TU WIEN Informatics

Scalable Web-Based Multi-Volume Rendering

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Lukas Herzberger, BSc B.A.
Matrikelnummer 01006039

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing.(FH) Dr.techn. Johanna Beyer

Wien, 13. Juli 2023

Lukas Herzberger	Eduard Gröller

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Scalable Web-Based Multi-Volume Rendering

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Lukas Herzberger, BSc B.A.
Registration Number 01006039

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing.(FH) Dr.techn. Johanna Beyer

Vienna, 13th July, 2023

Lukas Herzberger	Eduard Gröller

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Lukas Herzberger, BSc B.A.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. Juli 2023

Lukas Herzberger

Danksagung

An erster Stelle möchte ich mich bei meiner Betreuerin Johanna Beyer für die kontinuierliche Unterstützung während der Arbeit an meiner Diplomarbeit bedanken. Deine Leidenschaft für die Forschung war für mich eine unschätzbare Inspiration. Ich konnte mich jederzeit an dich wenden, wenn ich vor Herausforderungen stand. Darüber hinaus möchte ich dir einfach dafür danken, eine Freundin zu sein, mir die coolsten Orte in Boston gezeigt zu haben und generell für einen wirklich coolen Sommer.

Ich möchte auch Markus Hadwiger für seine hervorragende Betreuung, Unterstützung und die wertvolle Zusammenarbeit danken. Deine Herzlichkeit und dein Vertrauen in meine Fähigkeiten haben mir wirklich viel Kraft gegeben.

Ich möchte Meister Eduard Gröller für seine großartige Betreuung danken. Vielen Dank, dass du mich ohne zu zögern unterstützt hast, dass du mich in deine Research Unit aufgenommen hast und dass du mir wertvolle Einblicke in die wunderbare Welt der Wissenschaft zu Teil werden hast lassen.

Ich möchte Robert Krüger für die Organisation meines Forschungsaufenthalts in Boston danken, was mir eine bereichernde Erfahrung ermöglichte. Ich danke auch Peter Sorger und Hanspeter Pfister für ihre freundliche Einladung, sich ihren Forschungsgruppen an der Harvard University anzuschließen. Als Teil ihrer Teams hatte ich die Gelegenheit, mit außergewöhnlichen Forscher*innen zu interagieren.

Ich möchte meinen wunderbaren Freunden danken, die mir immer dabei helfen, den Kopf freizubekommen und das Leben zu genießen. Ich möchte Jakob, Flo und Nec dafür danken, dass sie mich immer mit offenen Armen empfangen und mich daran erinnern, mich zu entspannen, wenn ich es brauche. Vielen Dank, Ferdi und Finnegan, für all die Abenteuer, die wir gemeinsam erlebt haben und jene, die noch kommen werden.

Vor allem möchte ich meiner Familie, meiner Partnerin Alice, meinen Eltern Rochus und Patricia und meinem Bruder Lorenz danken. Danke, dass ihr immer für mich da seid, ihr mir die Möglichkeit gebt, meine eigenen Ziele zu verfolgen und mich die Dinge aus einem anderen Blickwinkel sehen lasst. Ich bin wirklich unsagbar glücklich, euch alle zu haben.

Diese Arbeit wurde finanziell durch den NCI-Award R50CA252138 und den NSF-Award IIS-1901030 unterstützt. Ich möchte Clarence Yapp danken, der die 3D-Immunfluoreszenzdaten produziert hat, die ich zur Auswertung in meiner Arbeit verwendet habe.

Acknowledgements

First and foremost, I want to thank my supervisor Johanna Beyer for her consistent support during my thesis. Your guidance and passion for research have been an invaluable inspiration to me. I could always turn to you for assistance whenever I faced challenges. Moreover, I want to thank you for being a friend, for showing me the coolest places in Boston, and for a really fun summer.

I also want to thank Markus Hadwiger for his outstanding collaboration, mentorship, and support. Your kindness and trust in my abilities really gave me strength and played a crucial role in the progress of my research.

I wish to thank Meister Eduard Gröller for his mentorship and inspiring guidance. Thank you for supporting me without hesitation, for giving me the opportunity to join your research unit, and for giving me invaluable insights into the wonderful world of science.

I would like to thank Robert Krüger for arranging my research stay in Boston. His efforts made it possible for me to have a valuable and enriching experience. I am also grateful to Peter Sorger and Hanspeter Pfister for their kind invitation to join their research groups at Harvard University. Being part of their teams provided me with the opportunity to interact with exceptional researchers, which was truly inspiring.

I want to thank my wonderful friends, who are always helping me get my mind off of things, take a break, and enjoy life. I want to give a big shout-out to Jakob, Flo, and Nec for always welcoming me with open arms and reminding me to chill out when I need it. Thank you, Ferdi and Finnegan, for all adventures we've been through and those yet to come.

Most of all I want to thank my family, my partner Alice, my parents Rochus and Patricia, and my brother Lorenz. Thank you for always being there for me, for allowing me to pursue my own goals, and for letting me see things from another perspective. I'm really fortunate to have you all.

This work was financially supported by the NCI award R50CA252138 and the NSF award IIS-1901030. I want to thank Clarence Yapp who recorded and registered the 3D immunofluorescence data I used for performance evaluation.

Kurzfassung

Jüngste Fortschritte in Bildgebungsverfahren erlauben es umfangreiche volumetrische Datensätze mit einer großen Anzahl von Kanälen zu produzieren. Um solche Volumen interaktiv darzustellen, werden Out-of-Core Direct Volume Rendering (DVR) Methoden benötigt. Aus diesem Grund werden solche Datensätze zu einer Hierarchie mit mehreren Auflösungen herunterskaliert und diese jeweils in kleinere Bricks unterteilt, um nur jene Teile des Volumens, die zum dargestellten Bild beitragen, auf die GPU zu übertragen. Darüber hinaus erfordert das Rendering mehrerer Kanäle aufgrund des hohen Rechenaufwands für DVR sorgfältige Optimierung, da dieser mit der Anzahl der darszustellenden Kanäle steigt. Eine häufige Optimierung bei DVR ist Empty-Space Skipping, bei dem durchsichtige Bereiche im Volumen beim Rendern übersprungen werden.

Frühere Out-of-Core-DVR-Methoden sind nicht für Volumen mit mehreren Kanälen konzipiert und für diese daher nur bedingt geeignet. Oktree-basierte Methoden erfordern für jeden Abtastpunkt und jeden Kanal das Traversieren des Baumes. Darüber hinaus ist in früheren Ansätzen die räumliche Unterteilung des Oktrees an die verfügbaren Auflösungen und die Bricking-Granularität im Datensatz geknüpft. Dies führt zu einer suboptimalen Cache-Nutzung und macht Empty-Space Skipping kostspielig. Page-Table Hierarchien hingegen ermöglichen den direkten Zugriff auf jeden Brick aus jeder Auflösung, ohne eine Baumstruktur zu traversieren. Die räumliche Granularität für Empty-Space Skipping ist hier jedoch ebenfalls an die Bricking-Granularität im Datensatz geknüpft.

Wir stellen einen hybriden Volume-Rendering-Ansatz vor, der auf einem neuartigen Residenz-Oktree basiert. Dieser kombiniert die Vorteile von Page-Table Hierarchien mit jenen von klassischen Oktrees. Unser Ansatz entkoppelt die räumliche Unterteilung, die durch die Oktree-Struktur vorgegeben wird, von den Auflösungsstufen und der Bricking-Granularität im Datensatz. Anstatt dass jeder Knoten exakt einem Brick zugeordnet ist, wird jeder Residenz-Oktree-Knoten in jeder Auflösung mehreren Bricks zugeordnet. Dadurch ist es möglich, Auflösungen effizient und adaptiv auszuwählen und zu mischen, Abtastraten anzupassen und Cache-Fehlzugriffe zu kompensieren. Gleichzeitig erlauben Residenz-Oktrees flexibles Empty-Space Skipping, unabhängig der für das Caching verwendeten Bricking-Granularität. Wir haben unseren Ansatz mit WebGPU und WebAssembly als web-basierten, clientseitigen Renderer implementiert, um wissenschaftliche Zusammenarbeit zu erleichtern. Wir zeigen, dass unsere Methode Datensätze mit vielen Kanälen effizienter darstellt und GPU-Speicher besser nutzt als bisherige Methoden.

Abstract

Recent advances in imaging modalities produce large-scale volumetric data sets with a large number of channels. Interactive visualization of such data sets requires out-of-core direct volume rendering (DVR) methods such as octrees or page-table hierarchies. For this reason, data sets are both down-sampled into a multi-resolution hierarchy and divided into smaller bricks, in order to stream only those parts of the volume contributing to the rendered image to the GPU. Furthermore, rendering multiple channels requires careful optimization because the high computational cost of DVR grows with the number of channels to render. A common optimization in DVR is empty-space skipping where fully translucent regions in the volume are not sampled to reduce the number of loop iterations and texture look-ups during rendering.

Previous out-of-core DVR methods are designed for single-channel volumes and are only suitable for multi-channel volumes to a limited extent. In octree-based methods, accessing cached volume data requires traversing the tree for each sample and channel. Furthermore, in previous approaches, the spatial subdivision of the octree is intrinsically coupled to the down-sampling ratio and bricking granularity in the data set. This leads to suboptimal cache utilization and makes fine-grained empty-space skipping costly. Page-table hierarchies, on the other hand, allow access to any cached brick from any resolution without traversing a tree structure. However, their support for empty-space skipping is also tied to the bricking granularity in the data set and is thus limited.

We present a hybrid multi-volume rendering approach based on a novel *Residency Octree* that combines the advantages of out-of-core volume rendering using page tables with those of standard octrees. We enable flexible mixed-resolution out-of-core multi-volume rendering by decoupling the cache residency of multi-resolution data from a resolution-independent spatial subdivision determined by the tree. Instead of one-to-one node-to-brick correspondences, each residency octree node is mapped to a set of bricks in each resolution level. This makes it possible to efficiently and adaptively choose and mix resolutions, adapt sampling rates, and compensate for cache misses. At the same time, residency octrees support fine-grained empty-space skipping, independent of the data subdivision used for caching. Finally, to facilitate collaboration and outreach, and to eliminate local data storage, our implementation is a web-based, pure client-side renderer using WebGPU and WebAssembly. Our method is faster than prior approaches and efficient for many data channels with a flexible and adaptive choice of data resolution.

Contents

CHAPTER 1

Introduction

1.1 Motivation & Problem Statement

Recent advances in imaging modalities produce large-scale volumetric data sets with a large number of channels. Interactive visualization of such data sets requires out-of-core direct volume rendering (DVR) methods like octrees or page-table hierarchies. An example of large-scale data sets is Immunofluorescence (IF) imaging data. IF technologies (e.g., CyCIF [LIW+18]) are used in the field of digital histopathology to image biological tissue. The resulting multiplexed images contain information for millions of cells in up to 60 channels, where each channel represents the cells' response to one or more marker antibodies [KBJ+20, RCH+22], making proteins visible and thereby revealing the cells' types and functions.

File sizes of IF data sets range from gigabytes to terabytes, and continue to grow as sample sizes, the number of recorded channels, and imaging resolutions increase [LWC+23]. Since such file sizes often exceed the available memory, out-of-core techniques like bricking, multi-resolution hierarchies, and ray-guided rendering approaches, as discussed by Beyer et al. [BHP15], are necessary to visualize IF data sets.

Since it is common for multiple people to collaborate on these data sets using a heterogeneous set of tools and applications [KBJ+20, JKW+22, RCH+22], it is desirable to provide the data via a web server and use web-based visualization tools instead of relying on native applications. Due to limitations such as the lack of general-purpose computing on the GPU (GPGPU) in WebGL 2.0, previous web-based volume rendering research has often focused on minimizing the effects of network latency [YSG15, MKRE18, ACA+19] as well as on optimizing the rendering performance itself by offloading some or all rendering work to a dedicated server [WRT15, QCZ+17, RHH17]. However, with the emergence of WebAssembly [Ros19] and WebGPU [MNJ23], web-based scientific visualization applications are now comparable to native applications, both in terms of performance and

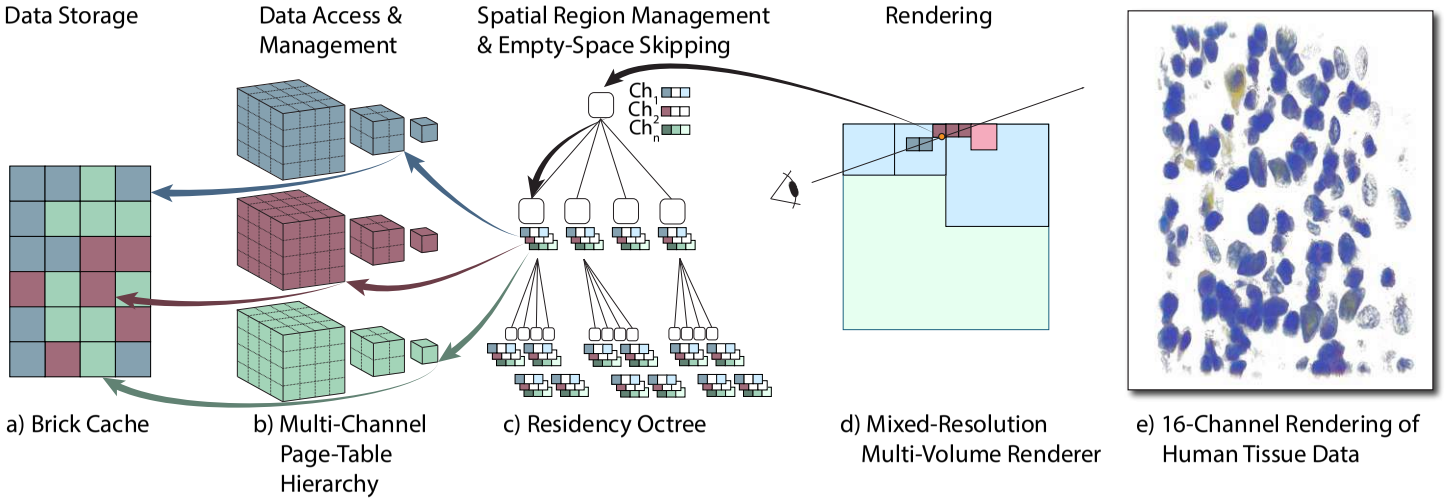


Figure 1.1: **Overview of our method.** Volume bricks of different resolution levels and channels are streamed into a brick cache (a), and referenced via a multi-channel page-table hierarchy (b). The *residency octree* (c) keeps track of the correspondence between spatial regions and the cache residency of bricks of different resolutions, enabling mixed-resolution, multi-channel rendering (d) with efficient, adaptive substitution of missing higher resolutions by available lower resolutions. Traversal happens for spatial regions corresponding to octree nodes instead of memory pages and is also independent of the number of channels. (e) 16-channel rendering of melanoma.

development effort, as, e.g., shown by Usher and Pascucci [UP20]. This now makes it feasible to design out-of-core volume rendering algorithms for web-based contexts that are similar to those developed for native applications.

Apart from the memory pressure introduced by large file sizes, another problem that arises when visualizing highly multiplexed data sets is the large number of different channels. Even though in practice only a subset of $m \leq n$ channels (e.g., $m = 4$) out of all n available channels is visualized at a time, rendering more than one channel using DVR requires careful optimization due to the performance impact of accessing each sample position in multiple volumes. A common optimization in DVR is empty-space skipping, where empty (i.e., fully translucent according to the current transfer-function) regions in the volume are not sampled in order to reduce the number of loop iterations and texture look-ups during rendering [ZSL18, ZHL19, ZML19, ZSL21, FW20, LCDP13, HAAB+18, DK19, DK20, XZLK19, BPH14]. However, most existing web-based volume renderers are neither designed for large-scale multi-channel data nor do they use optimizations such as fine-grained empty-space skipping techniques [MGP+22, Goo22].

The subset of visualized channels is usually user-defined and may change at run-time. For this reason, acceleration structures used to optimize rendering performance need to be flexible enough to allow for channel selection switches. Furthermore, channels are not necessarily equally important, for example, by having different frequency content, or they are simply less interesting to the user in the current context. This makes it desirable to render more important channels in higher resolution while rendering less important channels in lower resolution in order to reduce the memory required for storing the currently visible volume data on the GPU. Existing techniques for multi-volume data for native environments are not designed for channel switches and do not support rendering different channels at different resolutions [BPH14, DLJL22].

Ray-guided DVR methods have been shown to work best for large-scale volumetric data [BHP15]. They can be divided into two families: page-table-based and octree-based approaches. The former allows direct access to any cached brick from any resolution level. In octree-based approaches, each node represents exactly one brick in the data set. Because of the hierarchical tree structure, these approaches enforce the traversal algorithm used to access cached volume data during rendering as well as the order in which bricks of different resolutions are streamed in, i.e., from the root node to leaf nodes. This makes page-table-based approaches more flexible than octree-based ones, but they do not offer a clear and efficient strategy for substituting missing high-resolution data with lower-resolution data guaranteed to be resident in the cache [BHP15]. Both families allow for empty-space skipping by either skipping over empty pages or empty octree nodes, respectively.

1.2 Aim of the Work

We propose a novel residency octree, a hybrid data structure for out-of-core DVR of multi-volume data that combines the advantages of page tables with those of octrees.

It decouples the cache residency of multi-resolution data from a resolution-independent spatial subdivision determined by the tree. Instead of one-to-one node-to-brick correspondences, each residency octree node only represents a spatial region in the volume that is mapped to a set of bricks in each resolution level in the bricked volume hierarchy. This makes it possible to efficiently and adaptively choose and mix resolutions, adapt sampling rates, and compensate for cache misses. At the same time, this decoupling allows residency octrees to support fine-grained empty-space skipping, independent of the data subdivision used for caching. For this purpose, each residency-octree node stores transfer-function independent metadata, e.g., minimum and maximum scalar values in the spatial region represented by the node, alongside the information about resolution levels in which the node's corresponding bricks are currently resident in the cache. Internally, our data structure is backed by a multi-resolution multi-channel page-table hierarchy and a brick cache. By keeping information about multiple resolutions and channels in each node, the residency octree works well for multi-channel empty-space skipping and allows mixing different resolutions not only for single-channel but also for multi-channel data. Our data structure has been fully implemented on the GPU and facilitates efficient run-time changes to the selection of visible channels. With WebGPU [MNJ23], the method enables pure client-side out-of-core multi-volume rendering on the web. Figure 1.1 gives an overview of our system architecture.

1.3 Contributions

The main contributions made in this work are the following:

1. A novel hybrid data structure for out-of-core volume rendering of multi-volume data sets that decouples the resolution levels in a bricked volume hierarchy from the spatial subdivision determined by the residency octree. This decoupling makes it possible to efficiently and adaptively choose and mix resolutions, both between samples of a single channel and samples taken from multiple channels, adapt sampling rates accordingly, compensate for cache misses, and skip empty space.
2. A mixed-resolution multi-volume rendering algorithm to visualize multiple channels at different resolutions and dynamically take into account differences in channel importance.

We evaluate both the introduced data structure and the mixed-resolution multi-volume rendering algorithm by comparing our approach to previous methods in terms of GPU memory usage as well as run-time performance.

1.4 Structure of the Thesis

In Chapter 2, we examine previous and related work to put our work into context. We include related work on large-scale volume rendering (Section 2.1), multi-volume

rendering (Section 2.2), empty-space skipping (Section 2.3), and web-based volume rendering approaches (Section 2.4). Chapter 3 introduces important concepts and terminology that are used throughout this work.

In Chapter 4, we give a high-level overview of our system's architecture and explain how its individual parts work together. A multi-channel page-table hierarchy and brick cache discussed in Chapter 5 form the basis of the approach. The residency octree, the data structure introduced by this work, is discussed on a theoretical level in Chapter 6. Chapter 7 then shows how the residency octree is used in the mixed-resolution multi-volume rendering algorithm presented in this thesis to visualize multiple channels at different resolutions.

In Chapter 8, we give a detailed discussion of how we implemented the data structures and algorithms presented in Chapters 4 to 7 using WebGPU [MNJ23] and WebAssembly [Ros19].

We evaluate the run-time performance of our renderer and the GPU memory usage of the residency octree in Chapter 9. We compare our method with two reference implementations based on previous work. The results of this evaluation are discussed in Chapter 10.

Chapter 11 concludes this work. We summarize the concepts introduced in this thesis, as well as the results of the evaluation. Finally, we give an outline of future work.

CHAPTER 2

Related Work

The following sections give an overview of work related to this thesis. Research on large-scale volume rendering is discussed in Section 2.1. Section 2.2 gives an overview of previous multi-volume rendering approaches. Existing empty-space skipping methods are discussed in Section 2.3, and previous web-based techniques and applications, as well as file formats facilitating web-based volume rendering, are covered in Sections 2.4 and 2.5 respectively.

2.1 Large-scale Volume Rendering

Childs defines a data set as large if it is "too large to be processed ... (1) in its entirety, (2) all at once, and (3) exceeds the available memory" [BCH12, p. 9]. Following this definition, Beyer et al. [BHP15] define the scalability of GPU-based large-scale visualization approaches in terms of the minimal subset of the data required to render an image at a desired resolution - the *working set*. A visualization approach is considered scalable if the working-set size depends only on the output resolution and the data visible on the screen but is independent of the size of the whole data set.

Several scalable volume rendering techniques have been developed for desktop environments [BHP15]. For these approaches, the data set is (a) present in a multi-resolution hierarchy and (b) split into smaller bricks where all bricks across all resolutions have the same size in voxels. A multi-resolution hierarchy can either have a fixed or arbitrary spatial subdivision scheme and fixed or arbitrary down-sampling ratios between resolution levels. An example of a data structure using a fixed spatial subdivision scheme and fixed down-sampling ratios is the octree. Using arbitrary down-sampling ratios can be beneficial for anisotropic volumes. Figure 2.1 illustrates these concepts.

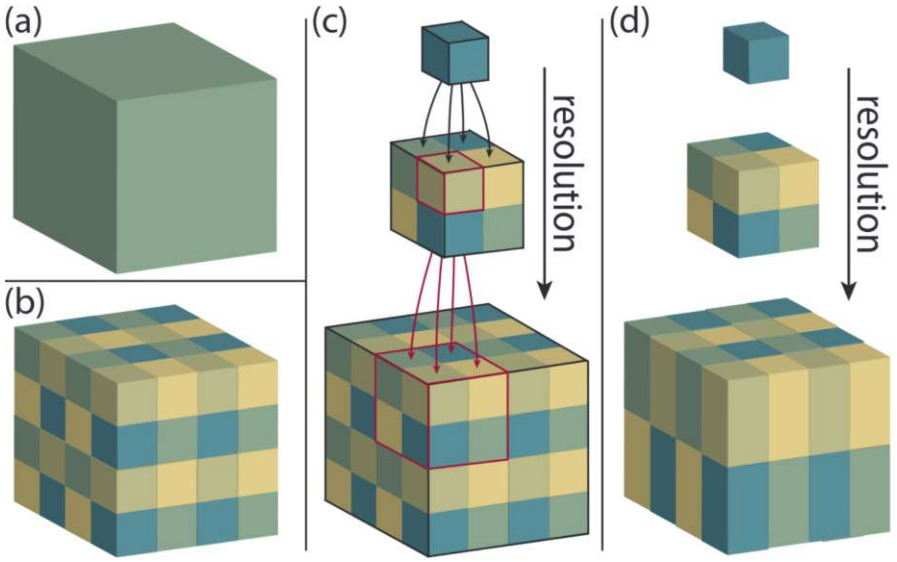


Figure 2.1: **Bricking & Multi-resolution hierarchies.** In out-of-core volume rendering the original volume (a) is divided into a bricked volume (b). Furthermore, the data is down-sampled into a multi-resolution hierarchy using fixed (c) or arbitrary (c) down-sampling ratios. Octrees (d) use a fixed down-sampling ratio. Source: [BHP15]

2.1.1 Culling

Early approaches to large-scale volume rendering were based on culling bricks that do not contribute to the output image, and thus determining the working set, before rendering [BHP15]. Bricks can either be culled if their bounding box is not within the camera's view frustum [AM00], if they are fully transparent with respect to the transfer function used [HSS+05, BHWB07], or due to occlusion [LMK03, GHSK03, GSHK04, MM10]. For performance reasons, these early approaches produce very conservative estimates of the working set, leading to sub-optimal memory utilization [BHP15].

