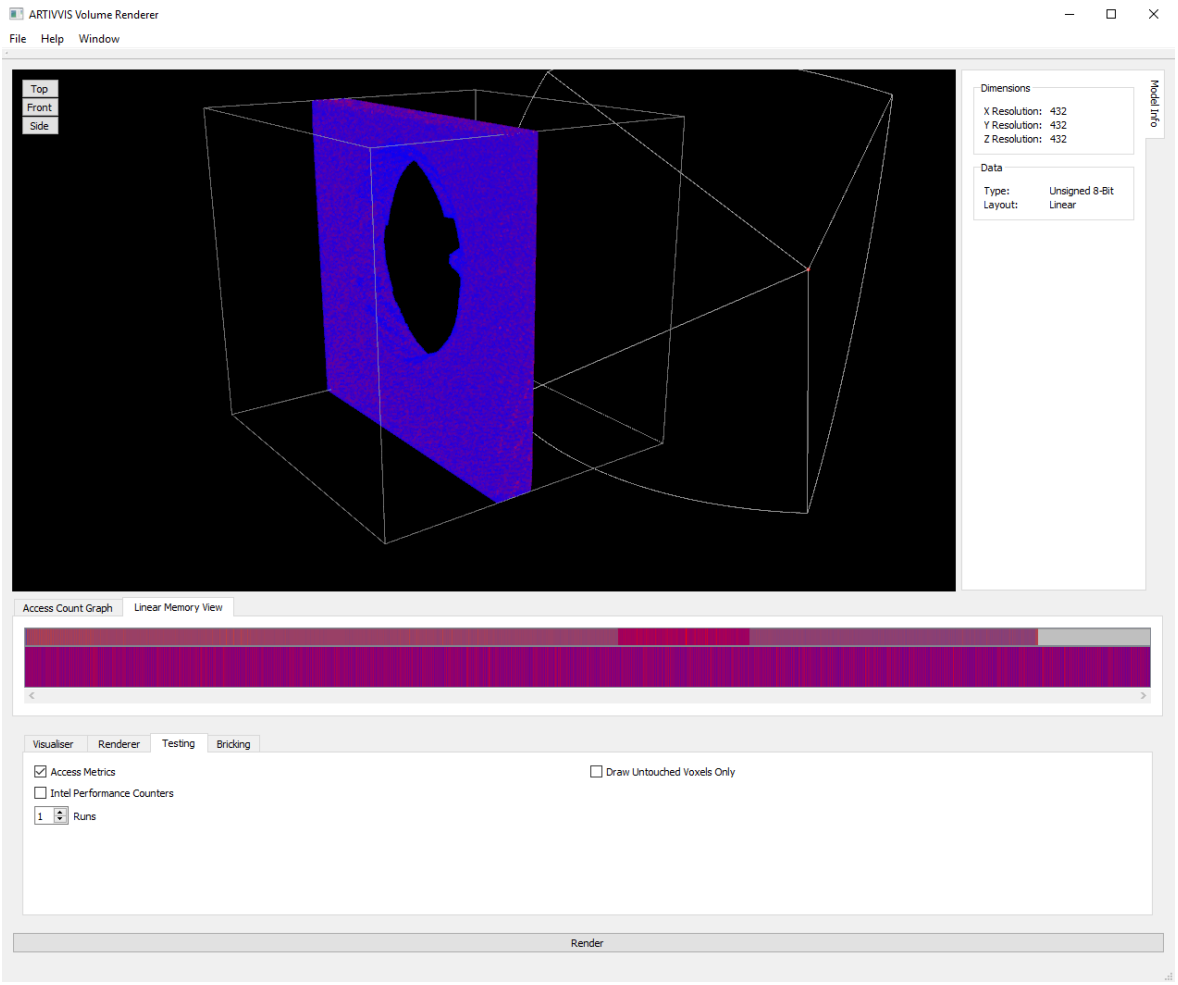


Figure A.1: Screen shot of the Memory Access Visualisation tool built as part of this work to visualise the access patterns of direct volume rendering. In this example, a portion of the accessed voxels of the Supernova dataset