2.1.2 Ray-guided volume rendering

In ray-guided volume rendering [CNLE09, HBJP12, FSK13, BPH14, SCRL20], the working set is determined during the ray casting process itself. In each frame, a list of brick requests is compiled to stream in missing volume data on demand. Ray-guided methods use a brick cache, e.g., a 3D texture, to store individual bricks of different resolutions in GPU memory in an unordered manner. Furthermore, an additional data structure is used to store for each brick in the volume if it is resident in the cache or not. During ray casting, this data structure is used to translate a sample location specified in a reference space that comprises the entire volume, e.g., normalized coordinates in a $[0, 1]^3$ unit cube, to texture coordinates within the corresponding brick in the cache. Beyer et al. [BHP15] refer to this process as *address translation*. Previous ray-guided rendering approaches use either octrees [CNLE09, BPH14, DLJL22] or page-table hier-

archies [HBJP12, FSK13, SCRL20] for address translation. In contrast to estimating the working set in advance, ray-guided approaches minimize the working-set size by only streaming in bricks that are accessed during rendering. However, depending on the data structure used for address translation, the working set may still contain bricks that do not contribute to the rendered image as shown in Section 9.6. Data structures for address translations typically also have rudimentary empty-space skipping capabilities, e.g., by marking fully translucent bricks as empty and thus not requiring them to be stored in GPU memory [CNLE09, HBJP12, FSK13, BPH14, SCRL20]. However, due to large brick sizes in voxels, e.g., 32^3 or larger, the spatial subdivision of a bricked volume is often not fine-grained enough for efficient empty-space skipping [FZZ+22].

2.1.3 Data structures for address translation in ray-guided rendering

The state-of-the-art methods for address translation in ray-guided rendering are either based on octrees [CNLE09, BPH14] or page-table hierarchies [HBJP12, FSK13, SCRL20].

Octrees

In octree-based ray-guided rendering, the down-sampling ratio between resolution levels in the multi-resolution hierarchy is intrinsically coupled to the spatial subdivision determined by the tree structure such that each node corresponds to exactly one brick in the data set [CNLE09, BPH14]. Crassin et al. [CNLE09] were the first to use an octree structure for address translation in the GigaVoxels system illustrated in Figure 2.2. In their approach, octree nodes are stored within a node pool. Each entry in the node pool is a group of eight nodes, which all have the same parent node. Each node in the octree stores a pointer to its corresponding brick-cache entry and a pointer to the node-pool entry storing its children. As an optimization, each node additionally stores metadata on the region represented by the node, e.g., if it only contains zero values, that is used to skip over translucent nodes during rendering. The node-pool acts as a cache for octree nodes, where only those nodes are kept in memory that have corresponding bricks stored in the brick cache. Octree nodes that are resident in the node pool are considered *active*. During ray casting, the octree hierarchy is traversed for each sample. In order to reach a high-resolution brick, the whole subtree, i.e., from the root node to the node corresponding to the high-resolution brick, has to be kept both in the node pool and the brick cache. While this makes it straightforward to substitute missing high-resolution data with low-resolution data, this also leads to potentially unused low-resolution bricks being stored in the cache and thus sub-optimal GPU memory utilization. Note that this is due to the tight coupling of the multi-resolution hierarchy in the data set and the spatial subdivision of the octree. Crassin et al. [CNLE09] use multiple render targets to record up to one missing brick per ray and report these brick requests to the CPU. Brix et al. [BPH14] extend the structure presented by Crassin et al. [CNLE09] to multi-channel data as discussed in Section 2.2.

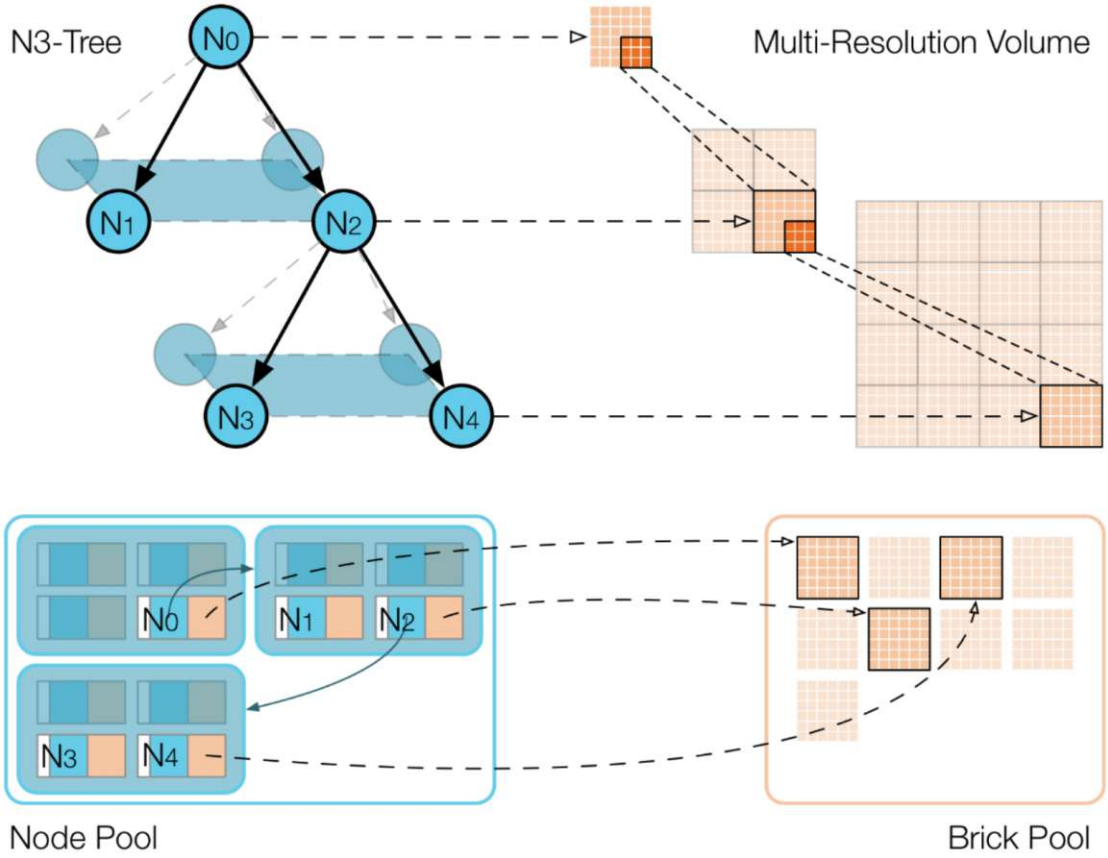


Figure 2.2: **The GigaVoxels system.** Each node in the octree (here: N3-Tree) keeps track of the cache residency of its corresponding brick in the multi-resolution volume. Bricks are stored in a brick pool and accessed via their corresponding nodes during rendering. Active nodes, i.e., nodes whose bricks are resident in the cache, are stored in a node pool. Source: [CNLE09]

Page-Table Hierarchies

Hadwiger et al. [HBJP12] propose a memory virtualization technique for volumetric data based on page-table hierarchies instead of a tree structure. Each page table in the page-table hierarchy represents a single resolution level in the volume hierarchy. A page table comprises multiple pages that allow referencing all bricks corresponding to the page table's resolution level. Each page stores if its corresponding brick is resident in the brick cache or not and if so, a pointer to its location in the brick cache. As an optimization, a page table may also mark bricks that are known to not contain any useful data with an `EMPTY` flag to eliminate the need to actually load or store them in the brick cache. As in the octree-based approach by Crassin et al. [CNLE09], this allows a renderer to skip over translucent bricks during ray casting. Because there is no explicit hierarchy between pages of different page tables involved, their method supports both arbitrary

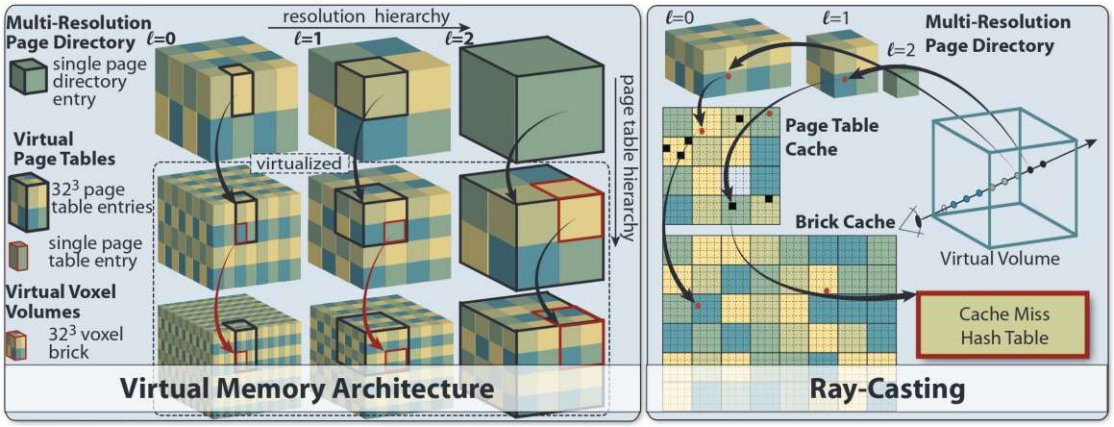


Figure 2.3: **Multi-resolution page-table hierarchy.** Left: The volume virtualization makes use of two orthogonal hierarchies: the resolution hierarchy in the data set, and the page-table hierarchy. Right: During ray casting, the page-table hierarchy (here: Multi-Resolution Page Directory) is used for address translation. Missing bricks are reported in a hash table. Source: [HBJP12]

down-sampling ratios for the data set and directly accessing volume data of a desired resolution without traversing a tree structure. However, their technique does not give a clear strategy for substituting missing bricks with other resolutions resident in GPU memory. To report missing bricks, Hadwiger et al. [HBJP12] use hash tables shared by all rays in an $N \times N$ tile in screen space. These hash tables are read back to the CPU once per frame. Since the page-table hierarchy is itself a hierarchy of volumes, the memory virtualization technique can be applied to produce a multi-resolution page-table hierarchy as illustrated in Figure 2.3. This recursive approach makes it possible to represent volume sizes in the exascale on the GPU [HBJP12].

Fogal et al. [FSK13] propose a similar approach but use a global hash table for reporting cache misses back to the CPU instead of one hash table per image tile. Sarton et al. [SCRL20] generalize the concept of page-table hierarchies for volumetric data sets for non-rendering purposes. They use CUDA shared memory to process cache misses on the GPU itself to minimize memory transfers between the CPU and the GPU. Whenever a brick-cache entry is used, the current timestamp is recorded in a global brick-usage buffer that comprises one timestamp per cache entry. For example, in a real-time volume rendering application, these timestamps could be the current frame's number. These timestamps are used for managing the brick cache in a Least Recently Used (LRU) manner and keeping frequently used bricks in the cache. For this reason, indices into the brick cache sorted by their most recent usage times are stored in an LRU buffer. To update the LRU buffer, a mask of brick-cache entries used at the current timestamp is generated from the timestamps in the usage buffer. This mask is then used to rearrange the last timestamp's LRU buffer such that the most recently used entries are moved to the front of the LRU buffer. Similarly, whenever a brick is missing, the current timestamp

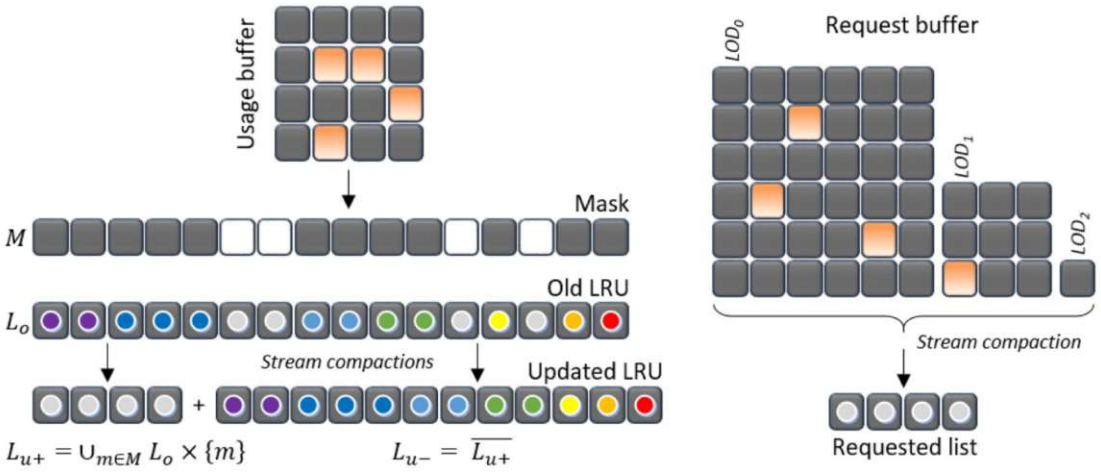


Figure 2.4: **Parallel cache management.** Left: Brick usages are marked in a global usage buffer. The usage buffer is processed in parallel to update the LRU buffer (here: Old LRU and Updated LRU). Right: A list of brick requests is compiled from a global request buffer with one entry per brick in the data set. Source: [SCRL20]

is recorded in a global request buffer that contains one timestamp per brick in the data set. A single stream compaction is used to compile a list of all bricks that have been reported missing at a given timestamp. Their parallel cache management approach is illustrated in Figure 2.4.

Residency Octree

Our approach combines the advantages of both octrees and page-table hierarchies by decoupling the cache residency of multi-resolution data from the spatial subdivision determined by the residency octree. Figure 2.5 illustrates how this novel approach differs from previous octree-based ones. Instead of having one-to-one correspondences between bricks and nodes, our approach decouples resolution levels in the data set and the spatial subdivisions determined by the octree structure such that each node maps to one or more bricks in each resolution level.

2.2 Multi-Volume Rendering

Multi-volume rendering refers to rendering multiple co-registered volumes at once. This is done by evaluating each sample along a viewing ray for all channels currently visible. Schubert and Scholl [SS11] give an overview of how multiple channels can be combined during rendering. They differentiate between (1) classification-level intermixing, (2) accumulation-level intermixing, and (3) image-level intermixing. In classification-level intermixing, at each ray sample, the sampled values of two channels are linearly interpolated using a weighting factor ω, e.g., $\omega = 0.5$, before evaluating an illumination model using

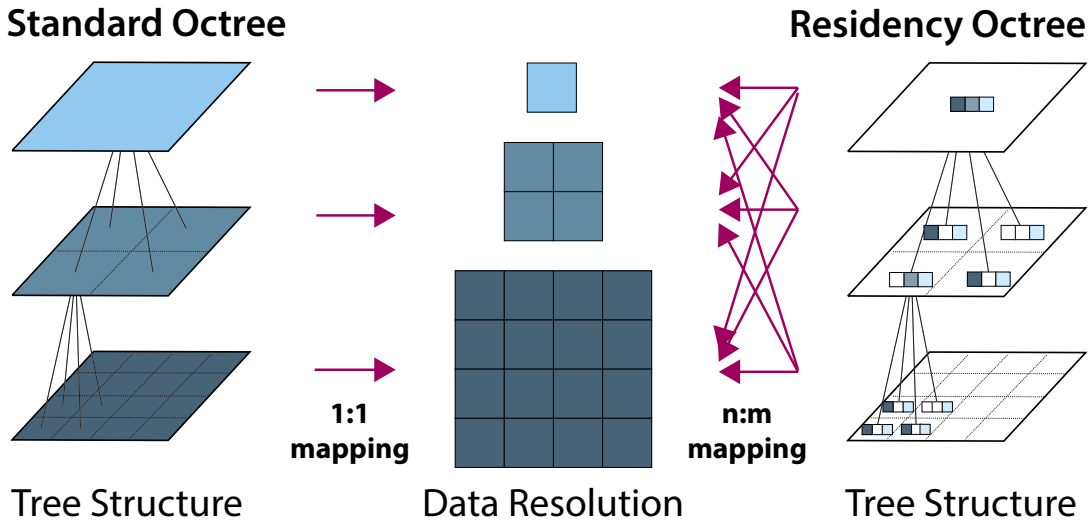


Figure 2.5: **Previous octree-based out-of-core approaches** (left) employ a one-to-one mapping between bricks and octree nodes. In contrast, the residency-octree nodes in our approach (right) represent geometric spatial regions, with each node mapping to multiple bricks and vice versa.

the interpolation result. Accumulation-level intermixing refers to separately evaluating the illumination model for each channel at each ray sample and then combining the results. This is more flexible than the other approaches but computationally also more expensive. For image-level intermixing, a complete image is rendered for each channel and the results are mixed using alpha blending. A problem that arises with image-level intermixing is that occlusions between different channels can not be handled correctly due to missing depth information in the accumulated intermediate results.

Brix et al. [BPH14] present an out-of-core rendering approach for multi-channel volume data sets that is based on Crassin et al.'s [CNLE09] octree-based memory virtualization scheme. Instead of keeping track of a single channel only, octree nodes, as well as bricks, store information on up to n, e.g., $n = 4$, channels. The number of channels is implementation-defined and can not be changed at run-time. In practice, because each brick-cache entry stores n channels and texture formats available on current GPU hardware are limited to four components, the number of channels n is limited to 4. To combine multiple channels, Brix et al. [BPH14] use accumulation-level intermixing. As an optimization, nodes corresponding to homogeneous bricks, i.e., all voxels contain the same value (up to a user-defined threshold), store the average value in the corresponding brick instead of a pointer to the brick's location in the brick cache. For multi-channel data, a node can only be marked homogeneous if its corresponding bricks in all channels are homogeneous. Their approach is implemented in Voreen [DLJL22].

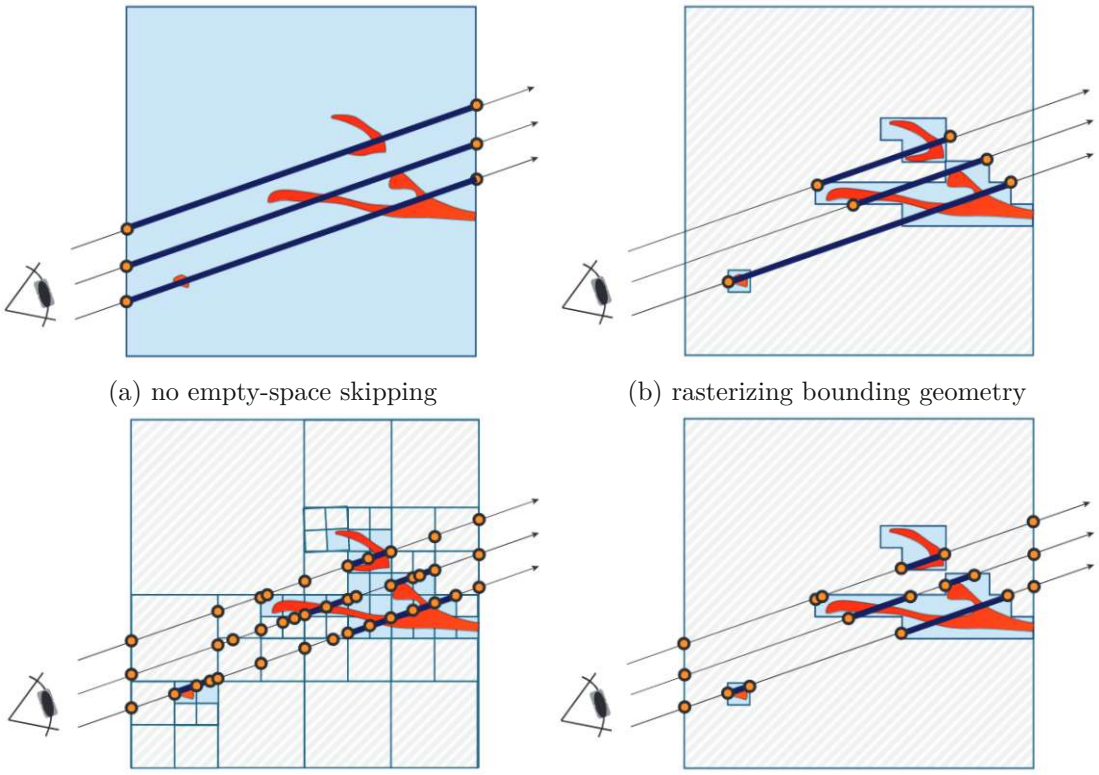
Viv [MGP+22] is a web-based volume renderer that supports multi-channel data by storing

each channel in a separate texture. The number of channels that can be represented simultaneously on the GPU is therefore limited only by the number of available texture bind points allowed by the hardware. WebGL 2.0 guarantees a minimum of eight texture bind points. Channels may be switched at run-time, which simply changes the texture bindings in the shader. However, Viv does not employ any out-of-core techniques to support large-scale data and is thus limited by the GPU memory available. Multiple channels are combined during rendering using accumulation-level intermixing. Neuroglancer [Goo22] uses the same technique for representing multiple channels on the GPU. In contrast to Viv, Neuroglancer uses bricking for volumetric data. They choose a global resolution level for each frame based on current viewing parameters and thus produce suboptimal working sets. Furthermore, rendering multiple channels requires users to adapt the fragment shader used for ray casting themselves.

Our mixed-resolution multi-volume rendering algorithm uses accumulation-level intermixing to combine different channels. The residency octree is backed by a channel-agnostic brick cache where each cache entry only contains data belonging to a single channel. The number of channels is thus not limited by the texture format used by the brick cache implementation. Furthermore, residency octrees support changes to the selection of visible channels at run-time by replacing only the parts of their nodes related to the affected channels.

2.3 Empty-Space Skipping

One of the major problems with DVR is its high computational cost due to evaluating hundreds to thousands of samples per ray in each frame. Empty-space skipping methods accelerate this process by determining regions of empty space that do not require extensive sampling, effectively reducing the number of samples that have to be evaluated. For this purpose, volumes are subdivided into smaller parts, often involving a tree-like hierarchy, like octrees [CNLE09, LCDP13, BPH14, FZZ+22], kd-trees [ZSL18, ZML19, ZSL21], or linear bounding volume hierarchies (LBVH) [ZHL19, FW20] that are traversed during rendering. Other approaches rasterize bounding geometry to speed up volume ray casting [LCDP13, HAAB+18, ZML19, WBD+21]. Figure 2.6 shows a comparison of different empty-space skipping methods. They differ in the number of look-ups into the data structure used for empty-space skipping and the number of samples along each viewing ray. There is often a trade-off between the two. Which of the two is more important to optimize depends on the respective use case as well as on different aspects of the volume data, such as the distribution of empty space and the size of the volume. If no empty-space skipping is used (Figure 2.6a), the whole volume has to be sampled. Standard rasterization of bounding geometry (Figure 2.6b) only skips empty space before the first and behind the last non-empty sample. Octrees (Figure 2.6c) allow a near-optimal distribution of samples but require a lot of look-ups into the tree structure. Hadwiger et al. [HAAB+18] propose *SparseLeap* (Figure 2.6d), a method that optimizes both the number of samples to evaluate and the number of look-ups into the empty-space skipping data structure.



(a) no empty-space skipping (b) rasterizing bounding geometry

(c) octree (d) *SparseLeap* [HAAB+18]

Figure 2.6: **Comparison of empty-space skipping methods.** Red structures depict non-empty voxels. Light blue backgrounds visualize the bounding geometry used by the respective method. Regions, where the volume is sampled, are denoted by bold ray segments, and dots show where the empty-space skipping data structure has to be queried by each method. Source: [HAAB+18]

2.3.1 Parallel Tree-Construction for Empty-Space Skipping

In recent years, several parallel tree-construction algorithms have been proposed to support run-time re-construction of acceleration data structures for empty-space skipping, e.g., as a result of changes to the transfer function used [ZSL18, ZHL19, ZML19, FW20, DK19, DK20, ZSL21]. Zellmann et al. [ZHL19] propose an adaptation of a linear bounding volume hierarchy (LBVH) construction algorithm for triangle geometry to sparse volumetric data by using the boundaries of non-empty bricks as bounding geometry. They show that their algorithm allows constructing an acceleration structure on the GPU at interactive frame rates for 1024^3 volumes and that their LBVH structure allows for two to three times higher frame rates than using no empty-space skipping during ray marching. While Zellmann et al. [ZHL19] divide the volume into a uniform grid with a fixed brick size for their LBVH construction algorithm, Fernandes and Walter [FW20] group voxels into buckets instead. For sparse volumes, Zellmann et al. [ZSL18] present a parallel

construction scheme for reconstructing a kd-tree structure based on summed-volume tables. While their work is designed for multi-core CPU architectures, Zellmann et al. [ZSL21] adapt their previous approach to a GPU-based construction. Zellmann et al. [ZML19] present a hybrid data structure combining a kd-tree structure at the root level and a uniform grid at the leaf-node level that is used for empty-space skipping. During rendering, the kd-tree is traversed on the CPU to generate a sorted list of non-empty leaf nodes that are then iterated over on the GPU.

Other approaches include the use of Chebychev distance maps [DK19, DK20], or subdividing the volume into non-uniform grids [XZLK19]. Deakin and Knackstedt [DK19] use Chebychev distance maps for empty-space skipping, which achieves speed-ups of 5.3 times the baseline frame rate on average. Since the distance maps are transfer-function dependent, they need to be updated on transfer-function changes. Deakin and Knackstedt [DK20] further improve their previous approach by resuming empty-space skipping after a single empty voxel instead of a fixed set length of empty voxels.

Our approach builds on an octree structure that is constructed in parallel on the GPU. Since the residency octree is designed to handle both empty-space skipping, and cache residency, we avoid reconstructing it when viewing parameters, e.g., transfer functions, change, and rely on incrementally constructing the octree.

2.3.2 Rasterizing Bounding Geometry

Instead of traversing a tree structure on the GPU, hardware-accelerated rasterization can be exploited to skip over empty space [LCDP13, HAAB+18, ZML19, WBD+21]. This is done by rasterizing bounding geometries, e.g., bounding boxes of leaf nodes of a tree structure [ZML19], and using the fragment positions of their front and back faces as the start and end points for ray casting. This can lead to fewer look-ups into the empty-space skipping data structure during rendering when compared to octree structures as illustrated in Figure 2.6.

Liu et al. [LCDP13] propose rasterizing proxy geometry representing a view-dependent cut of octree nodes. Hadwiger et al. [HAAB+18] also utilize rasterization hardware to avoid hierarchy traversal on the GPU by only traversing per-pixel lists of non-empty ray segments produced through rasterizing occupancy geometry. The occupancy geometry is generated from a lazily constructed occupancy-histogram tree where each node has one of the occupancy classes, *empty*, if it is completely empty, *non-empty*, if it is non-empty, or *unknown* if such information is not yet known. As illustrated in Figure 2.6d, this approach called SparseLeap optimizes both the number of look-ups into the data structure used for empty-space skipping and the number of samples along each viewing ray. An advantage of SparseLeap over other empty-space skipping techniques is that it is agnostic of the data structure used for managing volume data, e.g., page-table hierarchies, or octrees, and can instead be used in conjunction with them. Wang et al. [WBD+21] use a tile-based rendering scheme combined with an octree structure with a fixed leaf-node size to render

per-tile node lists. They note that while a smaller leaf-node size leads to less overlap of node lists between image tiles, the storage and traversal overhead is increased.

The residency octree shares with SparseLeap the incremental construction of the underlying tree structure. But instead of rasterizing octree nodes or other bounding geometry, the empty-space skipping algorithm traverses the residency octree directly on the GPU.

2.3.3 Transfer-Function Independent Empty-Space Skipping

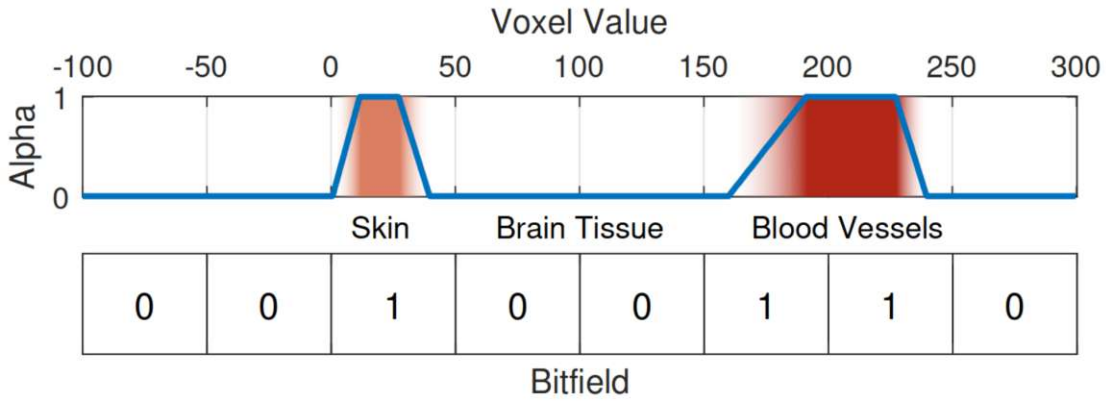


Figure 2.7: **Bitfield representation of a transfer function.** The transfer function (top) is quantized to an 8-bit bitfield (bottom). By representing the value range of an octree node in a similar fashion, testing if a node is translucent under the transfer function comes down to a single bit-wise AND operation. Source: [FZZ+22]

Many acceleration structures store transfer-function dependent visibility information and therefore require reconstruction after transfer-function changes [ZSL18, ZHL19, ZML19, FW20, DK19, DK20, ZSL21]. Faludi et al. [FZZ+22] propose storing a histogram of values contained in the spatial region represented by an octree node as a bitfield instead. Similarly, the visible value range of a transfer function is also represented as a histogram as illustrated in Figure 2.7. To determine if a node is empty, its bitfield is tested against the transfer function's bitfield using a bitwise AND operation. This allows for transfer-function changes at run-time without the need for reconstructing the acceleration structure. Their approach achieved the best results with a leaf-node size of 4^3 and optimized the traversal algorithm by starting at a subdivision level other than the root node which is expected to be non-empty anyway. Similarly, Brix et al. [BPH14] propose an octree structure that stores average values in non-empty homogeneous nodes.

Our approach also uses transfer-function independent culling information in residency-octree nodes for empty-space skipping without the need to reconstruct the tree on transfer-function changes. This has the advantage that cache-residency information also does not have to be redistributed over octree nodes on transfer-function changes.

2.4 Web-Based Volume Rendering

In web-based environments, volume rendering research is mostly focused on solutions to network latency when loading volumetric data [YSG15, YSG15, MKRE18, ACA+19] and improving volume rendering performance on the web, e.g., by offloading some or all of the rendering work to remote rendering servers and only displaying the rendered images in the browser or client application [FCS+10, BHS+11, WRT15, QCZ+17, RHH17, SRL19].

For large-scale volume rendering, several distributed server-side rendering schemes have been proposed [FCS+10, BHS+11, SRL19]. Beyer et al. [BHS+11] propose a distributed shared virtual-memory scheme for distributed server-side rendering of tera-scale volumes where multiple rendering nodes render only a part of the volume each. A similar approach is presented by Sarton et al. [SRL19] who use a single node server with multiple GPUs and CPUs instead. Campoalegre et al. [CNC13] transfer gradient octrees computed on the server side to a rendering client to reduce network traffic and avoid the need to compute gradients on the client. Our method is designed to work without a dedicated rendering backend and relies only on a file server instead. All volume rendering is done in the browser itself.

In contrast to server-side rendering, pure client-side renderers only require a file server as a backend [CSK+11, MF12, DGBNV18, AMBGA19, MGP+22, Goo22]. Manz et al. [MGP+22] present Viv, a WebGL-based library for visualizing biomedical data. While their library focuses on 2D data, it also allows DVR of 3D data. However, it is limited to the available GPU memory and thus does not support large-scale data. Neuroglancer [Goo22] is a WebGL-based tool for visualizing and annotating large-scale volumetric data. It uses a bricked multi-resolution hierarchy approach for its DVR view to scale to large data sets. In the DVR view, all bricks are selected from the same resolution based on global camera parameters before rendering a frame leading to suboptimal working sets. Current web-based volume rendering solutions are limited by WebGL's lack of compute pipelines and general shader storage buffers.

The method presented in this work is designed for client-side rendering. Our implementation (Chapter 8) makes use of the upcoming WebGPU standard, which allows a range of algorithms proposed for native environments to be implemented in web contexts thanks to its GPGPU capabilities [UP20].

2.5 File Formats

Open file formats for bricked multi-resolution hierarchies eliminate the need to run a dedicated back-end to pre-process data and facilitate integrating new software into existing workflows.

Besson et al. [BLL+19] describe the OME-TIFF format for storing both 2D and 3D multi-resolution, multi-channel biomedical imaging data. Multiple resolutions and channels are all stored within a single Tiff file. An application requiring only a subset of all resolutions

and channels, e.g., a single channel, has to load the entire file into memory. This makes OME-TIFF infeasible for web-based out-of-core rendering of 3D data.

Moore et al. [MAB+21] present OME-NGFF, an open file format for chunked biomedical 2D and 3D multi-resolution, multi-channel data sets. It allows fast access to individual chunks of the data set by storing them in separate files. This also makes it more useful for use in a cloud-based context than its predecessor OME-TIFF, since chunks may be distributed across multiple file servers.

Manz et al. [MGP+22] present indexed OME-TIFF, an open file format for biomedical image data sets that improves access times to individual chunks of an underlying OME-TIFF data set by storing byte offsets in a separate JSON file. They note, however, that OME-NGFF still outperforms indexed OME-TIFF in terms of access times for individual chunks of a data set.

Our implementation (Chapter 8) uses OME-Zarr, an implementation of OME-NGFF, for representing bricked multi-resolution volume data, with LZ4-compressed chunks for efficient data transfer over a network.

CHAPTER 3

Basics and Terminology

In this chapter, we establish concepts and terminology, which we are going to use throughout this thesis. We make a clear distinction between the resolution levels of a volume and the spatial, purely geometric subdivision determined by the residency octree, i.e., the geometric boundaries of octree nodes.

3.1 Brick Hierarchy

Each data set is represented as a bricked volume hierarchy consisting of multiple *resolution levels*. A *brick* in this hierarchy is a chunk of voxel data belonging to a specific resolution level. Each brick has a *size in voxels* (e.g., 32^3) that is the same for all resolution levels, and that is therefore independent of its *spatial extent*. The latter is determined by the resolution level the brick belongs to. To access a brick's data on the GPU, it has to be *resident* in a *brick cache*.

3.2 Residency-Octree Hierarchy

In contrast, a residency octree has multiple *subdivision levels*. While the downsampling ratio between resolution levels in the bricked volume hierarchy is flexible (see, e.g., the work by Hadwiger et al. [HBJP12]), the spatial subdivision determined by the octree is not. Instead, at each subdivision level in the octree, each dimension is halved such that each *node* in the tree has eight children. A node in the residency octree thus represents a region in the volume whose spatial extent is determined by the subdivision level it belongs to. Other than in previous work [CNLE09, BPH14], we do not associate a size in voxels with a single octree node, as illustrated in Figure 2.5. In previous octree-based out-of-core DVR approaches, the resolution levels of the bricked volume hierarchy are intrinsically coupled to the subdivision levels of the octree, due to a one-to-one mapping between bricks and octree nodes.

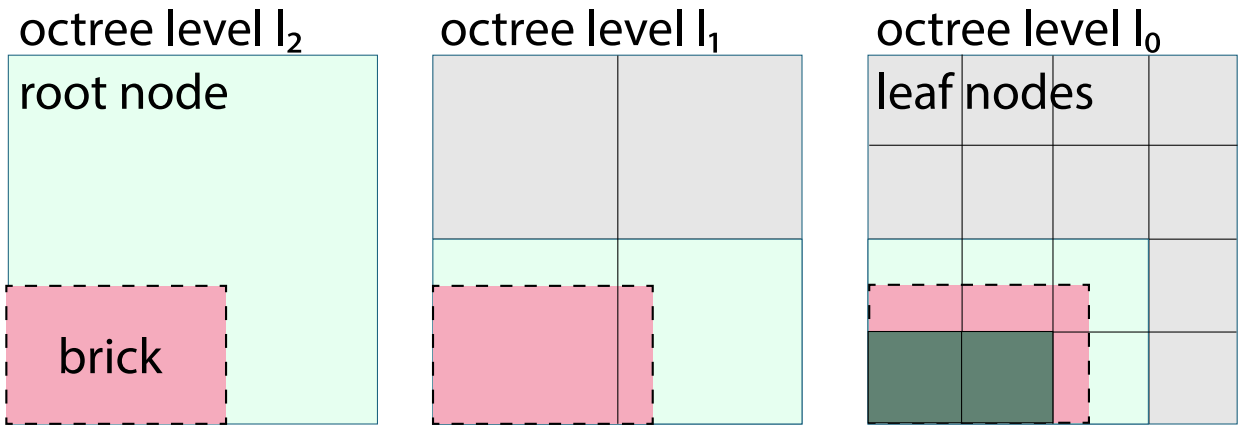


Figure 3.1: **Bricks vs. octree nodes.** A brick from one resolution level (red, its boundary is illustrated by dashed lines) maps to nodes in all subdivision levels (solid lines indicate octree node boundaries). Some nodes (light green) are only partially mapped in the brick's resolution level, while others are fully mapped (dark green) when this brick is resident in the cache.

In our approach, these two concepts are decoupled, such that an octree node corresponds to a *set* of bricks in each resolution level in the bricked volume hierarchy. The number of bricks of a resolution level required to fully cover the spatial extent of an octree node depends on the spatial extent of bricks in that resolution level, which differs from the spatial extent of the octree node. For instance, the spatial extent of a single brick in the lowest resolution level may cover the entire volume space, and thus cover all residency-octree nodes. Vice versa, a brick in a higher resolution level will only cover a part of the residency octree's root node because the latter covers the whole volume.

3.3 Fully vs. Partially Mapped Residency-Octree Nodes

If for a given octree node the cache-resident bricks of a given resolution level fully cover the node's spatial extent, we refer to the node as *fully mapped* in that resolution level. Similarly, if at least one brick of a resolution level whose spatial extent overlaps with a node's spatial extent is resident in the cache, the node is *partially mapped* in that resolution level. This implies that a node that is fully mapped in a resolution level is also partially mapped, but not vice versa. Furthermore, if a node is fully mapped in some resolution level, all of its child nodes are also fully mapped in that resolution level; and if a node is partially mapped in some resolution level, its parent node is too. Naturally, the same brick can both fully map some nodes, while partially mapping others, as Figure 3.1 illustrates, and an octree node may be partially and fully mapped in multiple resolutions at the same time.

3.4 Multi-Channel Data

If a volume has multiple channels, the bricked volume hierarchy is extended to a *bricked multi-volume hierarchy*, where we assume all channels to use the same brick size in voxels. The relationship of octree nodes and bricks in this multi-volume hierarchy is similar to the single-channel case, but instead of having a single set of corresponding bricks per resolution level, a node now has one such set for each channel and resolution level.

CHAPTER 4

System Overview

This chapter provides an outline of the architecture of our system. We briefly discuss the server side providing access to a bricked multi-volume hierarchy in Section 4.1. Section 4.2 then gives an overview of the client-side consuming and rendering volume data. We show how the residency octree (Section 4.2.1) and the ray-guided, mixed-resolution multi-volume renderer (Section 4.2.2) are used together to render large-scale multi-volume data in web browsers. Finally, we discuss our GPU-driven design in Section 4.2.3. The complete system architecture, with the server (Section 4.1) on the left and the client (Section 4.2) on the right, is illustrated in Figure 4.1.

4.1 Server

The bricks in the bricked multi-volume hierarchy are provided by a server. This server has to provide the following data to the client application:

- **Metadata:** The server needs to be able to provide some metadata about the data sets, such as the number of resolution levels, the number of channels, the brick size, etc.
- **Bricked volume data:** Bricks of all resolution levels need to be accessible individually to avoid having to load the whole data set into memory.

Depending on the implementation, the server may either only offer file storage, or it may have some additional capabilities, e.g., querying metadata for specific regions in the volume that can be used for empty-space skipping. Examples of such metadata are minimum and maximum scalar values, or a histogram of values within a region in the volume as proposed by Faludi et al. [FZZ+22].

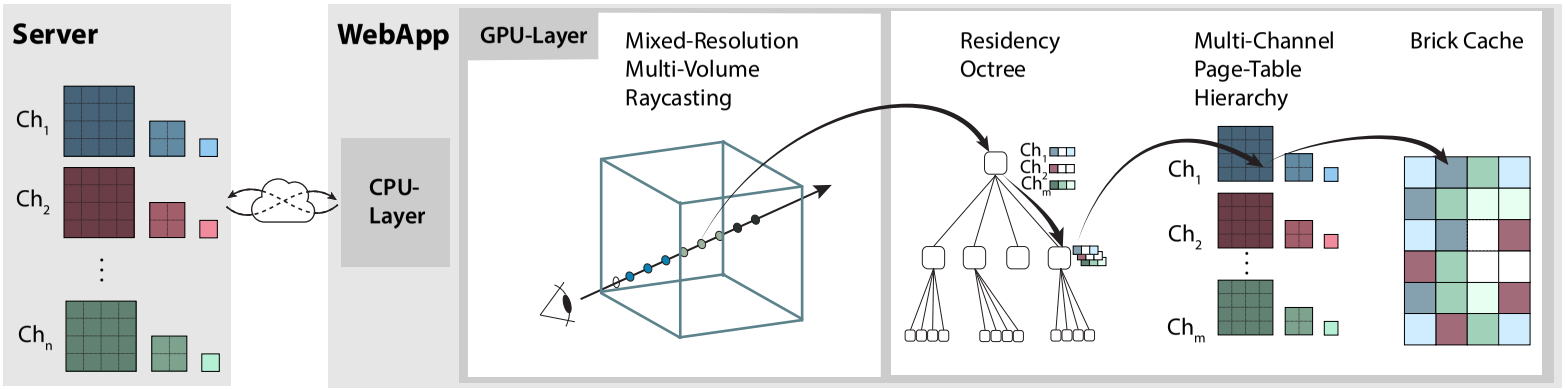


Figure 4.1: **System overview.** A bricked multi-resolution multi-volume hierarchy with n channels is provided by one or more file servers. On the client side, our mixed-resolution multi-volume renderer traverses the volume via the residency octree, with each octree node containing $m \leq n$ slots corresponding to the in-cache availability of different resolutions for different channels. Actual data access then references a multi-channel page-table hierarchy. Bricks are streamed into the cache on demand. The residency octree makes it possible to efficiently mix resolutions of different channels and substitute alternative resolutions on cache misses.

4.2 Client

Bricks and metadata are consumed by a client-side multi-volume rendering application that uses a ray-guided renderer to determine which data to fetch from the server. It consists of two main parts: the novel residency octree presented in detail in Chapter 6, and the mixed-resolution multi-volume renderer discussed in Chapter 7.

4.2.1 Residency Octree

The residency octree manages information about volume data on the GPU by keeping track of which bricks that are currently resident in the brick cache, partially or fully map to which octree nodes. It does this in conjunction with a page-table hierarchy similar to those presented in previous work [HBJP12, SCRL20], for multiple channels, as discussed in Chapter 5. Additionally, the residency octree provides transfer-function independent metadata for each node, which the renderer uses for empty-space skipping. For multi-volume data with n different channels, the residency octree can represent $m \leq n$ channels at a time. The maximum number of visible channels m is user-defined and set at initialization time. The mapping of channels in the data set to channels referenced by our hybrid data structure can change at run-time.

4.2.2 Mixed-Resolution Multi-Volume Renderer

The mixed-resolution multi-volume renderer accesses both metadata and cached volume data through the residency octree. If either is missing for a node, it generates a request for the missing data to be streamed in from the server. Since these requests are generated by ray traversal during rendering our method is ray-guided.

4.2.3 GPU-Driven Architecture

Both the residency octree and the mixed-resolution multi-volume renderer can be fully implemented on the GPU as shown in Chapter 8. The CPU is mainly needed for driving the overall rendering. It dispatches rendering and memory management commands to the GPU, forwards brick requests to the server, and uploads bricks received from the server to the GPU. In order to keep buffer copies from the GPU to the CPU small, we specify a global limit on the number of recorded cache misses per frame, similar to previous approaches [FSK13, SCRL20].

CHAPTER 5

Multi-Channel Page-Table Hierarchy

In this chapter, we discuss the multi-channel page-table hierarchy that forms the base of the residency octree presented in Chapter 6. We show how we extend the algorithms and data structures presented in earlier work to support multiple channels. Section 5.1 gives an overview of page-table hierarchies. How we extend this concept to multi-channel data is presented in Section 5.2. In Section 5.3, we show how voxels and bricks in the multi-channel page-table hierarchy can be addressed via a reference space using normalized coordinates in the range $[0, 1)^3$. We explain how brick IDs are used to communicate cache misses to the CPU in Sections 5.4 and 5.5. In Section 5.6, we discuss the LRU scheme used for managing the brick cache. Finally, we list the parameters of the multi-channel page-table hierarchy in Section 5.7.

5.1 Page-Table Hierarchy

Page-table hierarchies draw inspiration from memory virtualization schemes in operating systems [HBJP12]. They virtualize bricked volume hierarchies without enforcing down-sampling ratios or spatial subdivision schemes on data sets. Instead, each brick in the data set is an independent chunk of data, analogous to individual files in a file system. Thus there is no implicit hierarchy between bricks of different resolution levels, allowing page-table hierarchies to support direct access to any brick in the data set without the need of traversing a tree structure such as an octree. Due to this direct access, page-table hierarchies are well suited for volume data management and are more efficient in terms of data-access patterns than other out-of-core volume rendering approaches such as octrees [HBJP12, FSK13, SCRL20]. A page table represents a resolution level in a bricked volume hierarchy and is thus a volume in itself where each voxel corresponds to a brick belonging to the page table's resolution level. A page-table hierarchy thus forms a

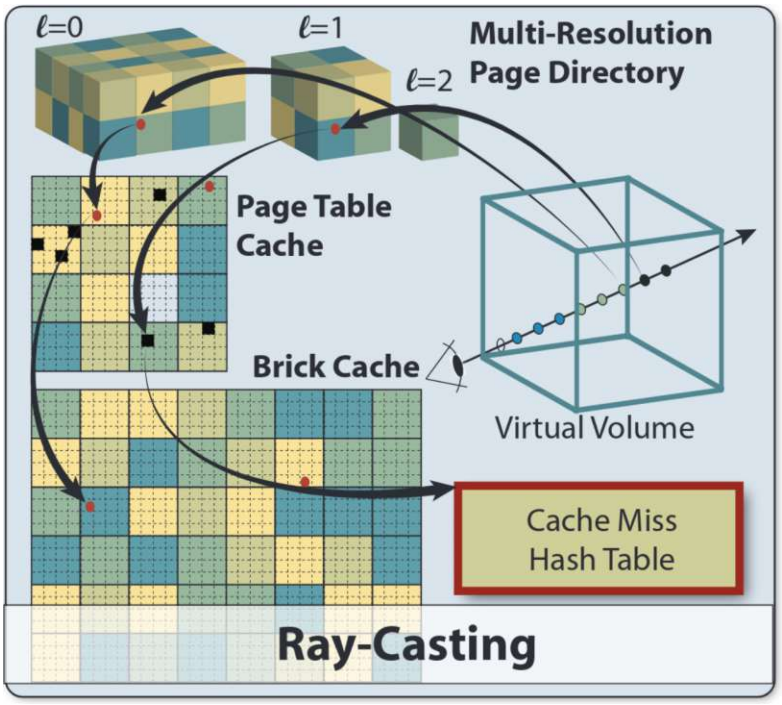


Figure 5.1: **Multi-resolution page-table hierarchy.** The page-table hierarchy (here: Multi-Resolution Page Directory) virtualizes the volume. It consists of one page table per resolution level in the data set (here: $l = 0$, $l = 1$, and $l = 2$) which keep track of the current location of their corresponding bricks in the brick cache. During ray casting, the page-table hierarchy is used for address translation. Missing bricks are reported in a hash table. In case the page-table hierarchy is virtualized itself, page tables are stored within a page-table cache. Source: [HBJP12]

volume hierarchy, which makes it possible to apply the virtualization scheme recursively to represent volumes of arbitrary dimensions [HBJP12]. Page tables virtualize volumes by translating normalized floating point coordinates, such as the coordinates of a ray sample, in a virtual volume space (e.g. $[0, 1)^3$) to the page table's reference space, or address space (Section 5.3). Figure 5.1 illustrates how page-table hierarchies are used in out-of-core volume rendering to access bricks of any resolution level directly.