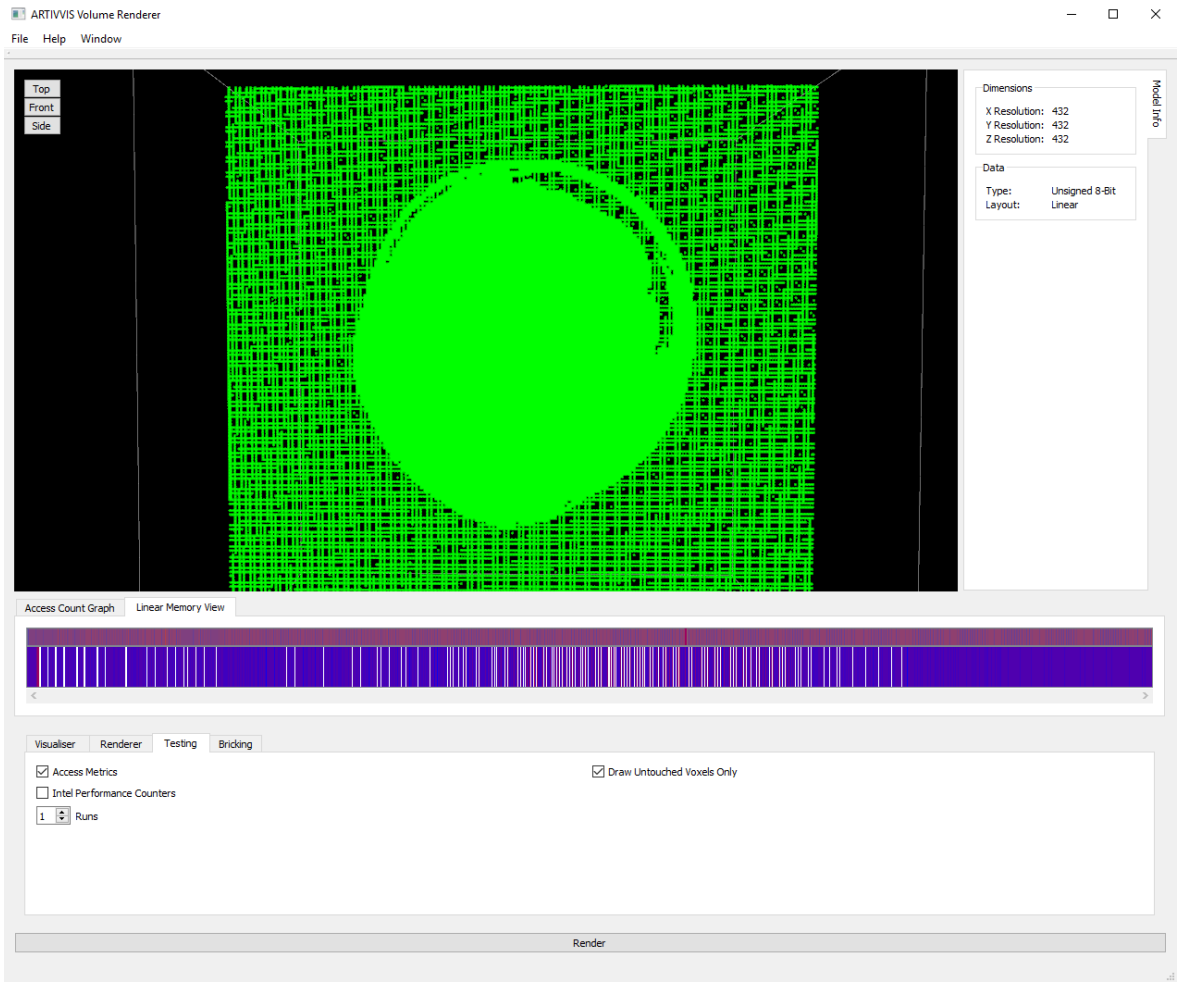


Figure A.2: Memory Access Visualisation tool showing a portion of the unsampled voxels of the Supernova data-set. This shows both the effect of ERT and spread as rays get further into the volume.

Appendix B

Datasets

In this appendix the author provides a list of the volume datasets most commonly used over the duration of this thesis. A detailed description of each dataset is given along with traits that exhibit different performance characteristics and problems for DVR. While other datasets may have been used during the research, these were the main test subjects.

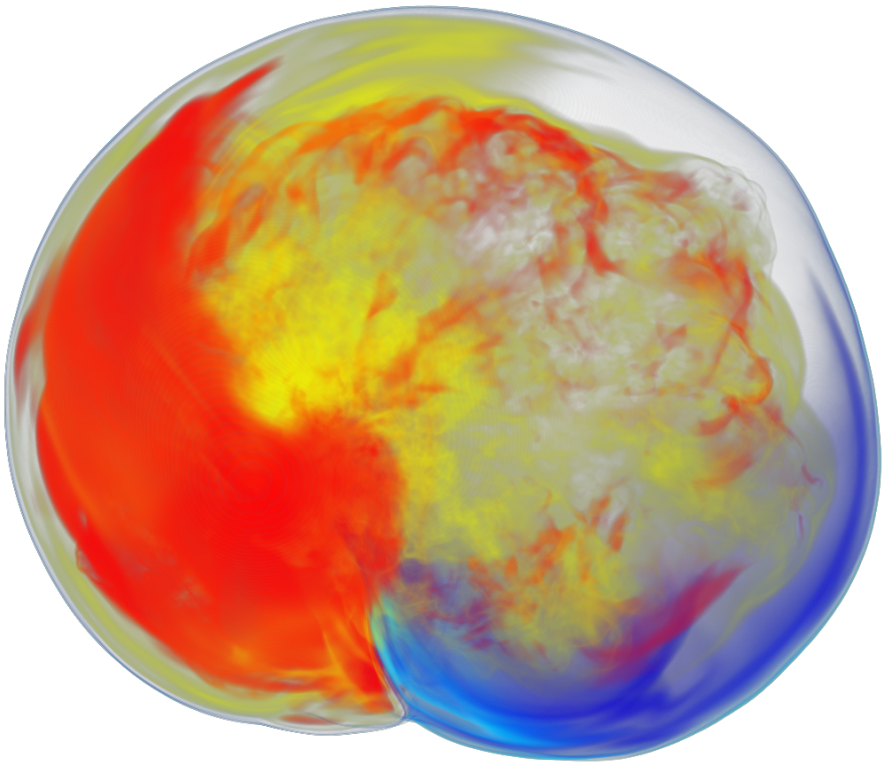


Figure B.1: Example rendering of a frame the Supernova dataset [98].