5.2 Extension to Multi-Volume Data

We use a page-table hierarchy to manage volume data on the GPU as proposed in earlier work [HBJP12] but fully implemented in GPU kernels similarly to earlier implementations [SCRL20], extended to multiple channels, and implemented in WebGPU (Chapter 8). For scalability, we utilize a brick cache that stores a working set of bricks on the GPU, referenced by a multi-channel page-table hierarchy illustrated in Figure 5.2. The latter fully virtualizes a bricked multi-volume hierarchy, by implementing a paged 3D address

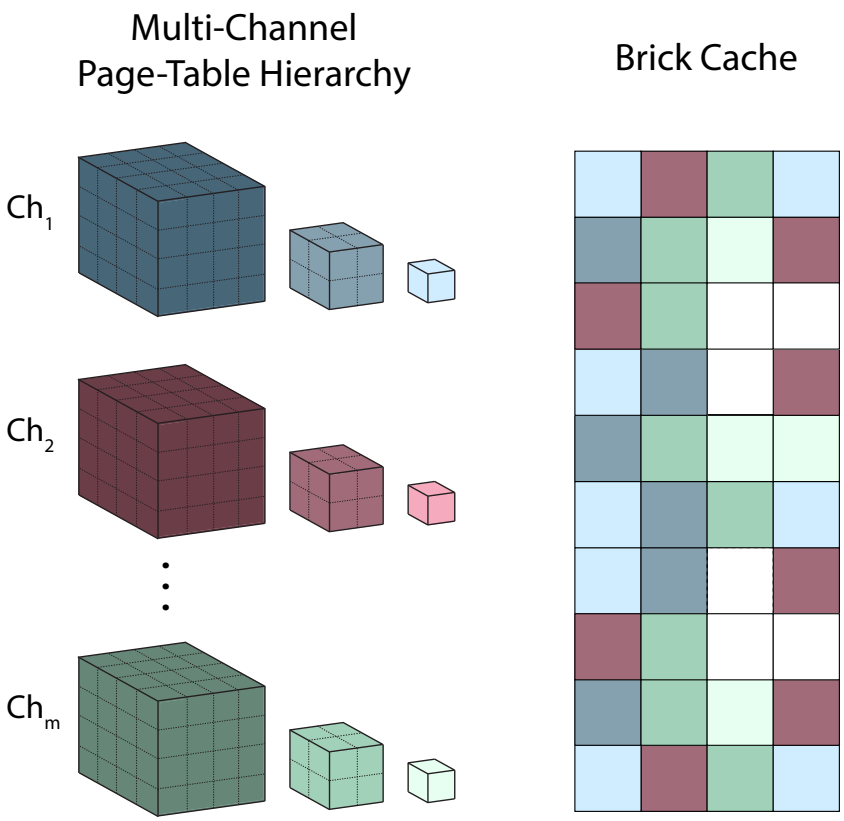


Figure 5.2: **Multi-channel page-table hierarchy.** The page-table hierarchy (left) is extended to keep track of multiple channels using one page-table hierarchy per channel. All channels use the same brick cache (right) with all bricks having the same size in voxels.

space with page-table entries storing all information required for virtual-to-physical address translation and keeping track of the in-cache-residency of bricks. Each page table in the page-table hierarchy represents a single resolution level in the volume hierarchy, each comprising multiple pages that allow referencing all bricks corresponding to the page tables resolution level. Each page stores if its corresponding brick is resident in the brick cache or not and if so, a pointer to its location in the brick cache. In practice, the brick cache is implemented as a buffer or 3D texture [HBJP12, FSK13, SCRL20]. Pointers into the brick cache stored in pages are therefore the brick's offset within the cache. A page may also mark bricks that are known to be empty, i.e., if they contain only zero values, with an `EMPTY` flag to eliminate the need to store them in the brick cache.

Like previous out-of-core renderers [HBJP12, FSK13, SCRL20], all bricks in the multi-volume hierarchy have the same size in voxels even though they can have different spatial extents in the volume, depending on the resolution level they belong to. This greatly

simplifies cache management, since each entry in the brick cache has the exact same size. Pointers into the brick cache, i.e., brick offsets, are thus all an integer multiple of the brick size. For multi-channel volumes, we make the same assumption, such that extending this method to multiple channels is achieved by having one page-table hierarchy per channel, but all channels share the same brick cache.

5.3 Address Translation

To address voxel data, like earlier work [HBJP12, FSK13, SCRL20] we use virtual multi-resolution addresses $(l, \mathbf{p})$, where $l \in \mathbb{Z}$ is the resolution level and $\mathbf{p} \in [0,1)^3$ are the normalized floating-point coordinates in the virtualized volume's reference space ($[0,1)^3$).For a position $\mathbf{p} \in [0,1)^3$, e.g., a sample along a viewing ray, the address of the corresponding voxel $\mathbf{p}_v$ in a volume with extent $\mathbf{v} \in \mathbb{N}^3$ is computed as follows:

$$\mathbf{p}_v = \lfloor \mathbf{p} \circ \mathbf{v} \rfloor, \tag{5.1}$$

where $\circ$ is the element-wise product of two matrices, and $\lfloor\rfloor$ is the element-wise flooring operation. Similarly, Equation (5.1) is also used to compute the address of a page $\mathbf{p}_{v_{pt_l}}$ in the page-table pt_l at resolution level l, which is itself a volume with extent $\mathbf{v}_{pt_l} \in \mathbb{N}^3$. In practice, page-tables in a page-table hierarchy are often stored within the same buffer or texture, where each page-table has its own offset [HBJP12, FSK13, SCRL20]. In such a case, a page's address $\mathbf{p}_{v_{pt_l}}$ for a page-table at resolution level l has to take the page-table's offset $\mathbf{o}_l \in \mathbb{Z}^3$ into account:

$$\mathbf{p}_{v_{pt_l}} = \mathbf{o}_l + \lfloor \mathbf{p} \circ \mathbf{v}_{pt_l} \rfloor \tag{5.2}$$

In order to address a position in a multi-volume hierarchy, we extend the virtual multi-resolution addresses $(l, \mathbf{p})$ to virtual multi-resolution, multi-volume addresses $(l, c, \mathbf{p})$, where $c \in \mathbb{Z}$ is the channel index. Again, if all page tables of all resolutions and channels are stored in the same texture or buffer, a page table's offset $\mathbf{o}_{l,c}$ has to be taken into account when computing the page's address $\mathbf{p}_{v_{pt_{l,c}}}$.

To compute the address of a voxel $\mathbf{p}_b$ within a brick b for a position $\mathbf{p}$ using a page's pointer into the brick cache $\mathbf{o}_b$, i.e., the brick's offset into the texture used as a cache, we use the following equation:

$$\mathbf{p}_b = \mathbf{o}_b + \lfloor \mathbf{p} \circ \mathbf{v}_l \rfloor \texttt{mod} \mathbf{b}, \tag{5.3}$$

where $\mathbf{v}_l$ is the extent of the volume at resolution level l, $\mathbf{b}$ is the brick size, and $\texttt{mod}$ is the element-wise modulo operator.

5.4 Brick IDs

To efficiently communicate cache misses to the CPU, previous methods additionally assign unique integer IDs to bricks in bricked volume hierarchies that encode their position in the volume as well as their resolution level [HBJP12, FSK13, SCRL20]. As shown

by Hadwiger et al. [HBJP12], the bit-width of the data type used for brick IDs, e.g., 32-bit or 64-bit integers, constrains the maximum number of bricks in a volume that can be unequivocally identified by an integer ID. The number of bricks in a bricked volume hierarchy depends on the volume's size, the brick size in voxels, the number of resolution levels, and the down-sampling ratios between resolution levels, i.e., the number of bricks in each resolution level within the hierarchy. The number of bricks is reduced by increasing the brick size, reducing the number of resolution levels, or by more aggressive down-sampling.

To address bricks in bricked multi-volume hierarchies, we additionally encode the brick's channel index in its ID. The maximum number of channels that can be represented by a multi-channel page-table hierarchy is thus also constrained by the bit-width of the data type used for brick IDs. In order to still support more channels than can be represented by the multi-channel page-table hierarchy, we virtualize multi-channel data sets: We store a mapping of the channels currently used by our data structure to the channels in the multi-volume hierarchy. This mapping is used by the CPU layer of our system to translate brick IDs generated on the GPU to brick requests that are then forwarded to the server. This allows our system to address more channels and to exchange channels at run-time.

5.5 Brick Requests

We use the algorithm presented by Sarton et al. [SCRL20] for processing brick requests and transmitting them to the CPU: A 3D buffer, the *request buffer*, stores for each page in the multi-channel page-table hierarchy the timestamp it was last requested. If a page representing a brick that is not resident in the brick cache and is not known to be empty is accessed on the GPU, a cache miss is generated by writing the current timestamp to the page's corresponding location in the request buffer. To retrieve all cache misses for a given timestamp t, the brick IDs of all pages that have been requested at t are written to a request list as illustrated in Figure 5.3. This request list is then read back to the CPU to stream in the bricks that were reported missing from the server.

As Sarton et al. [SCRL20] proposed, the number of requested bricks per timestamp can be limited globally to limit traffic between the client and the server. This limit can either be a constant, e.g., defined by the user at initialization time, or a variable that changes at run-time. For example, the number of requested bricks can be decreased if working sets of successive frames are expected to be temporally incoherent, e.g., during user interactions, such as viewport changes.

5.6 Cache Management

We use the same LRU cache management scheme as Sarton et al. [SCRL20]: An additional buffer, the *LRU buffer*, stores a sorted list of linear indices into the brick cache referencing brick-cache entries. These indices are sorted by the last time the respective cache entry

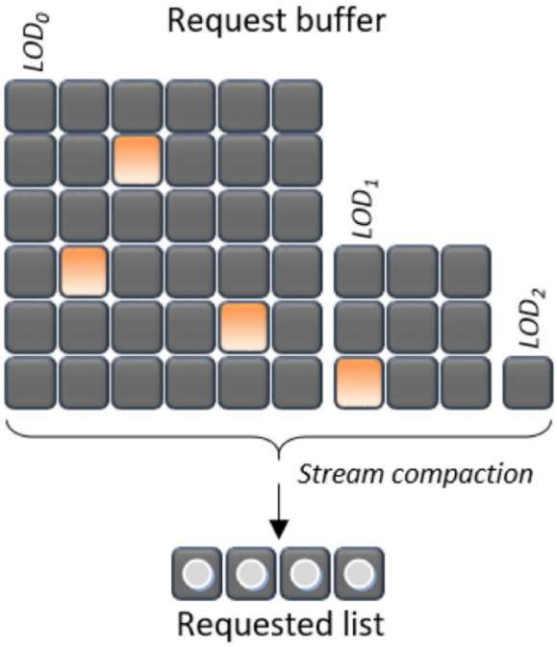


Figure 5.3: **Processing brick requests.** A list of brick requests is compiled from a global request buffer with one entry per brick in the data set. This request list is then read back to the CPU. Source: [SCRL20]

has been accessed on the GPU from the most to the least recent access. Furthermore, a buffer, the *usage-buffer*, stores for each entry in the brick cache the timestamp it was last accessed. It is used to update the LRU buffer. If a brick in the brick cache is accessed on the GPU, the usage of the brick is recorded by writing the current timestamp to the usage buffer. To update the LRU buffer using the usage buffer for a given timestamp t, the LRU buffer is rearranged such that all indices referencing brick-cache entries that have been accessed at t, are moved to the front while preserving the order of all other elements as illustrated in Figure 5.4.

5.7 Parameters

The multi-channel page-table hierarchy and its brick cache have the following parameters:

- The **brick size** in voxels, i.e., the size of a brick in the multi-volume hierarchy and the size of an entry in the brick cache.
- The **brick cache size** determines the maximum number of bricks in the working set.
- The **number m of channels** that can be referenced simultaneously by the page-table hierarchy.

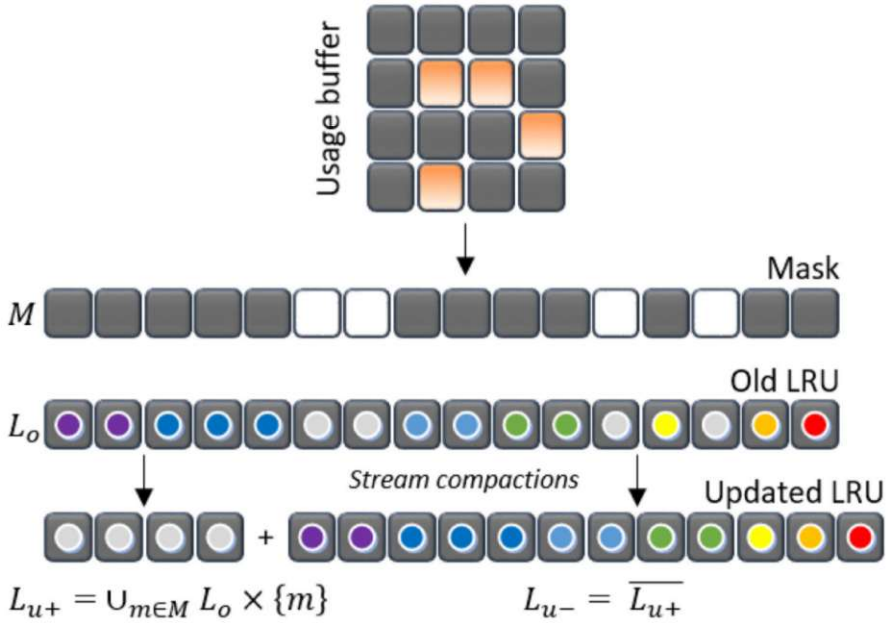


Figure 5.4: **Parallel cache management.** Brick usages are marked in a global usage buffer. A mask M is generated from the usage buffer to mark all indices in the old LRU buffer (here: Old LRU) L_o that have been used at the current timestamp. L_{u+}, the union of indices in L_o that are marked in M, is moved to the front of the updated LRU buffer (here: Updated LRU), while L_{u-}, the inverse of L_{u+}, is moved to the back of the buffer. The original order of indices in L_{u-} is preserved. Source: [SCRL20]

- The **brick ID data type** that is used for efficiently communicating brick requests to the CPU.

Note that depending on the platform, the data type used for brick IDs can not be set by the user. For example, shaders in WebGPU currently only support 32-bit integers.

CHAPTER 6

Residency Octree

In this chapter, we discuss the main contribution of this thesis: the residency octree. We first give a short overview of our data structure in Section 6.1 before describing its nodes in more detail in Section 6.2. Section 6.3 shows how residency octrees are extended to multiple channels to act as a base for efficient cache management and empty-space skipping for multi-volume data sets. The incremental construction of residency octrees is discussed in Section 6.4. Finally, the parameters of the residency octree are listed in Section 6.5.

6.1 Overview

The residency octree is a hybrid data structure for out-of-core multi-volume rendering combining the advantages of page-table hierarchies with those of octrees. It is illustrated in Figure 1.1, middle, and Figure 6.1. The residency octree is internally backed by a multi-channel page-table hierarchy discussed in the previous chapter, which references bricks of different resolution levels and channels stored in a single brick cache. In contrast to previous octree-based out-of-core methods [CNLE09, BPH14], the residency octree decouples the resolution levels in the bricked volume hierarchy from the spatial subdivision determined by the tree. Each node thus corresponds to a set of bricks in each resolution level instead of just one brick. Residency-octree nodes store the resolutions they are currently mapped in in a bitmap. This makes it possible to keep bricks of high-resolution levels resident in the cache without having to keep bricks of lower resolution levels that fall into the same spatial region of the volume in the cache if they are unused. For example, if a renderer is able to render the visible parts of a volume in the highest resolution level, the working set produced by a residency octree thus only consists of bricks of that resolution.

The decoupling of the tree's spatial subdivision and the volume hierarchy's resolution levels also allows residency octrees to subdivide the volume further than the spatial

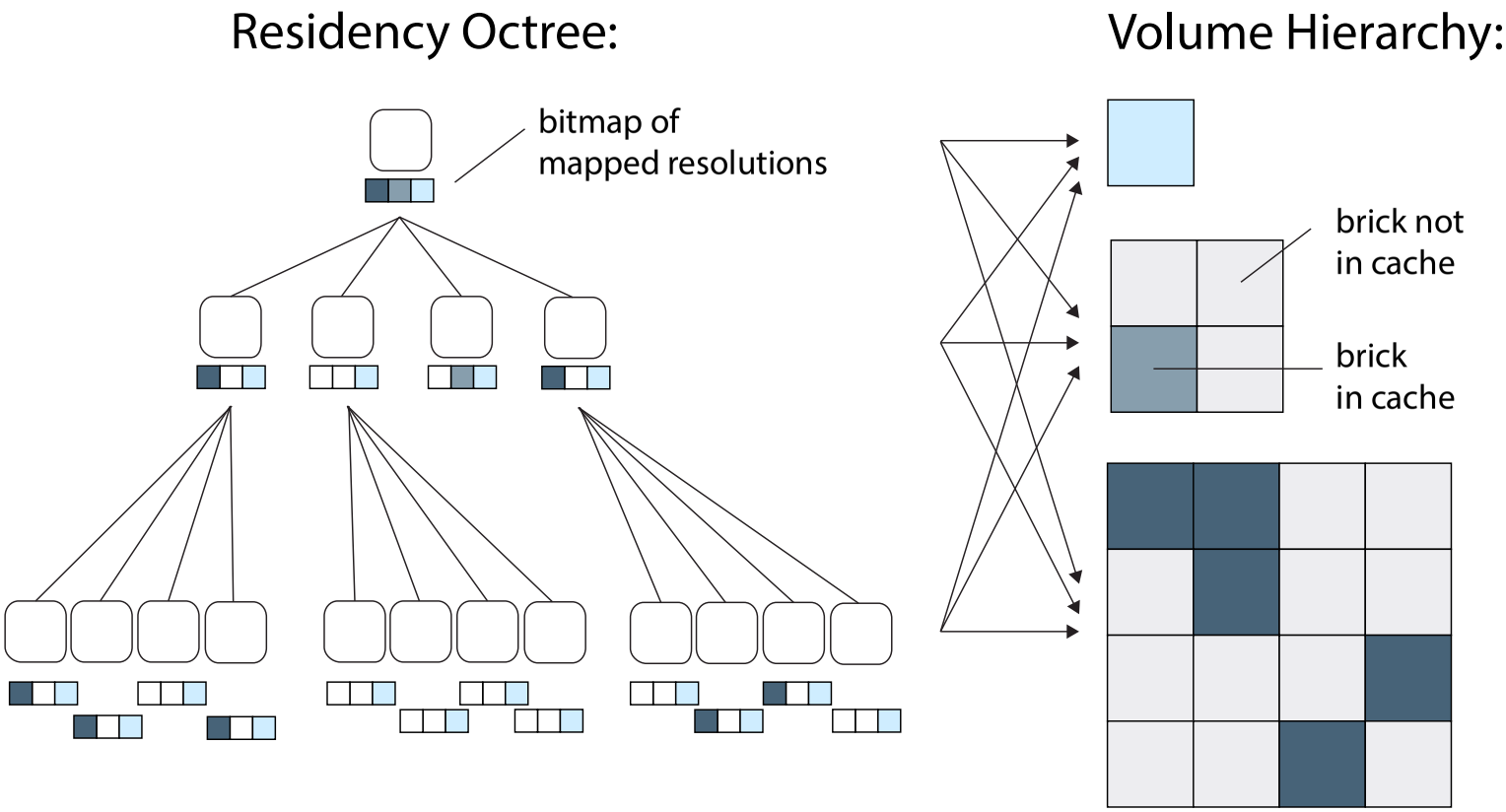


Figure 6.1: **Residency Octree.** Each octree node stores cache-residency information about all corresponding bricks of all resolution levels currently resident in the cache. The colors indicate the level of a brick in the resolution hierarchy. For illustration, here we focus on a single channel.

subdivision introduced by splitting the volume into bricks. As a result, residency octrees support more fine-grained empty-space skipping than previous approaches [CNLE09, HBJP12, FSK13, BPH14, SCRL20] for the same brick size used for cache management. By additionally storing transfer-function independent metadata, e.g., minimum and maximum scalar values, in its nodes, the empty-space skipping is further refined, such that nodes are not only considered empty if they contain only zero values but can be tested against a transfer-function's visible value range dynamically. In Section 6.2, we discuss the layout of residency-octree nodes, and in Section 6.4 we discuss how residency-octree nodes are updated when new data is streamed in.

6.2 Residency-Octree Nodes

As illustrated in Figure 6.2, for each resolution level, a residency octree node can be (1) *fully mapped*, (2) *partially mapped*, or (3) *not mapped* at all by the cache resident bricks. However, since we want to avoid having to load all bricks of a given resolution level when we only need a subset, we only keep track of each node's partial mapping. We use a ray-guided renderer to determine and stream in those parts of each node that currently contribute to the rendered image enabling our system to converge to a state where precisely those parts are mapped while unused parts of a node are not. Furthermore, each node represents a spatial region in the volume whose corresponding data contain a range of scalar values. Transfer-function independent metadata for this range, e.g., the minimum and maximum or a histogram of occurrences, is used to determine if a node is empty or homogeneous under the current viewing conditions. In these cases, a node's corresponding bricks do not need to be resident in the cache. If a node is empty or homogeneous, it is implicitly considered fully mapped in all resolution levels during octree traversal.

As illustrated in Figure 6.3, each node in the residency octree stores the following:

- Transfer-function independent metadata that can be used for empty-space skipping.
- Pointers to its children.
- A bitmap storing in which resolution levels the node is at least partially mapped.

Since the residency octree only stores metadata and information about cache residency, it is not necessary to load successive resolution levels or to load them as a whole. Instead, individual bricks can be loaded as needed while the tree is constructed and updated incrementally for those regions that are of interest as determined by the ray-guided renderer. Since the metadata stored in octree nodes is transfer-function independent and is not tied to a specific brick, only the residency information of a node needs to be updated when a brick is added to or removed from the brick cache (Section 6.4). On the other hand, because of the hierarchical structure of residency octrees, a node's metadata can be computed from its children as shown in Figure 6.3. Residency octrees can thus be constructed in parallel from the bottom up as discussed in Section 6.4 and Chapter 8.

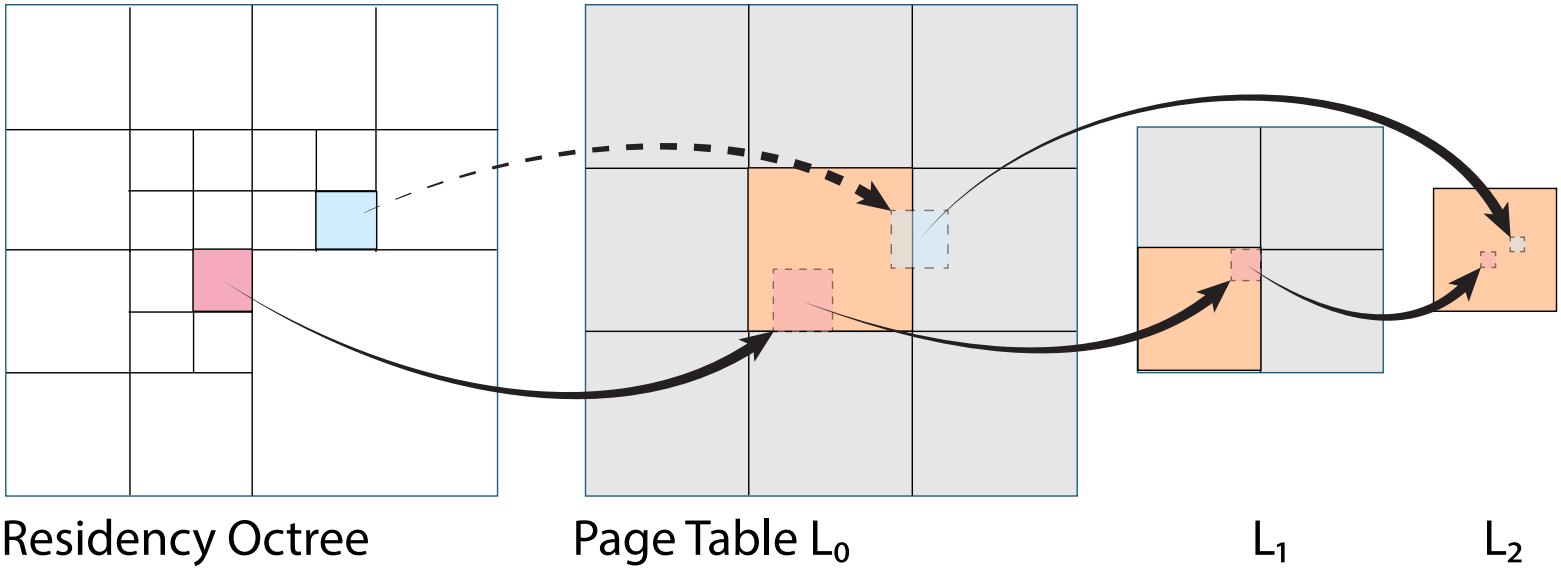


Figure 6.2: **Fully vs. partially mapped octree nodes.** Orange bricks in the page table (right) are resident in the cache. Dashed lines indicate that a node from the residency octree is partially mapped, and solid lines indicate that a node is fully mapped. While the red node in the octree (left) is fully mapped in all resolution levels, L_0, L_1, and L_2, in the page table (right), the blue node is only partially mapped in L_0, and fully mapped in L_2.

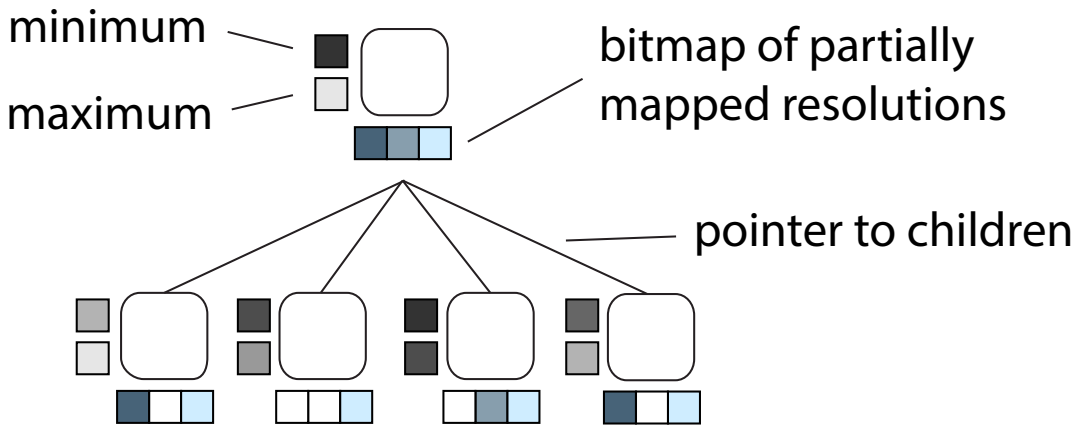


Figure 6.3: **Residency-octree nodes** store transfer-function independent culling information (here: minimum and maximum scalar values), pointers to their children, and a bitmap of partially mapped resolutions. A node's culling information can be computed from its children (here: the parent node's minimum and maximum are the minimum and maximum values of its children's respective values).

6.3 Extension to Multiple Channels

To extend the residency octree to multi-channel data, all channels share the same octree structure, but each node stores culling and residency information for up to $m \leq n$ channels, where m is the number of channels that can be referenced simultaneously in each octree node. Using the same octree structure for all channels while keeping a node's residency information independent for all channels makes it possible to use the same spatial subdivision for empty-space skipping, while at the same time rendering each channel in a different resolution.

Since the set of visible channels may change at run-time and volume data may be streamed in on a per-channel basis rather than loading all channels at once, a node may already store culling and residency information for one channel while this information is still missing for other channels. Therefore, each node also stores for each of the m channels whether the channel information it contains is already initialized or not using a special `INVALID` value. If a node's data for a channel is invalid, it stores a pointer to the node in the next lower subdivision level that has valid information for that channel, or a special `UNKNOWN` value if no such node exists. To reduce the memory required for each node, the pointer to the next node storing valid information replaces the metadata stored for that channel. Whenever a channel is replaced by another one, all nodes storing valid data for the old channel are marked as `INVALID` for the new channel.

6.4 Residency-Octree Updates

Residency-octree nodes store metadata and residency information, accessible during tree traversal. The residency octree needs to be updated when new metadata or bricks are streamed in from the server.

6.4.1 Updating Culling Information

As soon as new metadata for a volume region has been received by the client, the corresponding residency-octree node needs to be either (1) created and added to the octree, or (2) updated if it already exists. Whenever a new node is added to the residency octree, its metadata is initialized as `INVALID` for all channels except for the channel for which the metadata was received. The new node's residency information is also only initialized for this channel, by checking for each resolution level whether the new node is partially mapped or not. If the node already existed in the residency octree, its channel-specific metadata is updated and its residency information is initialized similarly. This only happens for multi-channel volumes (Section 6.2) when a node already holds metadata for another channel.

In case the server does not support requesting metadata for a region in the volume, e.g., if it is just a file server, this information has to be computed on the fly from the actual brick (voxel) data on the client. Since a residency-octree node is not tied to a specific resolution, it is up to the application to choose a resolution level for which the corresponding set of bricks should be fetched from the server in order to have sufficient data to compute the metadata. For example, if the resolution is too coarse it might happen that only a single voxel is taken into account for computing the culling data of a node. To avoid cases where too few voxels result in inaccurate metadata, a residency octree may specify a minimum number of voxels that have to be taken into account when computing the metadata for a node's spatial extent. If the client had to fetch new brick data in order to compute the node's culling data, each new brick can then also be uploaded to the GPU. This requires residency information to be updated as well.

6.4.2 Updating Residency Information

For each new brick received by the client, an available slot in the brick cache is selected for storing the new brick. Whenever a brick is stored in the brick cache, we determine all residency-octree leaf nodes whose spatial extent overlaps the spatial extent of the brick to mark them as partially mapped in the brick's resolution level and channel. This update is then propagated up the tree until the root node is reached.

Residency information is stored as bitmaps, and can therefore be updated for all non-leaf nodes by combining the bitmaps of child nodes via bit-wise `OR`. This is possible because residency-octree nodes store partial residency information and therefore only require a single brick of a resolution level to be resident in the brick cache in order to be partially

mapped in that resolution level. As soon as a node's bitmap remains unchanged by this operation, we terminate the upwards propagation of the update.

6.4.3 LRU Cache Eviction

If no brick-cache slot is available for a new brick, we use an LRU scheme as used in earlier work [SCRL20] to evict the least recently accessed brick from the cache in order to create an available slot. When a brick is evicted from the cache, we check for all leaf nodes whose spatial extent overlaps with the brick's spatial extent if they are still partially mapped after the evicted brick's overlap is removed. This is done by checking for each of a node's corresponding bricks in the resolution level that the evicted brick belongs to, if they are resident in the cache or not. Only if no other brick in this resolution level is resident in the brick cache, the node is no longer partially mapped in that resolution level, and its corresponding bit in the node's bitmap is set to zero. Updating non-leaf nodes is done in the same fashion as for newly added bricks, by setting their bitmaps to the bit-wise `OR` of their child nodes' bitmaps.

6.5 Parameters

The parameters of our multi-channel residency octree are

- The **number of spatial subdivisions** used to construct the tree.
- The **number of channels** that can be referenced by the residency octree.

Since the residency octree is backed by a multi-channel page-table hierarchy discussed in Chapter 5, the number of channels that can be referenced by the octree is equal to the number of channels that can be referenced by the page-table hierarchy.

CHAPTER 7

Mixed-Resolution Multi-Volume Rendering

In this chapter, we discuss our mixed-resolution multi-volume rendering algorithm. The residency-octree structure discussed in Chapter 6 handles cache-residency information and provides access to the volume data in conjunction with the multi-channel page-table hierarchy (Chapter 5). However, it does not control what data is needed on the GPU directly. Instead, we use a ray-guided volume-rendering algorithm in order to determine which data is needed during rendering. To render a scene, we march through the volume along viewing rays, and at each step traverse the residency octree (Section 7.1) to find the leaf node containing the current sample. The leaf node's metadata is then used to determine if the sample falls into a region of empty space that can be skipped, or if it might contribute to the rendered image. In the latter case, the node's residency information is used to fetch a brick currently resident in the brick cache in order to access its voxels for rendering. If the leaf node's metadata is missing or invalid, or no corresponding brick is resident in the brick cache, the respective data is requested from the server. Furthermore, we use the residency information stored in residency-octree nodes to substitute missing brick data in the desired resolution with bricks of another resolution as discussed in Section 7.2. In Section 7.3, we show how we extend the ray-guided algorithm to multiple channels. To save memory when visualizing multiple channels, our algorithm supports controlling the resolution levels used for rendering on a per-channel basis. This is discussed in Section 7.4.

7.1 Residency-Octree Traversal

We traverse the residency octree for each sample within the volume along a viewing ray. The traversal algorithm for a single channel is outlined in Algorithm 7.1. For brevity, we assume that the root node already contains metadata. Before traversing the residency

Algorithm 7.1: Residency-octree traversal. For each sample along the ray, we traverse the tree until we reach a maximum traversal depth or we find an empty node that allows us to skip over empty space. Missing metadata and bricks are reported so they can be streamed in from the server.

```
1  resolutionLevel ← chooseResolutionLevel(depth(ray));
2  traversalDepth ← chooseTraversalDepth(stepSize(ray));
3  node ← rootNode;
4  for ( i ← 0; i ≤ traversalDepth; i++ ) {
5      if isEmpty(node) then
6          ray ← skipNode(node, ray);
7          break
8      end
9      if isHomogeneous(node) then
10         ray ← renderAndSkipNode(node, ray);
11         break
12     end
13     if not(isPartiallyMapped(node)) then
14         reportBrickRequest(node, ray, resolutionLevel);
15         ray ← skipNode(node, ray);
16         break
17     end
18     if i < traversalDepth then
19         nextNode ← next(node, ray);
20         if nextNode.missing then
21             reportNodeRequest(nextNode);
22         else
23             node ← nextNode;
24             continue
25         end
26     end
27     page ← getPage(node, ray, resolutionLevel);
28     if page.unmapped then
29         reportBrickRequest(node, ray, resolutionLevel);
30         page ← getAlternativePage(node, ray, resolutionLevel);
31     end
32     if page.unmapped then
33         skipBrick(node, ray);
34     else
35         renderUntilNextBoundary(getBrick(page), node, ray);
36     end
37     break
38 }
```

octree, we choose the desired resolution level for the ray sample based on current viewing parameters such as the sample's distance to the camera. Furthermore, we choose a maximum traversal depth based on the current sampling step size, to avoid skipping a distance smaller than the distance to the next ray sample, and define the current node to be the root node. The tree is only traversed until this maximum traversal depth is reached.

For each node, we first check if it is empty, denoted here as the function `isEmpty`. Internally, `isEmpty` tests the node's culling information, e.g., the minimum and maximum scalar values in the node, against the current transfer function's visible range. If a node is empty, all samples along the ray that would fall into the empty node are skipped by intersecting the ray with the node's bounding box and advancing it to the position where it exits the node.

Similarly, we check if the current node is homogeneous, i.e., all voxels hold (approximately) the same value, using the function `isHomogeneous`. If a node is homogeneous, the node's value is used directly for rendering all samples along the ray that would fall into the node, and the ray is advanced to the next node. Using the same value for all samples in the same node avoids unnecessary tree traversals. The exact value used for rendering is implementation-dependent. For example, if a node stores minimum and maximum scalar values, the value can be either one of the extremes or a linear combination of the two.

The function `isPartiallyMapped` checks if a node is partially mapped in any resolution, and `not` denotes the logical negation of a boolean value. If a node is not partially mapped, all of its children are implicitly also not mapped. Thus we can stop the octree traversal immediately, report a brick request, and skip all samples that fall into the node in such a case. The function `reportBrickRequest` reports the node's brick in the chosen resolution level as requested using the request buffer discussed in Section 5.5.

If at this point, the target traversal depth is not yet reached, we try to advance the tree traversal to the next subdivision level, i.e., the node's child node. However, if the child node does not yet have any culling information (here: checked using `nextNode.missing`), the whole subtree is implicitly known to not contain any culling information. In that case, the child node's culling information is requested using the function `reportNodeRequest`, and the current node is used for rendering instead as if the target traversal depth would have been reached.

If the target traversal depth is reached and the current node is non-empty and inhomogeneous, the page of the chosen resolution level is retrieved from the page-table hierarchy. If the page is unmapped, i.e., its brick is not resident in the cache, a brick request is reported and we try to find an alternative resolution level in which the node is currently partially mapped using the function `getAlternativePage`. We discuss how a node's bitmap of partially mapped resolutions is used to find an alternative resolution in Section 7.2.

If the region in the residency octree node that the current sample falls into is not mapped, we advance the ray to the boundary of the current page's brick. Note that the next ray

sample may still be within the same node. Otherwise, we render all samples that fall into the brick. If the resolution level of this brick is lower than the desired resolution for the current sample's distance to the camera, the sampling rate along the ray can be decreased for the ray segment within the brick's bounds.

As an optimization, octree traversal may also start at a higher subdivision level, assuming lower subdivision levels are non-empty anyway. For example, the root node can usually be expected to be non-empty as pointed out by Faludi et al. [FZZ+22].

7.2 Choosing Alternative Resolutions

If a brick is not cache resident in the desired resolution, we try to render a brick of an alternative resolution level instead. Thanks to the bitmaps of partially mapped resolutions in each residency-octree node, we can use this information to check if bricks of those resolutions are in the cache directly, instead of having to do a random search through all resolution levels. The algorithms to compute the next lower and higher resolution levels a node is partially mapped in are given in Section 7.2.1 and Section 7.2.2 respectively. However, since a node only stores if it is partially mapped, a ray sample may still fall into a region within the node that is not mapped. We thus may have to iterate over candidate resolution levels retrieved from a node's bitmap.

Using the building blocks given in Sections 7.2.1 and 7.2.2, different strategies are possible. For example, a renderer can choose to only use lower resolutions a node is partially mapped in, prioritizing a low memory footprint over rendering quality. Similarly, only resolutions higher than the desired one could be taken into account. Other strategies include iterating over higher resolutions before taking lower resolutions into account, alternating between lower and higher resolutions, or choosing the resolution level that is closest to the desired one. In our mixed-resolution multi-volume renderer, we first try to find a brick from a lower resolution, before trying to find a higher resolution brick that is resident in the cache instead as shown in Algorithm 7.2.

7.2.1 Finding Lower Resolution Levels

To access the next lower resolution level in which an octree node is partially mapped, given its bitmap of partially mapped resolutions and the index of the desired resolution level, i.e., the one in which the node is not mapped, we use Algorithm 7.3.

We first compute a mask of all resolution levels in which the residency-octree node is partially mapped, and that are lower than the one we already checked. For each resolution level, the function `getLowerResolutionsMask` returns a bitmask where only the bits corresponding to resolution levels lower than the given index are ones. These masks are constant. The function can thus be implemented as an access into a constant array. The mask of all lower resolutions the node is partially mapped in is then the result of a logical `AND` of the node's bitmap and the mask returned by `getLowerResolutionsMask`.

Algorithm 7.2: Choosing an alternate resolution given a node's bitmap of partially mapped resolutions and the index of a resolution level.

```
1  nextResolution ← nextLower(node.bitmap, resolutionLevel);
2  page ← getPage(node, ray, resolutionLevel);
3  while nextResolution ≠ NONE ∧ page.unmapped do
4  |  nextResolution ← nextLower(node.bitmap, resolutionLevel);
5  |  page ← getPage(node, ray, resolutionLevel);
6  end
7  if page.unmapped then
8  |  nextResolution ← nextHigher(node.bitmap, resolutionLevel);
9  |  while nextResolution ≠ NONE ∧ page.unmapped do
10 |  |  nextResolution ← nextHigher(node.bitmap, resolutionLevel);
11 |  |  page ← getPage(node, ray, resolutionLevel);
12 |  end
13 end
```

Algorithm 7.3: Choosing a lower resolution given a node's bitmap of partially mapped resolutions and the index of a resolution level.

input : `partiallyMapped`: the node's bitmap of partially mapped resolutions.
`resolution`: the index of the resolution last tested.
output : The index of the next lower resolution.

```
/* Only those bits of "lower" corresponding to resolutions
   where the node is partially mapped and are strictly
   lower than the given resolution level will be ones.   */
1  lower ← partiallyMapped ∧ getLowerResolutionsMask(resolution);
2  if lower = 0 then
3  |  return NONE;
4  end
/* Resolution levels are ordered from 0 (highest) to k
   (lowest). The index of the next lower resolution level
   the node is partially mapped in, is the position of the
   first trailing one-bit of "lower".                    */
5  return firstTrailingBit(lower);
```

If the node is not mapped in any resolution lower than the given one, i.e., if the computed bitmask is zero, we return the special value `NONE`. Otherwise, we use the function `firstTrailingBit` to find the index of the first trailing one-bit of the mask computed in Line 1 of Algorithm 7.3. This is equal to the index of the next lower resolution level's index because we order resolutions from 0 (highest) to k (lowest). In case the function `firstTrailingBit` is not available on a platform, the index of the first trailing one-bit of an integer is also equal to the number of consecutive trailing zero-bits of an integer counting from the least significant bit.

7.2.2 Finding Higher Resolution Levels

The algorithm to find the next higher resolution level a residency-octree node is partially mapped in, is similar to the one choosing a lower resolution. It is listed as Algorithm 7.4.

Algorithm 7.4: Choosing a higher resolution given a node's bitmap of partially mapped resolutions and the index of a resolution level.

input : `partiallyMapped`: the node's bitmap of partially mapped resolutions.
`resolution`: the index of the resolution last tested.
output : The index of the next higher resolution.

```
/* Only those bits of "higher" corresponding to
   resolutions where the node is partially mapped and are
   strictly higher than the given resolution level will be
   ones.                                                  */
1 higher ← partiallyMapped ∧ getHigherResolutionsMask(resolution);
2 if higher = 0 then
3 |  return NONE;
4 end
  /* Resolution levels are ordered from 0 (highest) to k
     (lowest).  The index of the next higher resolution
     level the node is partially mapped in, is the position
     of the first leading one-bit of "higher".            */
5 return firstLeadingBit(higher);
```

As in Algorithm 7.3, we first compute a mask of all resolution levels the node is partially mapped in and that are strictly higher than the index of the given resolution level. Similar to `getLowerResolutionsMask`, the function `getHigherResolutionsMask` returns a bitmask where only the bits corresponding to resolution levels higher than the given resolution level are ones.

Again, if the node is not mapped in any resolution higher than the given one, we return the special value `NONE`. Otherwise, we compute the index of the next higher resolution the node is partially mapped in, by finding the index of the first leading one-bit of the mask computed in Line 1 of Algorithm 7.4.

If `firstLeadingBit` is not available on a platform, the index i of the first leading one-bit of an integer x can be computed in terms of `countLeadingZeros` instead:

$$i = \texttt{BIT\_WIDTH} - 1 - \texttt{countLeadingZeros(x)}, \tag{7.1}$$

where `BIT_WIDTH` is the bit-width of x.

7.3 Extension to Multiple Channels

For rendering multiple channels, we use a similar algorithm as the one presented in Section 7.1. The main difference is that we cannot terminate the traversal before all channels have been processed. The channels are organized based on their importance that, together with the current viewing parameters, determines both the sampling rate and the desired traversal depth (Section 7.4). Traversing the entire tree for each sample and channel would be computationally inefficient. Instead, we only traverse the tree hierarchy once, starting with the most important channel, i.e., the channel requiring the highest sampling frequency. To do this, the algorithm outlined in Algorithm 7.5 keeps track of an index in the sequence of channels. As soon as we reach a point where we would terminate the traversal in the single channel case, we simply stay in the same node but increase this index, and continue the traversal from there.

We start processing each node by checking if it holds any culling information for the current channel index. If it does not, we request culling data for this node and the current channel and continue with the next channel.

Because we have to take all visible channels into account, we can not skip a node if it is only empty for one channel. Instead, empty nodes can only be skipped if all channels have been found to be empty. For this reason, our algorithm keeps track of the number of channels for which empty nodes were found for a sample. While a node might be empty for one channel, another channel might require a few more traversal steps to reach a subdivision level in which the corresponding node is empty. The skipped distance is determined by the spatial extent of the last empty node we encounter during traversal. Since the spatial extents of nodes are strictly decreasing between tree traversal steps, we are sure to not skip non-empty space using this strategy. Figure 7.1 gives an example of how empty-space skipping may require more traversal steps in a multi-volume setting compared to a single-channel setting because multiple channels have to be checked instead of only one.

Similarly, homogeneous nodes can also only be rendered more efficiently due to their homogeneity if they are homogeneous for all channels. The algorithm keeps a count of channels for which homogeneous nodes have been encountered during octree traversal. Similar to empty nodes, the last homogeneous node we encounter during traversal determines the spatial extent that can be skipped. Furthermore, the last homogeneous node's values are used for rendering the node for each channel.

The desired resolution level for each channel is only determined if it is necessary, i.e., when bricks are requested or accessed for rendering. Since resolution levels are chosen

Algorithm 7.5: Residency-octree traversal for multiple channels.

```
1  traversalDepth ← chooseTraversalDepth(stepSize(ray));
2  node ← rootNode;
3  channelIndex ← 0;
4  emptyChannels ← 0;
5  while node.depth ≤ traversalDepth ∧ channelIndex <
     numChannels do
6      if noCullingInformation(node, channelIndex) then
7          reportNodeRequest(node, channelIndex);
8          channelIndex++;
9          continue
10     end
11     if isEmpty(node, channelIndex) then
12         if emptyChannels = numChannels − 1 then
13             ray ← skipNode(node, ray);
14             break
15         end
16         emptyChannels++;
17         channelIndex++;
18         continue
19     end
       /* Homogeneous nodes are handled similarly to empty
          nodes.  Left out here for brevity.                  */
20     if not(isPartiallyMapped(node, channelIndex)) then
21         resolutionLevel ← chooseResolutionLevel(depth(ray),
            channelIndex);
22         reportBrickRequest(node, ray, resolutionLevel, channelIndex);
23         channelIndex++;
24         break
25     end
26     if node.depth < traversalDepth then
27         nextNode ← next(node, ray);
28         if nextNode.missing then
29             reportNodeRequest(nextNode);
30         else
31             node ← nextNode;
32             continue
33         end
34     end
35     resolutionLevel ← chooseResolutionLevel(depth(ray),
        channelIndex);
       /* Fetching and rendering a brick is similar to the
          single channel scenario and left out for brevity.  */
36     channelIndex++;
37 end
```

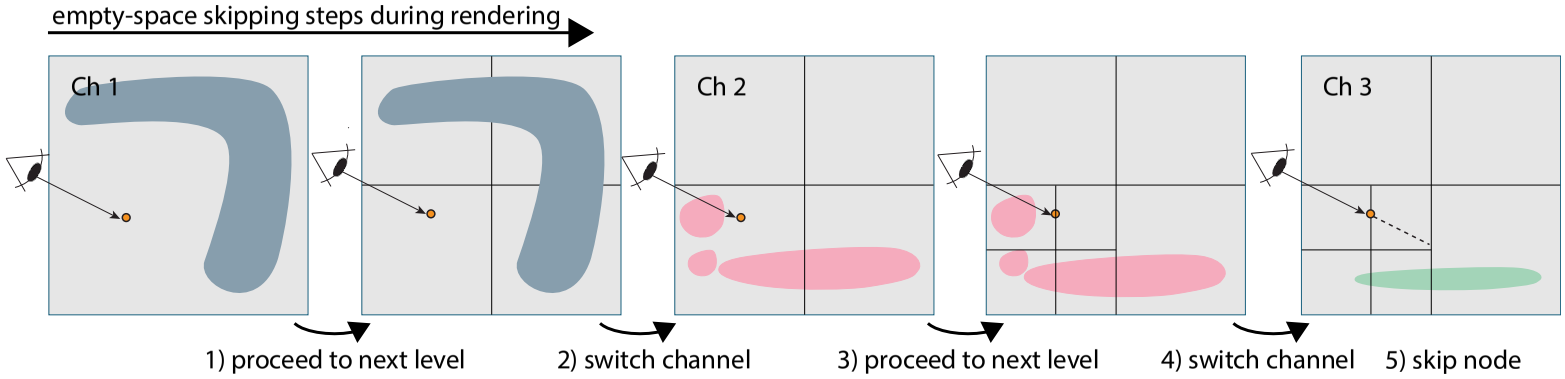


Figure 7.1: **Empty-space skipping for multiple channels.** Traversal starts at the root node and the first channel. The node is non-empty, so we proceed to the next subdivision level (1). For the first channel, this node is empty, so we can stop traversing the tree for this channel and proceed to the next one (2). The node is non-empty for channel 2, so we proceed to the next subdivision level (3). We repeat this procedure until we finally can skip the largest node in which all channels are empty (5).