Supernova

The Supernova dataset [98] shown in figure B.1 is one of the most substantial time-varying datasets this project used, and represents a simulation of the few moments of a supernova. The dimensions of the volume are 432^3 32-bit floating-point values, with over 300 individual frames of data. Variations of this dataset were created with differing data types (e.g. 8-bit unsigned integers) and up-scaled dimensions.

This dataset contains a large amount of empty-space around the sphere-like entity and — in general — regardless of the transfer function, there are many regions of the volume that would generate large groups of active sub-division leaves.

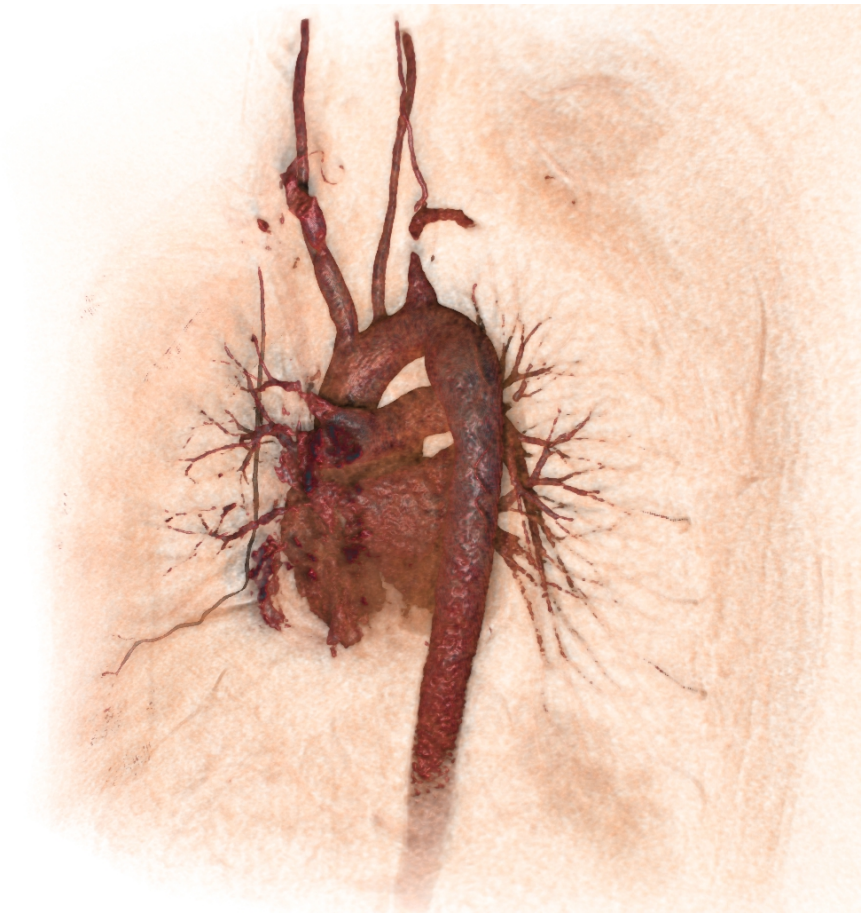


Figure B.2: Example rendering of the Subclavia dataset, provided as part of the open-source Inviwo application.

Subclavia

This dataset, shown in figure B.2 represents an anatomical CT scan of the heart and the subclavian arteries. This dataset is provided as part of the open-source Inviwo visualisation application [99]. The volume itself is one of the smaller datasets used at a resolution of 512x512x96 voxels of 8-bit unsigned integers. Depending on the transfer function used, this dataset can exhibit an interesting problem for ESS, as long, thin strands of data are prevalent throughout the volume, with much empty space in between.

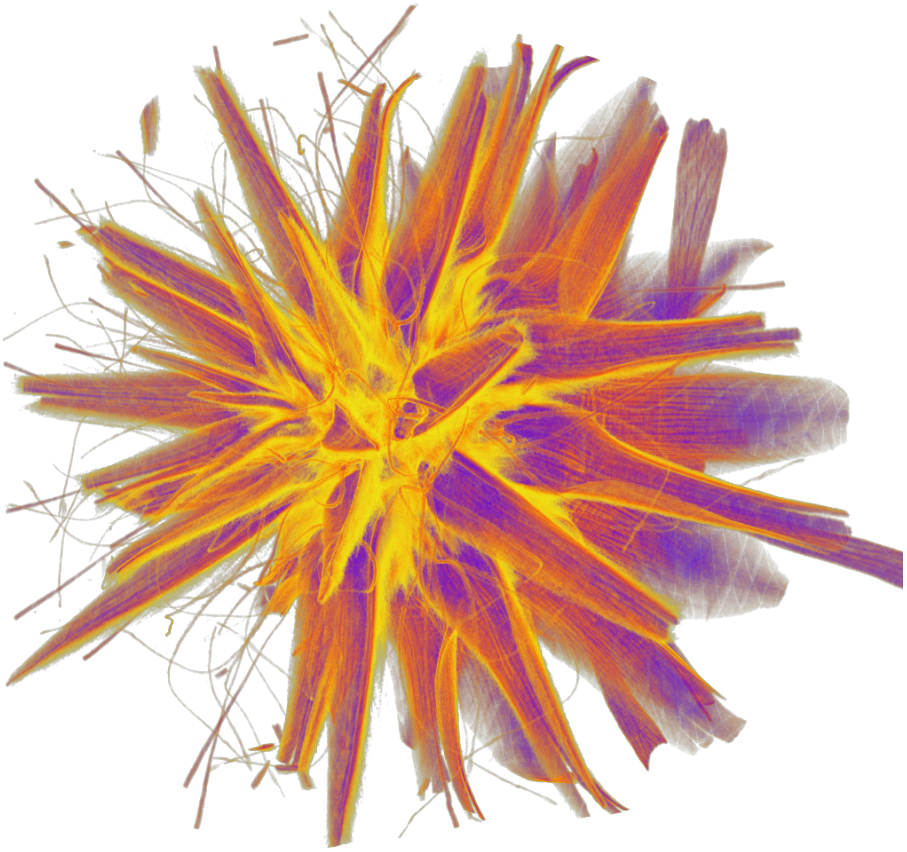


Figure B.3: Example rendering of the UZH Flower dataset.

UZH Flower

The Flower dataset is provided by University of Zurich (UZH) as part of an open-access libary of μ-CT scanned objects [108]. This particular dataset is a μ-CT of a *leucadendron rubrum* flower. This, along with the beechnut dataset discussed later, is one of the most high-resolution datasets used in this project without being up-scaled. The volume dimensions are 1024^3 voxels of 8-bit unsigned integers, measuring at exactly 1GB of data. The volume exhibits a large amount of empty-space, but due to the complex construction of the flower, the amount of active regions in the volume rarely changes much, thanks to the thin strands of plant media present.

Figure B.4: Example rendering of the UZH beechnut μ-CT dataset.

UZH Beechnut

This μ-CT of a dried beechnut is also provided as part of the UZH research dataset library [108]. Larger in both dimensions and bit-depth than the flower dataset, the beechnut μ-CT volume is 1024x1024x1546 with 16-bit unsigned integer voxels.

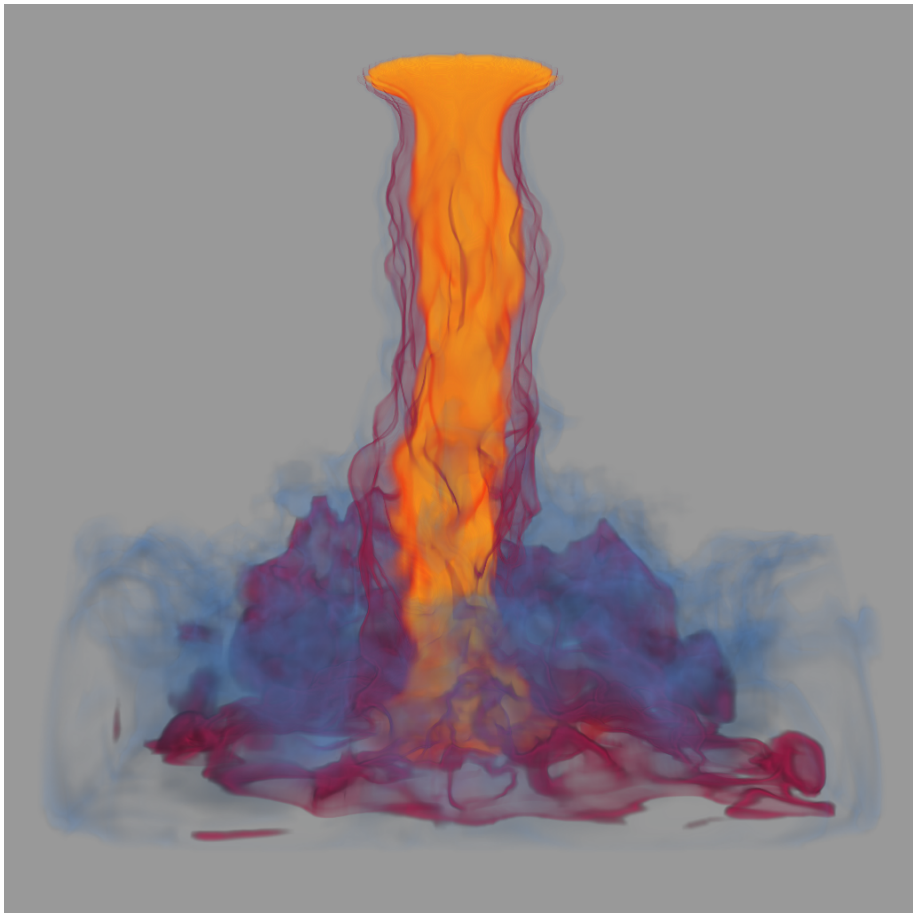


Figure B.5: Example rendering of a single frame from the smoke simulation dataset.

Smoke Simulation

This dataset is one of the only datasets created as part of the project in which this thesis was a part of, used primarily in Noonan et al's work [35]. The smoke simulation volume is a relatively small, time-varying volume of 256^3 8-bit unsigned integer voxels with 300 individual frames of data.

Bibliography

[1] M. Agus, E. Gobbetti, J. A. I. Guitián, F. Marton, and G. Pintore, "GPU accelerated direct volume rendering on an interactive light field display," *Computer Graphics Forum*, vol. 27, no. 2, pp. 231–240, 2008.

[2] E. Gobbetti, F. Marton, and J. A. Iglesias Guitián, "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *Visual Computer*, vol. 24, no. 7-9, pp. 797–806, 2008.