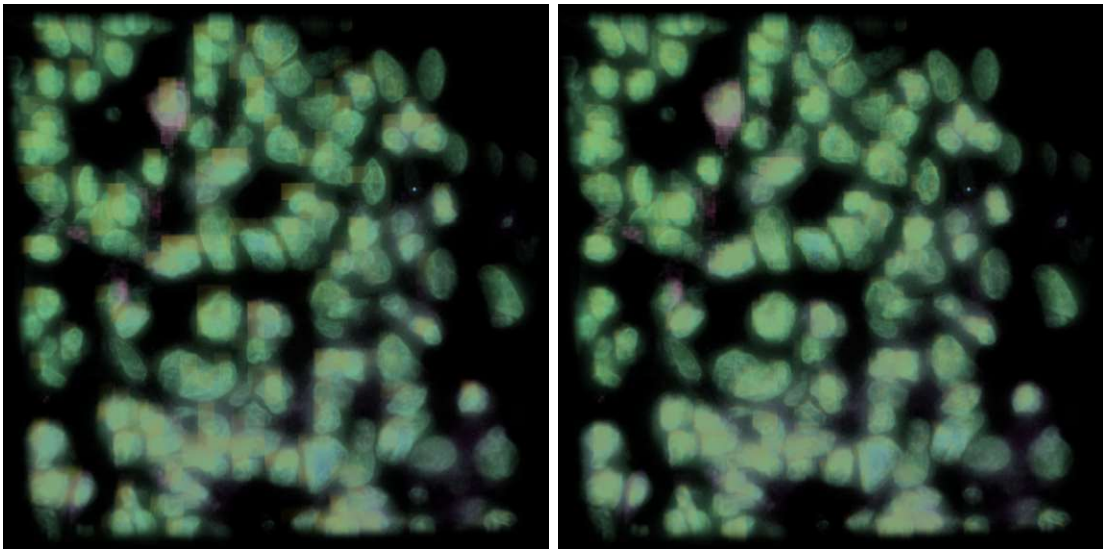
on a per-channel basis rather than choosing one globally for all channels, our renderer supports mixing resolutions (Section 7.4). The sequence of channels that need to be evaluated for a ray sample depends on the chosen sampling rate for each channel. For example, a channel with high-frequency content may require a higher sampling rate while another channel may only need to be sampled at every second step along the ray.

7.4 Mixing Resolutions

When rendering large-scale multi-volume data, it may be desirable to render different channels in different resolutions. This could be due to a channel having a lower frequency content than the others, or simply one channel not being as important as others for the user, possibly depending on the current viewpoint. Similarly, a volume might have high-frequency content only in some parts of the space.

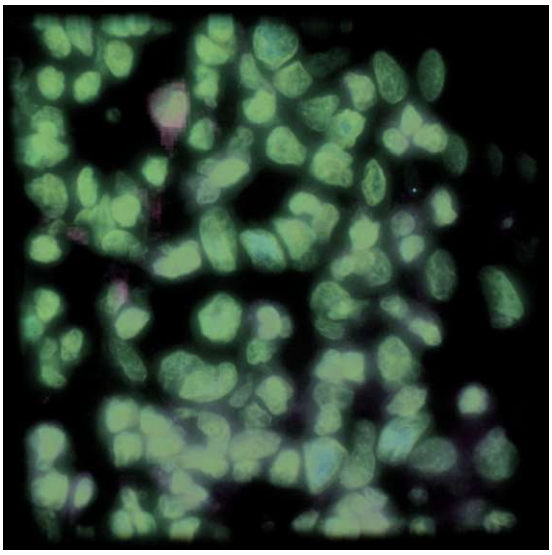
With large-scale data requiring out-of-core methods, we can exploit this by limiting the range of resolutions that are uploaded to the GPU on a per-channel basis to both save memory at run-time and reduce the number of samples that need to be evaluated for all channels. In our system, each channel has a function that constrains the range of resolution levels to choose from, when computing a resolution level for a channel. For example, it is based on current viewing parameters like a ray sample's distance to the camera. Conceptually, this function can be quite general, from user-controlled upper and lower bounds for resolution levels to a function taking into account multiple different parameters like frequency content, viewing parameters, and transfer functions.

Figure 7.2 gives an example of multi-channel human tissue data rendered with mixed resolutions. Here the resolution level for each channel is set explicitly for demonstration purposes. While some channels are always rendered in the highest resolution available, others are rendered at lower resolutions.

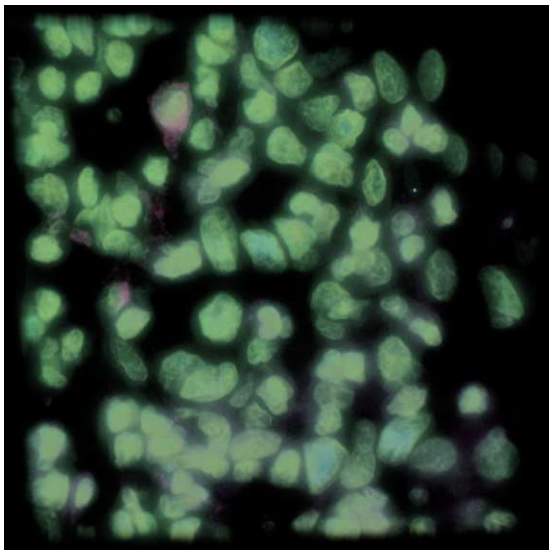


(a) pink = l_3, orange = l_5, rest = l_0

(b) pink = l_3, orange = l_4, rest = l_0



(c) pink = l_3, rest = l_0



(d) all = l_0

Figure 7.2: **Mixed resolution rendering of five channels of human tissue data.** The pink and orange channels are rendered at lower resolutions than the other channels, which are all rendered at the highest resolution level (l_0). The pink channel is rendered at resolution level l_3 (a-c), and the orange channel is rendered at resolution levels l_5 (a), l_4 (b), and l_0 (c,d). In (d) all channels are rendered at the highest resolution.

CHAPTER 8

Implementation

In this chapter, we discuss our implementation of the presented methods as a web-based, pure client-side volume rendering application that consumes bricked volume data provided by a server.

8.1 Server

For our implementation, we use a file server providing bricked multi-resolution multi-volume hierarchies as OME-Zarr files, an implementation of OME-NGFF [MAB+21]. We convert data sets in other formats, e.g., raw binary data, to OME-Zarr in a pre-processing step. Our server does not support querying culling metadata for specific regions in the volume, so they have to be computed on the client.

8.2 Client

Our client-side application is implemented in Rust and compiled into WebAssembly. It uses the WebGPU API for issuing GPU commands. It uses three separate threads: the main thread controls the UI, a render thread that runs our algorithm and handles all CPU-GPU communication, and a brick-loading thread that in turn uses a pool of worker threads to pre-process data fetched from the file server. Since WebGPU does not yet allow multiple threads to access the same GPU resources [MNJ23], brick data needs to be transmitted from the brick-loading thread to the render thread. In the render thread, received brick data is stored in a First-In-First-Out (FIFO) queue. At the beginning of each frame, a user-defined number of bricks is drained from this queue and uploaded to the GPU.

Data pre-processing

For simplicity, we convert all volume data fetched from the server to unsigned 8-bit integer data as a pre-processing step in the brick-loading thread. To convert values from a higher-precision data type, e.g., 16-bit integers, to unsigned 8-bit integer values, we apply one or more of the following pre-processing methods and scale the result to a range of $[0, 255]$:

- `scale`: Divide each value by a user-defined constant.
- `scaleToMax`: Divide each value by the channel's approximate maximum value. The channel's approximate maximum value is determined automatically by the application by using the maximum value in the lowest resolution representation of the channel.
- `logTransform`: Take the common logarithm of each value.

The pre-processing method or combination of pre-processing methods is chosen by the user at initialization time.

Furthermore, user-defined minimum and maximum thresholds in the range $[0, 1]$ may be specified. Each value outside of the range given by these thresholds, i.e., outside of the range $[min * 255, max * 255]$, where min and max are the user-defined thresholds, is set to zero during pre-processing. This is used to remove unwanted values from the data, e.g., noise produced during imaging. Instead of transferring bricks containing only zero values to the render thread, a special `EMPTY` value is transmitted as an optimization.

8.3 Multi-Channel Page-Table Hierarchy

The multi-channel page-table hierarchy is modeled after the fully GPU-driven approach presented by Sarton et al. [SCRL20]. However, we do not apply the memory virtualization recursively on the page-table hierarchy itself.

8.3.1 Page-Table Hierarchy

We use a single 3D texture in the `rgba32uint` texture format to store the multi-channel page-table hierarchy. Each page table is a 3D sub-region within this texture, where each pixel represents a page, i.e., a single brick in the page table's corresponding channel and resolution level. A page stores a 3D offset into the brick cache in the RGB channels and a state flag in the alpha channel. The state of a brick is one of the following:

- `UNMAPPED`: the corresponding brick is currently not resident in the cache and thus the page's offset into the brick cache is invalid.
- `MAPPED`: the corresponding brick is resident in the cache. The page's offset into the brick cache is valid.

- `EMPTY`: the corresponding brick contains only zero values and thus does not need to be stored in the cache. The page's offset into the brick cache is invalid.

We use an additional page-table meadata-buffer to store an array of metadata about each individual page table in the multi-channel page-table hierarchy, like its offset within the 3D texture, on the GPU. Each individual page table's metadata is accessed using its index.

To report cache misses to the CPU, we use a 3D texture in the `r32uint` texture format with the same resolution as the multi-channel page-table hierarchy as a request buffer. For each page in the multi-channel page-table hierarchy, this request buffer stores the timestamp, i.e., the frame number, when it was requested for the last time. After each frame, the request buffer is processed to compile a list of requested brick IDs in a compute pass.

The multi-channel page-table hierarchy uses unsigned 32-bit integers for brick IDs. They consist of 24 bits for the spatial offset (8 bits each for the x, y, and z coordinates) of the brick relative to the origin within its channel and resolution, and 8 bits for the page-table index, i.e., the index into the page-table meadata-buffer. Having 8 bits for the page-table index allows for $(m \times k) \leq 256$ page tables in total, where m is the number of channels and k is the number of resolution levels that can be represented on the GPU. The parameters m and k are chosen by the user at initialization time.

Supporting arbitrary numbers of channels

A data set may have more than the m channels that can be represented on the GPU using 32-bit brick IDs. To support such data sets, our implementation stores a mapping of channels in the multi-channel page-table hierarchy to the channels in the data set and vice versa. The mapping can change at run-time through user interaction. This mapping is used to translate requested brick IDs that are unique with respect to the GPU-resident multi-channel page-table hierarchy to brick IDs unique with respect to the complete data set. We refer to these brick IDs as *local* and *global* brick IDs respectively.

Mapping a GPU buffer to the CPU, i.e., making its memory available for reading or writing on the host side, is an asynchronous operation in WebGPU [MNJ23]. Since waiting for this operation to complete would stall the CPU side of the application, we do not wait for the buffer containing requested brick IDs to be mapped. Instead, we check at the beginning of each frame if the buffer's memory is already mapped and if so, read its contents. Requested brick IDs are thus read with a delay of at least one frame. If the mapping of local to global brick IDs changes, local brick IDs produced during a previous frame may therefore be translated incorrectly. For this reason, we store not a single mapping of channels in the page-table hierarchy to channels in the data set, but a list of such mappings - each with an associated timestamp at which it replaced the previous mapping. Similarly, we store the timestamp with each set of local brick IDs at which it was created. This timestamp is used to determine the appropriate mapping for translating local brick IDs to global ones.

8.3.2 Brick Cache

Our implementation uses a single-channel 3D texture in the `r8unorm` texture format as its brick cache, i.e., each entry in the brick cache corresponds to a single brick in the multi-resolution multi-volume hierarchy. The brick cache implementation is agnostic to the number of channels in the data set. The cache size C_{size} is chosen by the user at initialization time and is rounded up to be a multiple of the brick size B_{size}.

The brick cache is managed in an LRU manner using an array of linear indices into the brick cache sorted by the most recent access time of the corresponding brick-cache entry. This array is only resident in GPU memory in a storage buffer called the LRU buffer that is updated each frame. Whenever a brick-cache entry is accessed during rendering, the current global timestamp, i.e., the current frame number, is written into a usage buffer. The usage buffer is implemented as a 3D texture in the `r32uint` format with one voxel per brick-cache entry. The usage-buffer size U_{size} is defined as

$$U_{size} = (\frac{C_{size_x}}{B_{size_x}}, \frac{C_{size_y}}{B_{size_y}}, \frac{C_{size_z}}{B_{size_z}}). \tag{8.1}$$

We use a series of WebGPU compute passes to update the LRU buffer after each frame in the following manner: As discussed in Section 5.6, we first process the usage buffer to produce a mask of brick-cache entries that have been accessed at the current timestamp. We then use a sequence of compute passes to compute the exclusive scan of this mask. The result of this scan is taken to move all indices in the LRU buffer referencing brick-cache entries that have been used at the current timestamp to the front of the LRU buffer while preserving the order of the remaining elements.

8.3.3 Cache Updates

A user-defined number of brick data received from the file server is uploaded to the GPU each frame. For each of the received bricks, the brick data comprises the brick's global brick ID, and either the special `EMPTY` value if it contains only zero values or the pre-processed brick. As a first step, the brick's global brick ID is translated to a local brick ID. If the brick's channel is currently not represented by the multi-channel page-table hierarchy, the brick is discarded. If the brick is `EMPTY`, only the brick's corresponding page in the multi-channel page-table hierarchy is marked as `EMPTY`. Otherwise, the brick's corresponding page is marked as `MAPPED`, and the brick's data is added to the brick cache. To do so, the LRU buffer is used to find either free cache entries or ones that have not been accessed recently, as discussed in Section 6.4.3. Whenever a brick is removed from the brick cache, its corresponding page is marked as `UNMAPPED`. When the brick cache update is completed, two lists of local brick IDs are passed to the residency octree for further processing:

- a list containing the local brick IDs of all bricks for which the corresponding page has been marked `EMPTY` or `MAPPED`, and

- a list containing the local brick IDs of all bricks for which the corresponding page has been marked `UNMAPPED` in the current update.

How these two lists are used in the residency-octree construction is discussed in detail in the next section.

8.4 Residency Octree

The residency octree is implemented as a full, pointerless octree for simplicity. Instead of allocating new octree nodes only when they are needed, all nodes are pre-allocated and have a fixed location in the storage buffer containing the residency-octree nodes. This removes the extra indirection introduced by child-node pointers in octree nodes when accessing the children of a node and thus simplifies data management in the octree. Each node consists of m unsigned 32-bit integers, where m is the number of channels. Each channel uses 16 bits for keeping track of partially mapped resolutions and 8 bits each for minimum and maximum scalar values in the region represented by the node.

We use minima and maxima instead of histograms of occurrences as proposed by Faludi et al. [FZZ+22] for empty-space skipping. While histograms support efficient testing against multi-modal transfer functions, they do so at the cost of precision by quantization. For unimodal transfer functions, extremes allow more control and are closer to optimal for empty-space skipping due to higher precision. The renderer is designed for multi-channel immunofluorescence data. Each channel represents the response of cell material to a single marker fluid leading to a single structure being captured in each channel. To visualize such structures, it is sufficient to use unimodal transfer functions and we thus decided to implement empty-space skipping using minimum and maximum values for each octree node.

8.4.1 Volume Subdivision

The number of subdivision levels is indirectly defined by the user at initialization time by specifying an approximate leaf-node size in voxels. This leaf-node size is then used to iteratively subdivide the volume until a subdivision level is reached where the size of a node in voxels is lower or equal to the given leaf-node size in each dimension. At each step, each of the volume's dimensions is either halved or not subdivided at all depending on the other two dimensions. For example, if in one iteration, a node's extent is twice as large in one dimension than it is in the other two, e.g., $N_{extent} = (4, 2, 2)$, where N_{extent} is a node's extent, then it will only be subdivided in that dimension. It is possible that the constructed octree actually resembles a quadtree or binary tree in some subdivision levels depending on the volume's dimension and the chosen approximate leaf-node size.

8.4.2 Incremental Construction

The incremental parallel construction of the multi-channel residency octree is fully implemented on the GPU. Since we use a full octree, nodes only have to be updated when bricks are added to or removed from the brick cache. Our application starts residency-octree updates directly after a frame's brick-cache update has been completed. We process updates from the leaf-node level up to the root node in a breadth-first manner but make use of indirect compute passes to determine during the upwards propagation if we can terminate the process early on the GPU. Starting at the leaf-node level, an update-candidate buffer is used to mark the parent nodes of updated nodes as potential candidates to process in the next higher level. We use a series of indirect compute passes to process the lists of added and removed local brick IDs in the following manner:

1. **Process removed bricks.** As a first step, for all bricks that are no longer resident in the cache, the corresponding leaf nodes in the residency octree are determined. For each node that is no longer partially resident in the removed brick's resolution, the bitmap of partially mapped resolutions is updated and its parent node is marked as an update candidate in the update-candidate buffer.

2. **Compute culling metadata for new bricks.** For each scalar value in each newly added brick, the corresponding leaf node in the residency octree is determined. If the value is lower than the node's minimum or greater than its maximum the node's respective culling metadata is updated and its parent node is marked as an update candidate in the update-candidate buffer. Note that a node's minimum and maximum values need to be written using atomic operations, i.e., `atomicMin` and `atomicMax` respectively, because multiple threads may update the same node concurrently.

3. **Process added bricks.** For each brick that has been added to the cache, all corresponding leaf nodes in the residency octree are determined and marked as partially mapped in the brick's resolution level. If a node's bitmap of partially mapped resolution is changed by this operation, its parent node is marked as an update candidate in the update-candidate buffer.

4. **Compile a list of update candidates.** The update-candidate buffer is processed to compile a list of all nodes that have been marked as update candidates and need to be processed in the next step. Furthermore, the update-candidate buffer is cleared for subsequent iterations. The number of update candidates is used to control the number of workgroups dispatched for the next indirect compute pass, i.e., Step 5. If no candidates are left to process, no further workgroups are dispatched, and the update process is terminated.

5. **Process update-candidates.** For each update candidate, the node's culling metadata and bitmap of partially mapped resolutions are recomputed from its child nodes. If at least one of the node's values is changed by this operation, the node's parent node is marked as an update candidate in the update-candidate buffer.

6. **Repeat.** Steps 4 and 5 are repeated until the root node is reached, or no update candidate is left.

8.5 Mixed-Resolution Multi-Volume Renderer

The mixed-resolution multi-volume renderer is implemented in a WebGPU compute shader. To reduce the number of diverging branches in the shader, each visible channel is evaluated at each sample along a ray even though some channels may be rendered at a lower resolution than others. The sampling rate along each ray can thus be determined globally for all channels instead of on a per-channel basis. It is computed in terms of the step size dt_{step} between two samples based on the channel requiring the highest sampling rate, i.e., the channel rendered at the highest resolution, using Equations (8.2) and (8.3).

$$dt_i = \frac{1}{v_i * s_i * d_i}, \tag{8.2}$$

where v_i is the volume's extent in dimension i, s_i is the voxel spacing in dimension i, d_i is the ray's direction in dimension i, and $i \in [0, 1, 2]$.

$$dt_{step} = \texttt{min}(dt_0, dt_1, dt_2) * stepScale, \tag{8.3}$$

where *stepScale* is a user-defined scaling factor to increase or decrease the sampling rate at run-time.

Our system's UI has a slider for each channel to control the lower and upper bound for the range of resolutions to choose from during rendering as discussed in Section 7.4. A channel's importance is determined by the user-defined range of resolutions, where a channel with a higher upper bound is considered more important than a channel that is allowed to be rendered at lower resolutions.

As an optimization, the renderer never starts tree traversal at the root level, but at a user-defined subdivision level, or the parent level of the subdivision level at which the tree traversal was terminated for the previous sample. Since we use a full octree, this optimization does not require any changes to our shader code. Instead, each node can be accessed directly via its index.

CHAPTER 9

Evaluation and Results

In this chapter, we discuss how we evaluate the residency octree and mixed-resolution multi-volume rendering algorithm proposed in this work using the implementation discussed in Chapter 8. We evaluate the mixed-resolution multi-volume rendering algorithm in terms of run-time performance (Section 9.4) using six different data sets described in Section 9.3. We show how the decoupling of resolution levels and the residency octree's spatial subdivision is beneficial with an increasing number of channels in Section 9.5. The residency octree's optimized cache utilization in terms of smaller working sets compared to existing page table and octree-based approaches is shown in Section 9.6. The environment we use for evaluating our method, as well as two reference implementations we compare our method against are discussed in Section 9.1 and Section 9.2 respectively.

9.1 Evaluation Environment

We evaluate our method using the implementation discussed in Chapter 8 on a computer running Ubuntu 22.04.1 LTS. The computer has an AMD Ryzen 7 2700X CPU, 32 GB RAM, and a GeForce GTX 1080 as a dedicated GPU with 8 GB GPU memory. Since WebGPU is a browser API, we use version 108.0.5359.40 of the Chromium browser in our experiments. We enable the browser's experimental WebGPU support and use Chromium's Vulkan backend for dispatching WebGPU commands to the GPU. Furthermore, we enable the use of WebGPU's timestamp-query feature [MNJ23] in order to accurately time individual GPU operations by writing a timestamp in nanoseconds at the start and end of each compute pass to a buffer. This buffer's contents are then read back to the CPU once per frame. We enable the timestamp-query feature by allowing unsafe API calls in the Chromium browser.

9.2 Reference Implementations

We evaluate the residency octree and mixed-resolution multi-volume renderer by comparing it to two other approaches modeled after the current state of the art in out-of-core volume rendering as discussed in Chapter 2: (1) rendering with just a multi-channel page-table hierarchy as described in Chapter 5, and (2) an octree-based out-of-core renderer based on Crassin et al.'s [CNLE09] approach extended to multiple channels.

9.2.1 Multi-Channel Page-Table Hierarchy

The first approach uses the multi-channel page-table hierarchy to access cached volume data directly, instead of traversing an octree for each sample. It skips empty space only if the current bricks of all channels are empty. Since bricks are only marked as `EMPTY` if all of their voxels are zero instead of providing a transfer-function aware method for this purpose, this approach is expected to skip empty space only in rare cases. Furthermore, this approach does not substitute missing bricks by rendering the volume in other resolutions that may be resident in the brick cache. Instead, all samples that fall into the missing brick are skipped for the missing brick's channel.

For very dense volumes, this approach is expected to achieve higher or at least comparable frame rates when compared to the other two approaches, since the overhead of traversing a tree can not be compensated by skipping over empty space by the other methods. In terms of GPU memory utilization with respect to the number of bricks in the working set, this method is expected to perform worse than our method, because bricks containing at least one non-zero value are not considered `EMPTY` and thus have to be kept in memory even though they might be completely transparent under the current viewing conditions.

9.2.2 Octree

The second approach we compare our method to is based on the out-of-core renderer presented by Crassin et al. [CNLE09] and is similar to the octree-based method discussed by Brix et al. [BPH14]. We extend the single-channel approach by Crassin et al. [CNLE09] by storing in each node m pointers to the nodes' m corresponding bricks, where m is the number of visible channels. Additionally, we store culling metadata in the form of minimum and maximum scalar values for each channel in each of the nodes. We use the same culling data as the residency octree implementation in this approach but use a one-to-one mapping of bricks and octree nodes. Furthermore, in order to render parts of the volume in a high resolution, the whole subtree has to be resident in the cache. To enforce this, we restart the tree traversal for each visible channel when evaluating a ray sample. In case of a missing brick in the desired resolution, the next lower resolution is rendered instead.