[3] J. A. I. Guitián, E. Gobbetti, F. Marton, J. A. Iglesias?Guitián, E. Gobbetti, and F. Marton, "View-dependent exploration of massive volumetric models on large-scale light field displays," in *Visual Computer*, vol. 26, pp. 1037–1047, jun 2010.

[4] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "GigaVoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D '09*, I3D '09, (New York, NY, USA), p. 15, ACM, 2009.

[5] M. Hadwiger, J. Beyer, W. K. Jeong, and H. Pfister, "Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 2285–2294, dec 2012.

[6] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister, "SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 974–983, 2018.

[7] L.-G. Factory, "Looking-glass product website." https://lookingglassfactory.com/products/the-looking-glass/. Accessed: 2019-09-23.

[8] Avegant, "Avegant light-field technology web page." https://www.avegant.com/light-field. Accessed: 2019-09-23.

[9] Daqri, "Daqri web page." https://daqri.com/. Accessed: 2019-09-23.

[10] Holografika, "Holografika web page." https://holografika.com/. Accessed: 2019-09-23.

[11] T. M. Peters, C. J. Henri, P. Munger, A. M. Takahashi, A. C. Evans, B. Davey, and A. Olivier, "Integration of stereoscopic DSA and 3D MRI for image-guided neurosurgery," *Computerized Medical Imaging and Graphics*, vol. 18, no. 4, pp. 289–299, 1994.

[12] T. Sielhorst, C. Bichlmeier, S. M. Heining, and N. Navab, "Depth Perception âĂŞ A Major Issue in Medical AR: Evaluation Study by Twenty Surgeons," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2006* (R. Larsen, M. Nielsen, and J. Sporring, eds.), (Berlin, Heidelberg), pp. 364–372, Springer Berlin Heidelberg, 2006.

[13] Nvidia, "Nvidia turing architecture whitepaper." https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. Accessed: 2019-01-06.

[14] D. Ganter, D. Hardman, and M. Alain, "Light-Field DVR on GPU for Streaming Time-Varying Data," in *Pacific Graphics*, 2018.

[15] D. Ganter and M. Manzke, "An analysis of region clustered bvh volume rendering on gpu," in *High-Performance Graphics 2019*, The Eurographics Association and John Wiley & Sons Ltd., 2019.

[16] D. Ganter and M. Manzke, "An analysis of region clustered bvh volume rendering on gpu," *Computer Graphics Forum*, vol. 38, no. 8, pp. 13–21, 2019.

[17] S. Bruton., D. Ganter., and M. Manzke., "Synthesising light field volumetric visualizations in real-time using a compressed volume representation," in *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 2: IVAPP,*, pp. 96–105, INSTICC, SciTePress, 2019.

[18] S. Martin, S. Bruton., D. Ganter., and M. Manzke., "Using a depth heuristic for light field volume rendering," in *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: GRAPP,*, pp. 134–144, INSTICC, SciTePress, 2019.

[19] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, pp. 29–37, may 1988.

[20] T. Porter and T. Duff, "Compositing digital images," in *ACM Siggraph Computer Graphics*, vol. 18, pp. 253–259, ACM, 1984.

[21] K. Engel, M. Kraus, and T. Ertl, "High-quality Pre-integrated Volume Rendering Using Hardware-accelerated Pixel Shading," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '01, (New York, NY, USA), pp. 9–16, ACM, 2001.

[22] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "A Data Distributed, Parallel Algorithm for Ray-traced Volume Rendering," in *Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, (New York, NY, USA), pp. 15–22, ACM, 1993.

[23] W. M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," in *Proceedings of the 1993 symposium on Parallel rendering - PRS '93*, PRS '93, (New York, NY, USA), pp. 7–14, ACM, 1993.

[24] A. Law and R. Yagel, "Multi-Frame Thrashless Ray Casting with Advancing Ray-Front," in *Proceedings of the Conference on Graphics Interface '96*, vol. i of *GI '96*, (Toronto, Ont., Canada, Canada), pp. 70–77, Canadian Information Processing Society, 1996.

[25] K.-L. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel Volume Rendering Using Binary-Swap Compositing," *IEEE Computer Graphics and Applications*, vol. 14, pp. 59–68, jul 1994.

[26] M. E. Palmer, S. Taylor, and B. Totty, "Exploiting Deep Parallel Memory Hierarchies for Ray Casting Volume Rendering," in *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '97, (New York, NY, USA), pp. 15—-ff., ACM, 1997.

[27] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh, "Hybrid sort-first and sort-last parallel rendering with a cluster of PCs," in *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (HWWS '00)*, HWWS '00, (New York, NY, USA), pp. 97–108, ACM, 2000.

[28] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller, "A refined data addressing and processing scheme to accelerate volume raycasting," *Computers & Graphics*, vol. 28, no. 5, pp. 719–729, 2004.

[29] B. Mora, J. P. Jessel, and R. Caubet, "A New Object-order Ray-casting Algorithm," in *Proceedings of the Conference on Visualization '02*, VIS '02, (Washington, DC, USA), pp. 203–210, IEEE Computer Society, 2002.

[30] W. Hong, F. Qiu, and A. Kaufman, "GPU-based object-order ray-casting for large datasets," in *Fourth International Workshop on Volume Graphics, 2005.*, pp. 177–240, jun 2005.

[31] W. Usher, I. Wald, J. Amstutz, J. GÃijnther, C. Brownlee, and V. Pascucci, "Scalable ray tracing using the distributed framebuffer," *Computer Graphics Forum*, vol. 38, no. 3, pp. 455–466, 2019.

[32] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navrátil, "Ospray-a cpu ray tracing framework for scientific visualization," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 931–940, 2016.

[33] J. Clyne and J. Dennis, "Interactive Direct Volume Rendering of Time-Varying Data," in *Proceedings of the 1st Joint IEEE VGTC - EUROGRAPHICS Sym-*

posium on Visualization (VisSym˜'99) (E. Gröller, H. Löffelmann, and W. Ribarsky, eds.), (Vienna), pp. 109–120, Springer Vienna, 1999.

[34] M. Balsa Rodríguez, E. Gobbetti, J. A. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. K. Suter, "State-of-the-art in compressed GPU-based direct volume rendering," *Computer Graphics Forum*, vol. 33, no. 6, pp. 77–100, 2014.

[35] T. Noonan, L. Campoalegre, and J. Dingliana, "Temporal Coherence Predictor for Time Varying Volume Data Based on Perceptual Functions," in *Vision, Modeling & Visualization* (D. Bommes, T. Ritschel, and T. Schultz, eds.), pp. 33–40, The Eurographics Association, 2015.

[36] R. Shekhar and V. Zagrodsky, "Cine MPR: Interactive Multiplanar Reformatting of Four-Dimensional Cardiac Data Using Hardware-Accelerated Texture Mapping," *IEEE Transactions on Information Technology in Biomedicine*, vol. 7, pp. 384–393, dec 2003.

[37] Q. Zhang, R. Eagleson, and T. M. Peters, "Dynamic real-time 4D cardiac MDCT image display using GPU-accelerated volume rendering," *Computerized Medical Imaging and Graphics*, vol. 33, no. 6, pp. 461–476, 2009.

[38] N. Morrical, W. Usher, and I. Wald, "Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing," vol. 1.

[39] Nvidia, "Nvidia gm107 architecture whitepaper." http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf. Accessed: 2019-01-06.

[40] M. J. Doyle, C. Fowler, and M. Manzke, "A hardware unit for fast SAH-optimised BVH construction," *ACM Transactions on Graphics*, vol. 32, p. 1, jul 2013.

[41] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "SGRT: a mobile GPU architecture for real-time ray tracing,"

in *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13*, (New York, New York, USA), pp. 109–119, ACM Press, 2013.

[42] M. J. Doyle, C. Tuohy, and M. Manzke, "Evaluation of a BVH construction accelerator architecture for high-quality visualization," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, pp. 83–94, jan 2018.

[43] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, no. 3, pp. 136–141, 2012.

[44] X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy Through Microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 72–86, jan 2017.

[45] G. Knittel, "The ULTRAVIS System," in *IEEE Symposium on Volume Visualization, VV 2000*, pp. 71–79, oct 2000.

[46] J. Mensmann, T. Ropinski, and K. H. Hinrichs, "An Advanced Volume Raycasting Technique using GPU Stream Processing," in *International Conference on Computer Graphics Theory and Applications (GRAPP '10)*, pp. 190–198, 2010.

[47] E. W. Bethel and M. Howison, "Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning," *International Journal of High Performance Computing Applications*, vol. 26, pp. 399–412, nov 2012.

[48] M. Labschutz, S. Bruckner, M. E. Groller, M. Hadwiger, and P. Rautek, "JiT-Tree: A Just-in-Time Compiled Sparse GPU Volume Data Structure," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, pp. 1025–1034, jan 2016.

[49] T. Fogal and J. H. Krüger, "Tuvok - An Architecture for Large Scale Volume Rendering," in *15th Vision, Modeling and Visualization Workshop '10*, pp. 139–146, 2010.

[50] T. Fogal, A. Schiewe, and J. Kruger, "An analysis of scalable GPU-based ray-guided volume rendering," in *IEEE Symposium on Large Data Analysis and Visualization 2013, LDAV 2013 - Proceedings*, pp. 43–51, IEEE, oct 2013.

[51] B. Liu, G. J. Clapworthy, F. Dong, and E. C. Prakash, "Octree rasterization: Accelerating high-quality out-of-core GPU volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, pp. 1732–1745, oct 2013.

[52] R. K. Hoetzlein, "GVDB: Raytracing Sparse Voxel Database Structures on the GPU," *High Performance Graphics*, pp. 109–117, 2016.

[53] K. Museth, "VDB: high-resolution sparse volumes with dynamic topography," *ACM Transactions on Graphics*, vol. 32, no. 3, pp. 1–22, 2013.

[54] OpenVDB, "OpenVDB."

[55] K. Museth, "Hierarchical digital differential analyzer for efficient ray-marching in OpenVDB," vol. 3, pp. 1–1, 2014.

[56] S. Laine and T. Karras, "Efficient sparse voxel octrees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 8, pp. 1048–1059, 2011.

[57] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, vol. 2005, (New York, New York, USA), pp. 15–22, ACM Press, 2005.

[58] A. Knoll, S. Thelen, I. Wald, C. D. Hansen, H. Hagen, and M. E. Papka, "Full-resolution interactive CPU volume rendering with coherent BVH traversal," in *IEEE Pacific Visualization Symposium 2011, PacificVis 2011*, pp. 3–10, IEEE, 2011.