Because whole subtrees have to be resident in the cache for rendering the volume in high resolution, the memory footprint in terms of the number of bricks in the working set is expected to be higher when compared to the residency octree. This approach is expected

to achieve frame rates comparable to our method for a single channel, since both use a similar empty-space skipping strategy. However, for multiple channels, the octree has to be traversed for each channel to ensure that all subtrees are resident in the cache. It is thus expected to perform worse than our method for multiple channels.

9.3 Data Sets

To evaluate our method, we use both single-channel data sets and biomedical data sets generated by Cyclic Immunofluorescence (CyCIF) [LIW+18] with a large number of channels. Their exact dimensions, number of resolution levels, number of channels, file sizes, as well as their data types are listed in Table 9.1. Renderings of the data sets used for evaluating our method are shown in Figure 9.1. We use the following three single-channel data sets in our evaluation:

- *Aneurism*: A scan of an aneurism within the arteries of a human head.
- *Bonsai*: A scan of a bonsai tree.
- *Kingsnake*: A scan of a Lampropeltis getula (common kingsnake) egg.

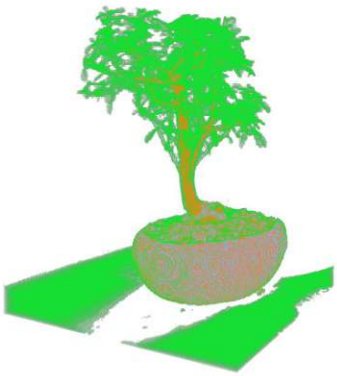
To evaluate multi-channel settings with single-channel data sets, we simulate multi-channel volumes with n channels by duplicating the volume n times. During rendering, we use different transfer functions for each channel. For the *Bonsai* data set, for example, we use different transfer functions for the value ranges corresponding to leaves and grass, the flower pot, and the bonsai's trunk and branches (Figure 9.1b). For the *Kingsnake* data set, we use transfer functions that highlight the snake's skeleton. Due to noise in the scan, there are almost no regions that contain only zero values.

We use three different biomedical data sets generated using CyCIF in our benchmarks. Each shows a human tissue sample containing cancer cells. The different channels in each data set show different elements within the tissue, e.g., DNA, certain types of cancer cells, or types of antibodies. The following biomedical data sets are used in our benchmarks:

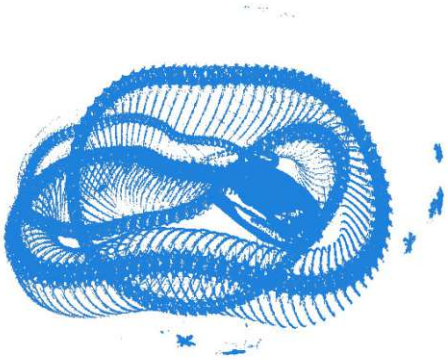
- *CyCIF Small* [NMV+22]: A scan of malignant melanoma in human skin tissue.
- *CyCIF Medium*: A scan of malignant melanoma in human skin tissue.
- *CyCIF Large*: A scan of cancer in human tonsil tissue.

To avoid issues related to slight inaccuracies in the physical positioning of tissue samples between imaging cycles in the imaging process, microscopes used in this process are often configured to record a region that is slightly larger than the actual tissue sample. This leads to completely empty image slices at the top and bottom of the image stack comprising the volume. In the *CyCIF Small* data set such empty regions at the top and

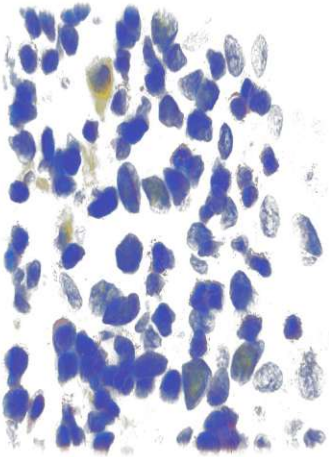
(a) Aneurism



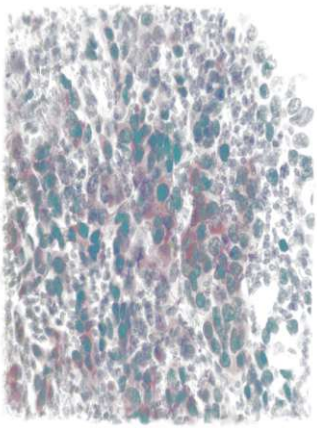
(b) Bonsai
(here with 4 simulated channels)



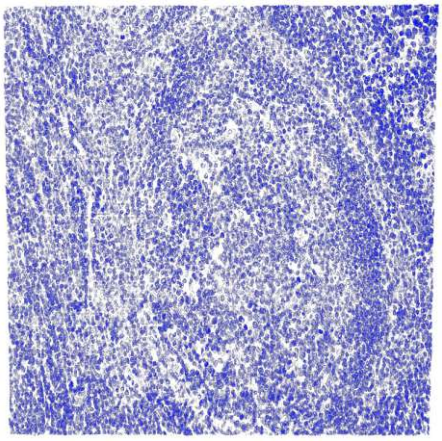
(c) Kingsnake



(d) CyCIF small



(e) CyCIF medium



(f) CyCIF large

Figure 9.1: **Data sets used for evaluation.**

bottom of the volume, i.e., along the volume's z-axis, have been removed in pre-processing. This has not been done for the other two CyCIF data sets, however.

9.4 Rendering Performance

To evaluate the rendering performance of our mixed-resolution multi-volume renderer, we measure the computation time of the compute pass performing the volume rendering using WebGPU's timestamp-query feature [MNJ23]. We measure the performance in ten-second intervals during which the camera is rotating around the center about the y-axis (pointing up), and average the results of ten iterations each.

To compare the performance of our method to the other two approaches (Section 9.2), we use the same down-sampling ratio and spatial subdivision for both the bricked multi-volume hierarchy and the residency octree. We use a cache size of 4 GB and a brick size (and approximate leaf-node size) of 32^3 voxels.

As to not give an unfair advantage to any of the methods, in most of our benchmarks most bricks visible are already resident in the cache. All single-channel data sets are tested with one and four channels each. The biomedical data sets, however, are tested with multiple channels. *CyCIF Small* [NMV+22] is tested with 16 visible channels. *CyCIF Medium* is tested with all its 15 channels being visible. The *CyCIF Large* data set is tested with four visible channels rendered at the second-highest resolution level. The reason for this is that keeping a channel of the *CyCIF Large* dataset requires roughly four gigabytes of memory in the highest resolution, and one gigabyte of memory in the next lower resolution level. This allows for only four channels to be resident in the cache at the same time.

As an optimization, we start residency octree and octree traversal at the third subdivision level counting from the root node. We also do not start tree traversal at the root level for each sample, but from the parent node's subdivision level in the tree structure of the last node visited for the last sample. Because we use a full octree in our implementation (Chapter 8), we can access the correct node for a ray sample in each subdivision level directly.

9.4.1 Results

The results of the benchmarks are shown in Table 9.1. Our method outperforms the other approaches in all benchmarks except for rendering the two small single-channel data sets, *Aneurism* and *Bonsai*. Rendering the *Aneurism* data set with a single channel, our method achieves the same frame rate as the octree reference implementation. The octree reference implementation is slightly faster in rendering the *Bonsai* data set with just a single channel. However, there is only a 0.1-millisecond difference between our method and the octree reference implementation. As discussed in Section 9.2, this similarity in rendering performance is expected with a single visible channel due to the similarity of the two methods with respect to empty-space skipping. However, the expected similarity

Table 9.1: **Performance evaluation** of our method for several data sets (Section 9.3). We list the general information about the data set and the results of our benchmarks. Note that the data size is the size of the uncompressed volume, not of the OME-Zarr data set we converted them to. We compare our method (Ours) against a multi-resolution multi-channel page-table (PT) only, and an octree-based approach (Oct.).

Dataset & Description	**Data Size and Type, # Resolution Levels**	**DVR Performance** Avg. ms / frame (num. channels)
Aneurism 256 × 256 × 256 Channels: 1 (simulated 4)	16.8 MB (8 bit) Resolution levels: 4	PT: 5.1 (1) / 15.0 (4) Oct.: **4.8** (1) / 15.8 (4) Ours: **4.8** (1) / **12.3** (4)
Bonsai 256 × 256 × 256 Channels: 1 (simulated 4)	16.8 MB (8 bit) Resolution levels: 4	PT: 4.7 (1) / 11.9 (4) Oct.: **4.5** (1) /6.7 (4) Ours: 4.6 (1) / **6.5** (4)
Kingsnake 1024 × 1024 × 795 Channels: 1 (simulated 4)	833.6 MB (8 bit) Resolution levels: 6	PT: 15.0 (1) / 42.5 (4) Oct.: 11.6 (1) / 60.7 (4) Ours: **5.1** (1) / **19.8** (4)
CyCIF Small 1024 × 1024 × 40 Channels: 29	2.4 GB (16 bit) Resolution levels: 5	PT: 34.0 (16) Oct.: 122.8 (16) Ours: **25.7** (16)
CyCIF Medium 1024 × 1024 × 105 Channels: 15	6.6 GB (16 bit) Resolution levels: 5	PT: 40.4 (15) Oct.: 155.1 (15) Ours: **37.8** (15)
CyCIF Large 5632 × 4352 × 160 Channels: 38	149 GB (8 bit) Resolution levels: 9	PT: 24.4 (4) Oct.: 123.5 (4) Ours: **22.4** (4)

between the two methods can not be observed in rendering the *Kingsnake* data set with just a single visible channel. In this case, our method is more than twice as fast as the octree approach.

While the page-table hierarchy approach achieves times similar to our method for rendering the *Aneurism* and *Bonsai* data sets with a single visible channel, this is not the case for the *Kingsnake* data set. This is due to a lot of space in this data set being transparent under the transfer function used, even though the actual voxel values are non-zero. In such cases, the limited empty-space skipping capabilities of page-table hierarchies become apparent.

The benefits of our multi-volume rendering algorithm in comparison to the octree approach are clearly visible when rendering data sets with multiple channels. While the difference is not large for the smaller datasets, *Aneurism* and *Bonsai*, the octree-based approach no

longer achieves interactive frame rates for all other data sets. Especially for the three biomedical data sets, *CyCIF Small*, *CyCIF Medium*, and *CyCIF Large*, this method takes more than 120 milliseconds to render a single frame. As expected, this approach suffers from its high memory footprint and from the computational cost of restarting tree traversal for each channel for multi-channel data sets.

The multi-channel page-table hierarchy performs significantly worse than the residency octree when rendering the *Aneurism*, *Bonsai*, and *Kingsnake* data sets. Again, the reason for this is that empty space can only be skipped if all values within a brick are zero. Since we duplicated the channels for these data sets to simulate multiple channels, the distribution of empty space is the same in all channels. The negative effects in the single-channel case are amplified with multiple channels. This is not the case for the CyCIF data sets, where the difference between our method and the page-table hierarchy approach is not as large, even though our method is clearly faster in all three cases. For these thin data sets, our empty-space skipping approach has less of an advantage over the page-table approach. The overhead of traversing an octree is not compensated as much as for cubical data by the little empty space that is skipped for these thin data sets.

9.5 Decoupling Resolution Levels from Spatial Subdivision

In Section 9.4, we evaluated the performance of our method when we use the same down-sampling ratio and spatial subdivision for both the bricked volume hierarchy and the residency octree, i.e., the tree structure was coupled to the resolutions in the data set. However, due to the conceptual decoupling of these two phenomena, the residency octree supports using a different granularity for empty-space skipping than the bricking granularity used for the data set. Thus the residency octree can achieve higher frame rates since efficient empty-space skipping typically requires a finer granularity than bricking as shown by Faludi et al. [FZZ+22].

To evaluate the rendering performance when using residency octrees with a spatial subdivision decoupled from the resolution levels in the data set, we employ a similar setup as in the benchmarks discussed in Section 9.4. We use a brick size of 32^3 voxels for all methods, ours, and the two reference approaches (Section 9.2). However, we use different approximate leaf-node sizes for the residency octree: 32^3, 16^3, 8^3. We also evaluate our method with an approximate leaf-node size of $8 \times 8 \times 4$ to show that the spatial subdivision used for a residency octree is arbitrary. This can be beneficial for anisotropic volumes as pointed out by Beyer et al. [BHP15].

We only use the *CyCIF Small* and *CyCIF Medium* data sets in this evaluation. We compare the performance for rendering the data sets with different numbers of visible channels in order to analyze how our method scales with respect to the number of channels. For the *CyCIF Small* data set, we use $m \in [1, 16]$ visible channels. For the *CyCIF Medium* data set, we use $m \in [1, 15]$ visible channels.

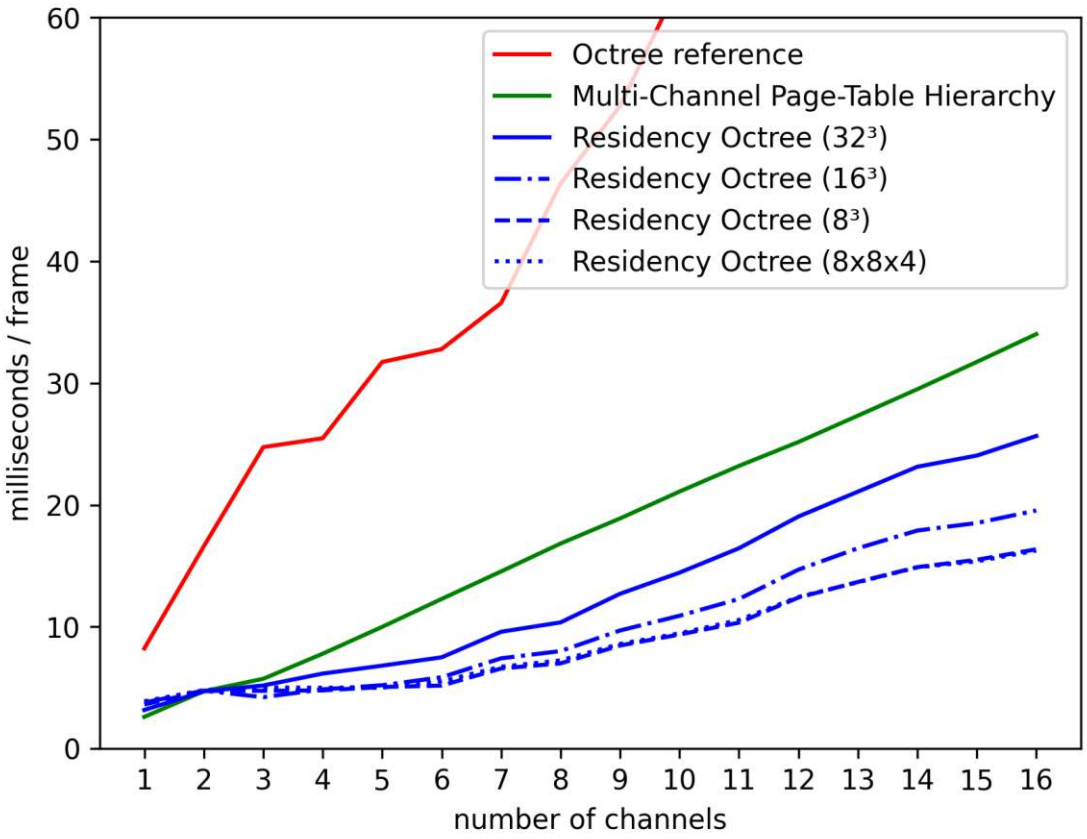


Figure 9.2: **Performance results - *CyCIF Small*.** Results of decoupling brick and leaf-node sizes for the *CyCIF Small* data set. Small residency-octree nodes mixed with larger brick sizes achieve the best performance.

9.5.1 Results

Figure 9.2 shows the results for the *CyCIF Small* data set using different numbers of spatial subdivisions and different numbers of visible channels. The data set is very thin and does not contain many empty bricks of size 32^3. Therefore, in this case, our method performs similarly to accessing cached volume data through the multi-channel page-table hierarchy directly when visualizing up to three channels. However, for larger numbers of channels, our method clearly outperforms the other two methods. Using a more fine-grained spatial subdivision for culling than the bricking granularity, the advantages of the residency octree become even more apparent. The residency octree also supports completely different spatial subdivision schemes that are better suited for anisotropic volumes, e.g., where a leaf node roughly corresponds to $8 \times 8 \times 4$ voxels as demonstrated in Figure 9.2.

As already shown in the previous benchmark (Section 9.4), the octree-based approach

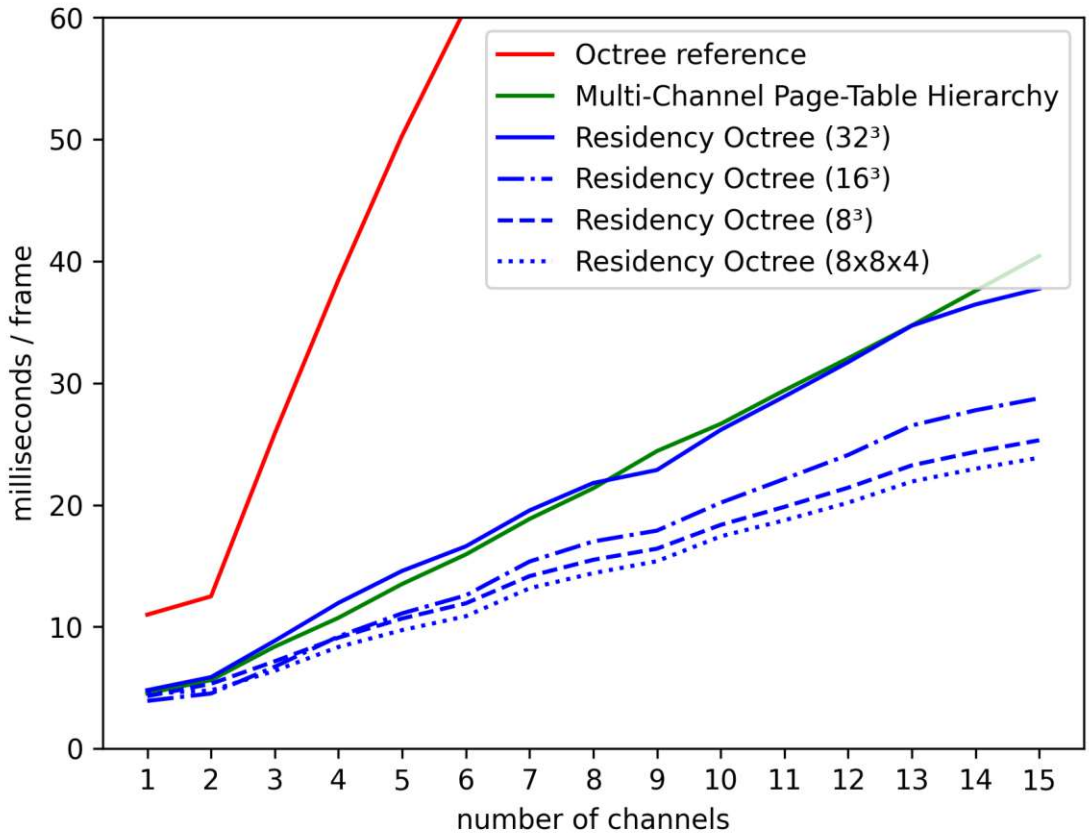


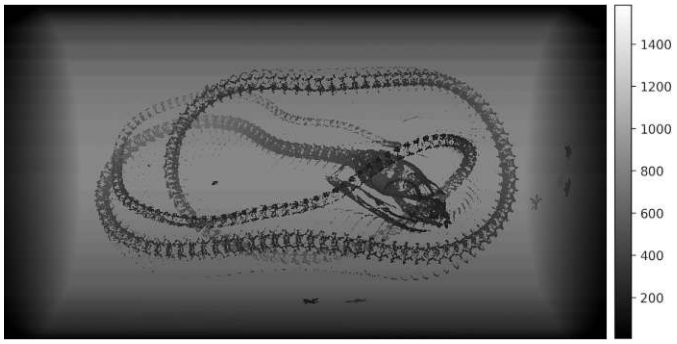
Figure 9.3: **Performance results - *CyCIF Medium*.** Results of decoupling brick and leaf-node sizes for the *CyCIF Medium* data set. Small residency-octree nodes mixed with larger brick sizes achieve the best performance.

suffers from the increased computational cost of restarting the tree traversal for each visible channel. Similar to our approach, the page-table hierarchy-based approach scales linearly with the number of visible channels.

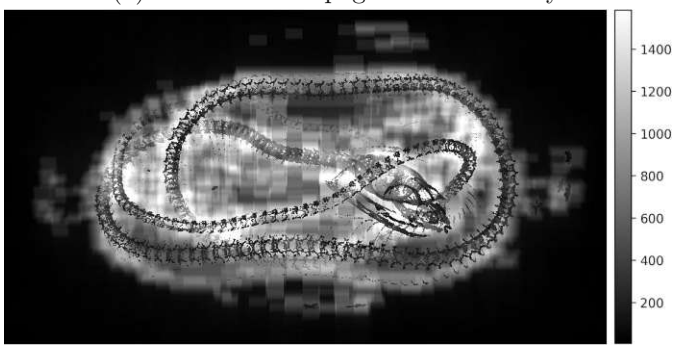
Figure 9.3 shows the results for the *CyCIF Medium* data set. While the results are similar to the ones for the *CyCIF Small* data set, the residency octree and the page-table hierarchy-based approach achieve similar frame rates if the residency octree's approximate leaf-node size is the same as the brick size. In this case, our method only performs better if the leaf-node size is decoupled from the brick size.

9.6 Working-Set Size

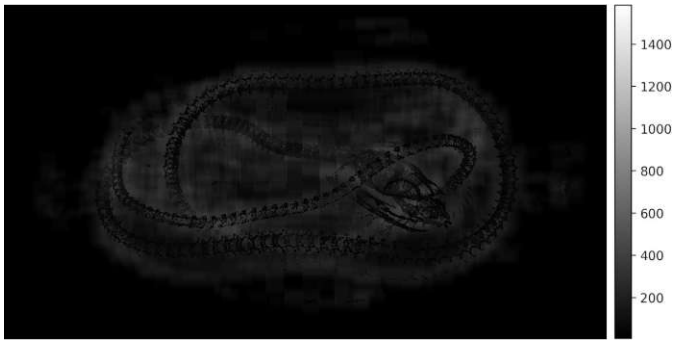
We evaluate the working-set size of all methods, i.e., the number of bricks they each require to be resident in the cache under the same viewing conditions, by visualizing the working-set size for each pixel. Since page-table hierarchies require all values in a



(a) Multi-channel page-table hierarchy



(b) Octree



(c) Residency octree

Figure 9.4: **Number of bricks required to be resident in the cache.** Brighter values indicate more bricks are required to be kept in memory for a pixel. Page-table hierarchies only have limited empty-space skipping capabilities. They access bricks that contain values that are outside of the currently visible range (a). Octrees require lower resolutions to be resident in the cache in order to render higher resolutions (b). The residency octree has neither of these limitations and only requires bricks that are visible under the current viewing conditions to be in the cache (c).

brick to be zero in order to skip it, they require bricks to be in the cache that contain non-zero values, even though they might not be visible with the current transfer function. For example, when rendering the *Kingsnake* data set, many transparent bricks near the image border are unnecessarily kept in memory, as shown in Figure 9.4a.

The residency octree supports transfer-function independent empty-space skipping. It only requires bricks that contain values in the visible range under the current viewing conditions to be resident in the cache. Octrees share this characteristic but require low-resolution bricks to be resident in the cache in order to render higher-resolution ones. In the case of the *Kingsnake* data set, many low-resolution bricks around the snake's skeleton are kept in the cache even though only high-resolution bricks actually contribute to the rendered image (Figure 9.4b). Because a residency octree's nodes are not tied to a single resolution, it does not have this limitation. This greatly reduces the number of bricks that are required to be resident in the cache for residency octrees in comparison to the two reference implementations as illustrated in Figure 9.4c.

CHAPTER 10

Discussion and Limitations

In this chapter, we discuss the advantages and disadvantages of our approach. Section 10.1 lists the limitations of the method and highlights aspects that could be improved in the future. The findings of the evaluation are discussed in Sections 10.2 and 10.4. The general advantages of our approach for multi-channel rendering in web-based contexts are discussed in Sections 10.3 and 10.4. Finally, in Section 10.5, we discuss the method's support for mixing different resolutions.