[59] S. Zellmann, J. P. Schulze, and U. Lang, "Binned k-d tree construction for sparse volume data on multi-core and gpu systems," *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–1, 2019.

[60] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: A General Purpose Ray Tracing Engine," *ACM Transactions on Graphics*, vol. 29, no. 4, p. 1, 2010.

[61] M. Meißner, M. Doggett, J. Hirche, and U. Kanus, "Efficient space leaping for ray casting architectures," in *Volume Graphics 2001* (K. Mueller and A. E. Kaufman, eds.), (Vienna), pp. 149–161, Springer Vienna, 2001.

[62] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa, "Vizard ii: A reconfigurable interactive volume rendering system," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, (Aire-la-Ville, Switzerland, Switzerland), pp. 137–146, Eurographics Association, 2002.

[63] M. C. Doggett and G. R. Hellestrand, "A hardware architecture for video rate smooth shading of volume data," *Computers & Graphics*, vol. 19, no. 5, pp. 695 – 704, 1995.

[64] M. Levoy and P. Hanrahan, "Light field rendering," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*, SIGGRAPH '96, (New York, NY, USA), pp. 31–42, ACM, 1996.

[65] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, "The lumigraph," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*, (New York, New York, USA), pp. 43–54, ACM Press, 1996.

[66] A. Isaksen, L. McMillan, and S. J. Gortler, "Dynamically reparameterized light fields," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*, SIGGRAPH '00, (New York, NY, USA), pp. 297–306, ACM Press/Addison-Wesley Publishing Co., 2000.

[67] G. Wetzstein, D. R. Lanman, M. W. Hirsch, and R. Raskar, "Tensor displays: compressive light field synthesis using multilayer displays with directional backlighting," *ACM Transactions on Graphics*, vol. 31, pp. 1–11, jul 2012.

[68] F.-C. Huang, D. Luebke, G. Wetzstein, K. Chen, and G. Wetzstein, "The Light Field Stereoscope: Immersive Computer Graphics via Factored Near-eye Light Field Displays with Focus Cues," *ACM SIGGRAPH 2015 Emerging Technologies on - SIGGRAPH '15*, vol. 34, pp. 60:1—-60:12, jul 2015.

[69] N. A. Dodgson, "Autostereoscopic 3D Displays," *Computer*, vol. 38, pp. 31–36, aug 2005.

[70] J. A. I. Guitián and J. A. Iglesias Guitian, *Real-time GPU-accelerated Out-of-Core Rendering and Light-field Display Visualization for Improved Massive Volume Understanding*. PhD thesis, Iglesias Guitian, Jose A., 2011.

[71] F. Marton, M. Agus, E. Gobbetti, G. Pintore, and M. B. Rodriguez, "Natural exploration of 3D massive models on large-scale light field displays using the FOX proximal navigation technique," *Computers & Graphics*, vol. 36, no. 8, pp. 893–903, 2012.

[72] E. Yuasa, F. Sakaue, and J. Sato, "Generating 5D Light Fields in Scattering Media for Representing 3D Images," in *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, vol. 2017-July, pp. 1287–1294, jul 2017.

[73] J. Geng, "A volumetric 3D display based on a DLP projection engine," *Displays*, vol. 34, no. 1, pp. 39–48, 2013.

[74] B. Mora, R. Maciejewski, M. Chen, and D. S. Ebert, "Visualization and Computer Graphics on Isotropically Emissive Volumetric Displays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 221–234, mar 2009.

[75] S. K. Rushton and P. M. Riddell, "Developing visual systems and exposure to virtual reality and stereo displays: some concerns and speculations about the demands on accommodation and vergence," *Applied Ergonomics*, vol. 30, no. 1, pp. 69–78, 1999.

[76] G. Kramida, "Resolving the Vergence-Accommodation Conflict in Head-Mounted Displays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, pp. 1912–1931, jul 2016.

[77] G.-A. Koulieris, B. Bui, M. S. Banks, and G. Drettakis, "Accommodation and Comfort in Head-mounted Displays," *ACM Trans. Graph.*, vol. 36, pp. 87:1—-87:11, jul 2017.

[78] J. E. Cutting and P. M. Vishton, "Potency, and contextual use of different information about depth," *Perception of space and motion*, vol. 69, 1995.

[79] A. Bulbul, Z. Cipiloglu, and T. Capin, "A perceptual approach for stereoscopic rendering optimization," *Computers & Graphics*, vol. 34, no. 2, pp. 145–157, 2010.

[80] D. Lanman and D. Luebke, "Near-eye Light Field Displays," *ACM Transactions on Graphics*, vol. 32, pp. 1–10, nov 2013.

[81] S. J. Adelson and C. D. Hansen, "Fast Stereoscopic Images with Ray-traced Volume Rendering," in *Proceedings of the 1994 Symposium on Volume Visualization*, VVS '94, (New York, NY, USA), pp. 3–9, ACM, 1994.

[82] T. He and A. Kaufman, "Fast stereo volume rendering," in *Visualization '96. Proceedings.*, pp. 49–56, oct 1996.

[83] Y.-M. M. Koo, C.-H. H. Lee, and Y.-G. G. Shin, "Object-order template-based approach for stereoscopic volume rendering," *The Journal of Visualization and Computer Animation*, vol. 10, no. 3, pp. 133–142, 1999.

[84] R. Yagel and A. Kaufman, "Template-Based Volume Viewing," *Computer Graphics Forum*, vol. 11, no. 3, pp. 153–167, 1992.

[85] M. Wan, N. Zhang, H. Qu, and A. E. Kaufman, "Interactive stereoscopic rendering of volumetric environments," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, pp. 15–28, jan 2004.

[86] D. Ruijters, "Dynamic resolution in GPU-accelerated volume rendering to autostereoscopic multiview lenticular displays," *Eurasip Journal on Advances in Signal Processing*, vol. 2009, 2009.

[87] D. Ruijters, S. Zinger, L. Do, and P. H. N. de With, "Latency optimization for autostereoscopic volumetric visualization in image-guided interventions," *Neurocomputing*, vol. 144, pp. 119–127, 2014.

[88] T. Hübner and R. Pajarola, "Single-pass multi-view volume rendering," in *Proceedings of International Conference Computer Graphics and Visualization*, 2007.

[89] K.-C. Kwon, C. Park, M.-U. Erdenebat, J.-S. Jeong, J.-H. Choi, N. Kim, J.-H. Park, Y.-T. Lim, and K.-H. Yoo, "High speed image space parallel processing for computer-generated integral imaging system," *Optics Express*, vol. 20, p. 732, jan 2012.

[90] B. Battin, G. Valette, J. Lehuraux, Y. Remion, and L. Lucas, "A premixed autostereoscopic OptiX-based volume rendering," in *2015 International Conference on 3D Imaging, IC3D 2015 - Proceedings*, vol. 31 of *SIGGRAPH '11*, (New York, NY, USA), pp. 3–10, IEEE, mar 2016.

[91] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 235–246, mar 2010.

[92] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-Art in GPU-Based Large-Scale Volume Visualization," *Computer Graphics Forum*, vol. 34, no. 8, pp. 13–37, 2015.

[93] M. Isenburg, P. Lindstrom, and H. Childs, "Parallel and streaming generation of ghost data for structured grids," *IEEE Computer Graphics and Applications*, no. 3, pp. 32–44, 2010.

[94] W. Li, K. Mueller, A. E. Kaufman, Wei Li, K. Mueller, and A. E. Kaufman, "Empty space skipping and occlusion clipping for texture-based volume rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, vol. 4400, p. 42, IEEE Computer Society, 2003.

[95] C. Rezk-Salama, S. Todt, and A. Kolb, "Raycasting of light field galleries from volumetric data," *Computer Graphics Forum*, vol. 27, no. 3, pp. 839–846, 2008.

[96] C. Birklbauer and O. Bimber, "Light-field supported fast volume rendering," *ACM SIGGRAPH 2012 Posters on - SIGGRAPH '12*, p. 1, 2012.

[97] S. Opelt and O. Bimber, "Light-field caching," in *ACM SIGGRAPH 2011 Posters on - SIGGRAPH '11*, (New York, New York, USA), p. 1, ACM Press, 2011.

[98] J. Blondin, "Supernova Modelling," 2007.

[99] D. Jönsson, P. Steneteg, E. Sundén, R. Englund, S. Kottravel, M. Falk, A. Ynnerman, I. Hotz, and T. Ropinski, "Inviwo - a visualization system with usage abstraction levels," *IEEE Transactions on Visualization and Computer Graphics*, 2019.

[100] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, D. Weiskopf, C. Rezk-Salama, and D. Weiskopf, *Real-time volume graphics*. CRC Press, 2006.

[101] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil, "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, pp. 931–940, jan 2017.

[102] Q. Wu, W. Usher, S. Petruzza, S. Kumar, F. Wang, I. Wald, V. Pascucci, and C. D. Hansen, "VisIt-OSPRay : Toward an Exascale Volume Visualization System," *Eurographics Symposium on Parallel Graphics and Visualization*, vol. Vi, 2018.

[103] A. Knoll, I. Wald, P. A. Navrátil, M. E. Papka, and K. P. Gaither, "Ray tracing and volume rendering large molecular data on multi-core and many-core architectures," in *Proceedings of the 8th International Workshop on Ultrascale Visualization - UltraVis '13*, (New York, New York, USA), pp. 1–8, ACM Press, 2013.

[104] A. Knoll, I. Wald, P. Navratil, A. Bowen, K. Reda, M. E. Papka, and K. Gaither, "RBF volume ray casting on multicore and manycore CPUs," *Computer Graphics Forum*, vol. 33, pp. 71–80, jun 2014.

[105] D. Ruijters and A. Vilanova, "Optimizing GPU Volume Rendering," *Winter School of Computer Graphics (WSCG)*, vol. 14, 2006.

[106] F. C. Crow, "Summed-area tables for texture mapping," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 207–212, 1984.

[107] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-Time Concurrent Linked List Construction on the GPU," *Computer Graphics Forum*, vol. 29, pp. 1297–1304, aug 2010.

[108] UZH, "Research datasets." https://www.ifi.uzh.ch/en/vmml/research/datasets.html. Accessed: 2019-01-05.

[109] D. Lacewell, B. Burley, S. Boulos, and P. Shirley, "Raytracing prefiltered occlusion for aggregate geometry," in *2008 IEEE Symposium on Interactive Ray Tracing*, pp. 19–26, IEEE, 2008.