10.1 Limitations

The memory required for storing the residency octree scales linearly with the number of channels that can be visualized at the same time, because residency information is stored in the same way in each node for each channel. This is problematic for data sets with a large spatial extent and many channels to be visualized at the same time. Combining multiple channels into a single channel (e.g., by using dimensionality reduction techniques) could help avoid this issue. Additionally, for cache coherency, storing multiple channels in an interleaved manner, e.g., using a four-component texture format, might be beneficial.

The implementation uses minimum and maximum values for empty-space skipping. While this works well for unimodal transfer functions, a bitfield representation of a node's value range as proposed by Faludi et al. [FZZ+22] is better suited for multimodal transfer functions.

Furthermore, in the implementation, we currently let the user set the importance of each channel. We leave for future work investigating different methods to automatically determine the resolution range per channel and region in the volume based on its frequency content.

10.2 Residency Octrees Combine Advantages of Page-Tables and Octrees

Our method shares characteristics of existing out-of-core volume-rendering methods such as page tables and octrees, but combines their advantages while avoiding their disadvantages. Like page-table-based approaches, residency octrees support direct access to volume bricks of a desired resolution, instead of having to unnecessarily keep lower resolutions resident in the cache. However, page-table hierarchies have poor empty-space skipping capabilities, due to bricks of any resolution level being accessed directly instead of traversing a hierarchy. For example, a ray sample may fall into a small empty region in the volume. If the resolution level chosen for the sample is low and the brick accessed spans a larger, non-empty region, the small empty region is not skipped. Additionally, bricks are typically only flagged as empty if they only contain zero values to keep the memory requirements of page-table hierarchies low [HBJP12, FSK13, SCRL20]. Furthermore, page-table hierarchies lack a clear strategy for substituting missing high-resolution data with bricks from another resolution level. A missing brick can either not be substituted at all, or other resolutions have to be searched in the hope of finding one that is currently available instead. Residency octrees do not have these drawbacks. Since the spatial subdivision of a residency octree is independent of the resolution levels in the data set as well as the bricking granularity, they support more efficient empty-space skipping than page-table hierarchies as demonstrated in Section 9.5 As shown in Section 9.6, by storing transfer-function independent culling metadata in residency-octree nodes, bricks that are completely transparent under the current viewing conditions do not have to be kept in the cache. In case of cache misses, residency information stored in residency-octree nodes is used to substitute missing bricks with bricks from another resolution level. As discussed in Chapter 7, resolutions, for which the node is known to be partially mapped, can be directly queried using the bitmap stored in each node. This avoids unnecessary texture lookups for resolutions that are guaranteed to not be resident in the cache.

Similar to octree-based approaches, our tree data structure supports more efficient empty-space skipping than page tables. But by decoupling the resolutions in the data set from the spatial subdivisions determined by the tree, our approach allows for more fine-grained empty-space skipping than previous approaches do. At the same time, it is more flexible in terms of data-access patterns, producing smaller working sets than previous approaches.

10.3 Suitability for Web-Based Contexts

The system is designed for web-based environments where the data that is rendered may be provided by a third party. In such cases, the bricking granularity is not in the client-side renderer's control. For example, brick sizes may be large, e.g., 256^3, in order to keep the number of HTTP requests low on the server side. Since residency octrees decouple the bricking granularity of a data set from the spatial subdivision used for

empty-space skipping, they are not affected by these constraints. Instead, they may use any arbitrary spatial subdivision scheme that is best suited for the data set as shown in Section 9.5. As discussed in Section 10.2, this is not possible with previous approaches which are fully coupled to the brick size used for a data set.

10.4 Efficient Multi-Channel Rendering

Previous octree-based approaches either require traversing the tree structure for each sample and channel, or they have to check each channel at each tree traversal step. This is inefficient, especially for large numbers of channels, e.g., more than four. Page-table hierarchies, on the other hand, do not require a hierarchy traversal for each sample. Instead, each channel has to only be tested once per sample. However, as shown in Section 9.6 and discussed in Section 10.2, they exhibit only poor empty-space skipping capabilities due to being tied to the bricking granularity and being able to skip transparent bricks if they contain only zero values. As discussed in Chapter 7, the mixed-resolution multi-volume renderer traverses the octree only once for each sample. By doing this we minimize the performance cost of traversing a hierarchy when rendering multi-volume data as demonstrated in Sections 9.4 and 9.5. Because of this and the arbitrary spatial subdivision supported by residency octrees that can be optimized for each data set, our method is better suited for rendering volumes with large numbers of channels than previous approaches.

Furthermore, by virtualizing multiple channels our approach supports data sets with an arbitrary number of channels. Only the number of channels that can be visualized at the same time is constrained by the bit-width of the data type used for brick IDs as discussed in Section 5.4. This makes it feasible for streaming in multi-channel data from a remote server on demand, and efficiently switching out channels at run-time.

10.5 Flexible Mixing of Different Resolutions

In comparison to previous methods, residency octrees are more flexible in mixing different resolutions, while also skipping empty space in an efficient manner. Our method not only makes it possible to mix resolutions in rendering a single channel but also when rendering multiple channels. This makes it possible to optimize cache utilization by reducing the resolution for less important channels, or channels with less frequency content, and instead prioritizing channels that require higher quality rendering. Our implementation employs user-controlled upper and lower bounds of allowed resolution levels per channel to control importance. However, because our approach defines an abstract function to determine the importance of a channel, this could also be computed based on other parameters such as the current viewpoint, or the frequency content of an octree node.

CHAPTER 11

Conclusion and Future Work

In this chapter, we conclude the thesis by recapitulating our approach and findings and exploring potential avenues for future research.

11.1 Conclusion

In this work, we have presented the *Residency Octree*, a hybrid data structure combining page-table hierarchies and octrees, that is well-suited for client-side web-based out-of-core volume rendering of data sets with a large number of channels. The main outstanding characteristic of our approach is that the cache residency of multi-resolution data is decoupled from a resolution-independent spatial subdivision determined by the tree. Instead of being tied to a single brick of a single resolution, each residency octree node keeps track of the cache residency of a set of bricks in each resolution. This makes it possible to efficiently and adaptively choose and mix resolutions, adapt sampling rates, and compensate for cache misses by rendering other resolutions that are resident in the cache. At the same time, this decoupling allows residency octrees to support fine-grained empty-space skipping, independent of the data subdivision used for caching. Furthermore, residency octrees are constructed incrementally, i.e., subtrees that are never visible on screen are never constructed.

We have shown that residency octrees produce smaller working sets, i.e., require fewer bricks to be cache resident than previous approaches based on octrees or page-table hierarchies alone, and are thus scalable with respect to the size of a data set. Furthermore, we found that residency octrees allow for more efficient empty-space skipping than previous approaches. This is due to their support for spatial subdivisions that are more fine-grained than the bricking granularity in the data set. This characteristic of our data structure is also beneficial for scenarios where the bricking granularity can not be controlled by the rendering application, e.g., in web-based contexts.

We also presented a mixed-resolution multi-volume rendering algorithm to efficiently skip empty space when rendering multiple channels at once. This is achieved by only traversing the octree hierarchy once per sample instead of once per sample and channel. Our algorithm exploits the facts that multiple channels in a residency octree share the same spatial subdivision, and that all channels have to be transparent for a region to be skipped. Once the hierarchy has been traversed for one channel, processing subsequent channels can thus proceed by continuing the tree traversal from the same node, eliminating the need to restart the traversal from the root node.

We have shown that our approach allows for more efficient empty-space skipping than previous work and scales well with respect to the number of channels. On the other hand, due to the tight coupling of resolutions in the data set and the spatial subdivision of their tree structure, previous octree-based approaches require restarting the hierarchy traversal for each sample and channel. This scales poorly to large numbers of channels in a data set. Page-table hierarchies have only limited empty-space skipping capabilities in general. However, for very dense volumes where only a few regions are fully transparent, their simplicity may outweigh the cost of a hierarchy traversal for each sample.

In conclusion, residency octrees and our mixed-resolution multi-volume rendering algorithm are more efficient than previous approaches for volumetric data sets with a large number of visible channels, especially when optimizing the tree's subdivision for empty-space skipping. Their support for arbitrary spatial subdivisions decoupled from the bricking granularity dictated by the data set, makes residency octrees more suitable for web-based, client-side rendering. This is especially beneficial if the data set belongs to a third party and the brick size can not necessarily be expected to be tailored to the needs of a renderer. Furthermore, residency octrees require fewer bricks to be resident in the cache than previous approaches and optimize cache usage for large-scale rendering.

11.2 Future Work

We have shown that residency octrees and mixed-resolution multi-volume rendering offer an effective and scalable approach to web-based volume rendering for data sets with an arbitrary number of channels. The ability to mix different resolutions on a per-sample and a per-channel basis opens up promising new directions for further research. In the future, we intend to experiment with adaptively choosing the resolution for rendering a channel based on measures other than taking only a user-controlled importance into account. For example, a channel's frequency content could be considered to determine bounds for the appropriate resolution levels and corresponding sampling frequencies for rendering. Doing this not only globally for the whole channel, but taking into account the frequency content of individual octree nodes, i.e., in the region in the volume represented by the node, could further optimize cache utilization by producing even smaller working sets than our current approach. We also intend to experiment with other spatial subdivision schemes to exploit the benefits of decoupling data resolutions and subdivisions further. Moreover, exploring novel methods for representing volume data and efficiently combining multiple

channels in both the brick cache and octree nodes could be beneficial in optimizing cache coherency during rendering.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ACA+19] Felix-Constantin Adochiei, Radu-Ion Ciucu, Ioana-Raluca Adochiei, Sorin Dan Grigorescu, George Călin Seriţan, and Miron Casian. A WEB Platform for Rendring and Viewing MRI Volumes using Real-Time Ray-tracing Principles. In *Proceedings of the 11th International Symposium on Advanced Topics in Electrical Engineering (ATEE)*, pages 1–4, March 2019. ISSN: 2159-3604.

[AM00] Ulf Assarsson and Tomas Möller. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, 5(1):9–22, January 2000. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/10867651.2000.10487517.

[AMBGA19] Ander Arbelaiz, Aitor Moreno, Iñigo Barandiaran, and Alejandro García-Alonso. Progressive Ray-casting Volume Rendering with WebGL for Visual Assessment of Air Void Distribution in Quality Control. In *Proceedings of the 24th International Conference on 3D Web Technology*, Web3D '19, pages 1–8, New York, NY, USA, July 2019. Association for Computing Machinery.

[BCH12] E. Wes Bethel, Hank Childs, and Charles Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, October 2012. Google-Books-ID: 0zPOBQAAQBAJ.

[BHP15] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. State-of-the-Art in GPU-Based Large-Scale Volume Visualization: GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum*, 34(8):13–37, December 2015.

[BHS+11] Johanna Beyer, Markus Hadwiger, Jens Schneider, Won-Ki Jeong, and Hanspeter Pfister. Distributed Terascale Volume Visualization using Distributed Shared Virtual Memory. In *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 127–128, October 2011.

[BHWB07] Johanna Beyer, Markus Hadwiger, Stefan Wolfsberger, and Katja Bühler. High-Quality Multimodal Volume Rendering for Preoperative Planning of Neurosurgical Interventions. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1696–1703, November 2007.

[BLL+19] Sébastien Besson, Roger Leigh, Melissa Linkert, Chris Allan, Jean-Marie Burel, Mark Carroll, David Gault, Riad Gozim, Simon Li, Dominik Lindner, Josh Moore, Will Moore, Petr Walczysko, Frances Wong, and Jason R. Swedlow. Bringing Open Data to Whole Slide Imaging. In Constantino Carlos Reyes-Aldasoro, Andrew Janowczyk, Mitko Veta, Peter Bankhead, and Korsuk Sirinukunwattana, editors, *Digital Pathology*, Lecture Notes in Computer Science, pages 3–10, Cham, 2019. Springer International Publishing.

[BPH14] Tobias Brix, Jörg-Stefan Praßni, and Klaus Hinrichs. Visualization of Large Volumetric Multi-Channel Microscopy Data Streams on Standard PCs, July 2014. arXiv:1407.2074 [cs].

[CNC13] Lazaro Campoalegre, Isabel Navazo, and Pere Brunet Crosa. Gradient Octrees: A New Scheme for Remote Interactive Exploration of Volume Models. In *2013 International Conference on Computer-Aided Design and Computer Graphics*, pages 306–313, November 2013.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. GigaVoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, February 2009. Association for Computing Machinery.

[CSK+11] John Congote, Alvaro Segura, Luis Kabongo, Aitor Moreno, Jorge Posada, and Oscar Ruiz. Interactive Visualization of Volumetric Data with WebGL in Real-time. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pages 137–146, New York, NY, USA, June 2011. Association for Computing Machinery.

[DGBNV18] Jesús Díaz-García, Pere Brunet, Isabel Navazo, and Pere-Pau Vázquez. Progressive Ray Casting for Volumetric Models on Mobile Devices. *Computers & Graphics*, 73:1–16, June 2018.

[DK19] Lachlan Deakin and Mark Knackstedt. Accelerated Volume Rendering with Chebyshev Distance Maps. In *Proceedings of the SIGGRAPH Asia 2019 Technical Briefs*, SA '19, pages 25–28, New York, NY, USA, November 2019. Association for Computing Machinery.

[DK20] Lachlan J. Deakin and Mark A. Knackstedt. Efficient Ray Casting of Volumetric Images using Distance Maps for Empty Space Skipping. *Computational Visual Media*, 6(1):53–63, March 2020.

[DLJL22] Dominik Drees, Simon Leistikow, Xiaoyi Jiang, and Lars Linsen. Voreen – An Open-source Framework for Interactive Visualization and Processing of Large Volume Data, July 2022. arXiv:2207.12746 [cs].

[FCS+10] Thomas Fogal, Hank Childs, Siddharth Shankar, Jens Krüger, R. Daniel Bergeron, and Philip Hatcher. Large Data Visualization on Distributed Memory Multi-GPU Clusters. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 57–66, Goslar, DEU, June 2010. Eurographics Association.

[FSK13] Thomas Fogal, Alexander Schiewe, and Jens Krüger. An Analysis of Scalable GPU-based Ray-guided Volume Rendering. In *Proceedings of the 2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pages 43–51, October 2013.

[FW20] I.B. Fernandes and M. Walter. A Bucket LBVH Construction and Traversal Algorithm for Volumetric Sparse Data. In *Proceedings of the 33rd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 39–45, November 2020. ISSN: 2377-5416.

[FZZ+22] Balázs Faludi, Norbert Zentai, Marek Zelechowski, Azhar Zam, Georg Rauter, Mathias Griessen, and Philippe C. Cattin. Transfer-Function-Independent Acceleration Structure for Volume Rendering in Virtual Reality. In *Proceedings of the Conference on High-Performance Graphics*, HPG '21, pages 1–10, Goslar, DEU, 2022. Eurographics Association.

[GHSK03] Jinzhu Gao, Jian Huang, Han-Wei Shen, and James Arthur Kohl. Visibility Culling using Plenoptic Opacity Functions for Large Volume Visualization. In *IEEE Visualization, 2003. VIS 2003.*, pages 341–348, October 2003.

[Goo22] Google. Neuroglancer: WebGL-Based Viewer for Volumetric Data, August 2022. https://github.com/google/neuroglancer, accessed: 2022-09-11.

[GSHK04] Jinzhu Gao, Han-Wei Shen, Jian Huang, and James Arthur Kohl. Visibility Culling for Time-Varying Volume Rendering using Temporal Occlusion Coherence. In *IEEE Visualization 2004*, pages 147–154, October 2004.

[HAAB+18] Markus Hadwiger, Ali K. Al-Awami, Johanna Beyer, Marco Agus, and Hanspeter Pfister. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):974–983, January 2018.

[HBJP12] Markus Hadwiger, Johanna Beyer, Won-Ki Jeong, and Hanspeter Pfister. Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, December 2012.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[JKW+22] Jared Jessup, Robert Krueger, Simon Warchol, John Hoffer, Jeremy Muhlich, Cecily C. Ritch, Giorgio Gaglia, Shannon Coy, Yu-An Chen, Jia-Ren Lin, Sandro Santagata, Peter K. Sorger, and Hanspeter Pfister. Scope2Screen: Focus+Context Techniques for Pathology Tumor Assessment in Multivariate Image Data. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):259–269, January 2022.

[KBJ+20] Robert Krueger, Johanna Beyer, Won-Dong Jang, Nam Wook Kim, Artem Sokolov, Peter K. Sorger, and Hanspeter Pfister. Facetto: Combining Unsupervised and Supervised Learning for Hierarchical Phenotype Analysis in Multi-Channel Image Data. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):227–237, January 2020.

[LCDP13] Baoquan Liu, Gordon J. Clapworthy, Feng Dong, and Edmond C. Prakash. Octree Rasterization: Accelerating High-Quality Out-of-Core GPU Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(10):1732–1745, October 2013.

[LIW+18] Jia-Ren Lin, Benjamin Izar, Shu Wang, Clarence Yapp, Shaolin Mei, Parin M. Shah, Sandro Santagata, and Peter K. Sorger. Highly Multiplexed Immunofluorescence Imaging of Human Tissues and Tumors using t-CyCIF and Conventional Optical Microscopes. *eLife*, 7:e31657, July 2018. Publisher: eLife Sciences Publications, Ltd.

[LMK03] Wei Li, Klaus Mueller, and Arie Kaufman. Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *IEEE Visualization, 2003. VIS 2003.*, pages 317–324, October 2003.

[LWC+23] Jia-Ren Lin, Shu Wang, Shannon Coy, Yu-An Chen, Clarence Yapp, Madison Tyler, Maulik K. Nariya, Cody N. Heiser, Ken S. Lau, Sandro Santagata, and Peter K. Sorger. Multiplexed 3D Atlas of State Transitions and Immune Interaction in Colorectal Cancer. *Cell*, 186(2):363–381.e19, January 2023.

[MAB+21] Josh Moore, Chris Allan, Sébastien Besson, Jean-Marie Burel, Erin Diel, David Gault, Kevin Kozlowski, Dominik Lindner, Melissa Linkert, Trevor Manz, Will Moore, Constantin Pape, Christian Tischer, and Jason R. Swedlow. OME-NGFF: A Next-generation File Format for Expanding Bioimaging Data-access Strategies. *Nature Methods*, 18(12):1496–1498, December 2021.

[MF12] Movania Muhammad Mobeen and Lin Feng. High-Performance Volume Rendering on the Ubiquitous WebGL Platform. In *Proceedings of the 2012*

IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, pages 381–388, June 2012.

[MGP+22] Trevor Manz, Ilan Gold, Nathan Heath Patterson, Chuck McCallum, Mark S. Keller, Bruce W. Herr, Katy Börner, Jeffrey M. Spraggins, and Nils Gehlenborg. Viv: Multiscale Visualization of High-resolution Multiplexed Bioimaging Data on the Web. *Nature Methods*, 19(5):515–516, May 2022.

[MKRE18] Finian Mwalongo, Michael Krone, Guido Reina, and Thomas Ertl. Web-based Volume Rendering using Progressive Importance-based Data Transfer. In *Proceedings of the Conference on Vision, Modeling, and Visualization*, EG VMV '18, pages 147–154, Goslar, DEU, 2018. Eurographics Association.

[MM10] Stéphane Marchesin and Kwan-Liu Ma. Cross-Node Occlusion in Sort-Last Volume Rendering. In *Proceedings of the 10th Eurographics conference on Parallel Graphics and Visualization*, EG PGV'10, pages 11–18, Goslar, DEU, 2010. Eurographics Association.

[MNJ23] Myles Maxfield, Kai Ninomiya, and Brandon Jones. WebGPU. W3C Working Draft, W3C, March 2023. https://www.w3.org/TR/2022/WD-webgpu-20220908, accessed: 2023-03-02.

[NMV+22] Ajit J. Nirmal, Zoltan Maliga, Tuulia Vallius, Brian Quattrochi, Alyce A. Chen, Connor A. Jacobson, Roxanne J. Pelletier, Clarence Yapp, Raquel Arias-Camison, Yu-An Chen, Christine G. Lian, George F. Murphy, Sandro Santagata, and Peter K. Sorger. The Spatial Landscape of Progression and Immunoediting in Primary Melanoma at Single-Cell Resolution. *Cancer Discovery*, 12(6):1518–1541, June 2022.

[QCZ+17] Liang Qiao, Xin Chen, Ye Zhang, Jingna Zhang, Yi Wu, Ying Li, Xuemei Mo, Wei Chen, Bing Xie, and Mingguo Qiu. An HTML5-Based Pure Website Solution for Rapidly Viewing and Processing Large-Scale 3D Medical Volume Reconstruction on Mobile Internet. *International Journal of Telemedicine and Applications*, 2017:e4074137, May 2017. Publisher: Hindawi.

[RCH+22] Rumana Rashid, Yu-An Chen, John Hoffer, Jeremy L. Muhlich, Jia-Ren Lin, Robert Krueger, Hanspeter Pfister, Richard Mitchell, Sandro Santagata, and Peter K. Sorger. Narrative Online Guides for the Interpretation of Digital-pathology Images and Tissue-atlas Data. *Nature Biomedical Engineering*, 6(5):515–526, May 2022.

[RHH17] Mohammad Raji, Alok Hota, and Jian Huang. Scalable Web-embedded Volume Rendering. In *Proceedings of the 7th IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 45–54, October 2017.

[Ros19] Andreas Rossberg. WebAssembly Core Specification. Technical report, W3C, December 2019. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf, accessed:2023-03-02.

[SCRL20] Jonathan Sarton, Nicolas Courilleau, Yannick Remion, and Laurent Lucas. Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-of-Core Approach. *IEEE Transactions on Visualization and Computer Graphics*, 26(10):3008–3021, October 2020.

[SRL19] Jonathan Sarton, Yannick Remion, and Laurent Lucas. Distributed Out-of-Core Approach for In-Situ Volume Rendering of Massive Dataset. In Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode, editors, *High Performance Computing*, pages 623–633, Cham, 2019. Springer International Publishing.

[SS11] Nicole Schubert and Ingrid Scholl. Comparing GPU-based Multi-volume Ray Casting Techniques. *Computer Science - Research and Development*, 26(1):39–50, February 2011.

[UP20] Will Usher and Valerio Pascucci. Interactive Visualization of Terascale Data in the Browser: Fact or Fiction? In *Proceedings of the 10th IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 27–36, October 2020.

[WBD+21] Jiamin Wang, Chongke Bi, Liang Deng, Fang Wang, Yang Liu, and Yueqing Wang. A Composition-free Parallel Volume Rendering Method. *Journal of Visualization*, 24(3):531–544, June 2021.

[WRT15] Kongyot Wangkaoom, Paruj Ratanaworabhan, and Saowapak S. Thongvigitmanee. High-quality Web-based Volume Rendering in Real-time. In *Proceedings of the 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 1–6, June 2015.

[XZLK19] Jian Xue, Xiaoye Zhu, Ke Lu, and Yutong Kou. Parallel Volume Rendering Method for Out-of-Core Non-Uniformly Partitioned Datasets. In *Proceedings of the 2019 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, pages 599–602, July 2019.

[YSG15] Yeonsoo Yang, Ankit Sharma, and Armand Girier. Volumetric Texture Data Compression Scheme for Transmission. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 65–68, New York, NY, USA, June 2015. Association for Computing Machinery.

[ZHL19] Stefan Zellmann, Matthias Hellmann, and Ulrich Lang. A Linear Time BVH Construction Algorithm for Sparse Volumes. In *Proceedings of the*

2019 IEEE Pacific Visualization Symposium (PacificVis), pages 222–226, April 2019. ISSN: 2165-8773.

[ZML19] Stefan Zellmann, Deborah Meurer, and Ulrich Lang. Hybrid Grids for Sparse Volume Rendering. In *Proceedings of the IEEE Conference on Visualization 2019*, pages 1–5, October 2019.

[ZSL18] Stefan Zellmann, Jürgen P. Schulze, and Ulrich Lang. Rapid k-d Tree Construction for Sparse Volume Data. In *Proceedings of the Symposium on Parallel Graphics and Visualization*, EGPGV '18, pages 69–77, Goslar, DEU, June 2018. Eurographics Association.

[ZSL21] Stefan Zellmann, Jürgen P. Schulze, and Ulrich Lang. Binned k-d Tree Construction for Sparse Volume Data on Multi-Core and GPU Systems. *IEEE Transactions on Visualization and Computer Graphics*, 27(3):1904–1915, March 2021.