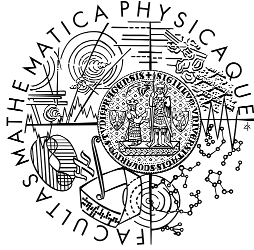


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Alisher Kenzhebayev

Vectorized traversal of sparse volumes for GPU path tracing

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Tomáš Iser, Ph.D.
Advisor of the master thesis: Tobias Rittig, Ph.D.
Study programme: Computer Science - Visual Computing and Game Development (IVVPA)

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I want to dedicate this work to my mother, who has been supporting me through this work, and thank her for showing never-ending love and care.

I strongly want to express my endless gratitude and a huge word of thanks to my Thesis Advisor, Tobias Rittig, Ph.D., for his persistence and constant help in the form of valuable advice throughout the implementation and making of this work.

I also want to express a huge word of thanks to my supervisor, Mgr. Tomáš Iser, Ph.D., for hours of advice given over our weekly calls and also for constant help with code implementation issues.

Title: Vectorized traversal of sparse volumes for GPU path tracing

Author: Alisher Kenzhebayev

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Tomáš Iser, Ph.D., Department of Software and Computer Science Education

Advisor: Tobias Rittig, Ph.D., Additive Appearance s.r.o.

Abstract: In this thesis we explore the use of dense and sparse data structures for the representation of voxelized data and their features to develop a NanoVDB format-compatible, vectorized renderer plugin with native GPU support. To that end, we provide a comparison of NanoVDB against the main features of relevant data structures, including justification for the degree of suitability for their use on GPU. This thesis's contribution is the addition of the C++ standard volumetric plugin for the Mitsuba 3 path tracer with added NanoVDB format support, allowing sparse voxel data to be used. The obtained comparison shows significant improvements on several backends using memory and render speed benchmarks, without any significant quality loss on the resulting images.

Keywords: GPU, path tracing, data structures, participating media, sparse volume textures

Contents

Introduction **7**

1 Background **9**
1.1 Overview . . . 9
1.2 Data structures . . . 9
1.2.1 Regular grids . . . 10
1.2.2 Octrees . . . 11
1.2.3 OpenVDB . . . 12
1.2.4 NanoVDB . . . 15
1.3 Renderers . . . 15
1.3.1 Mitsuba2 . . . 15
1.3.2 DrJit . . . 16
1.3.3 Mitsuba3 . . . 17

2 Methodology **18**
2.1 Work outline . . . 18
2.2 Comparing VDB structures and Dense grid . . . 19
2.3 Mitsuba 3 plugin implementation . . . 21
2.3.1 Accessing VDB values . . . 21
2.3.2 Rewriting the NanoVDB to DrJIT . . . 21
2.3.3 Framework unit testing . . . 22
2.4 Verification . . . 22
2.4.1 VDB format conversion . . . 23
2.4.2 Baseline images . . . 24
2.4.3 Setting up a comparison scene . . . 25
2.4.4 Implemented plugin render . . . 25
2.4.5 Performance and Benchmarks . . . 26

3 Implementation details **27**
3.1 Our setup . . . 27
3.2 Scene descriptors . . . 28
3.3 NanoVDB conversion . . . 30
3.4 Plugin implementation . . . 31
3.5 Unit testing . . . 33
3.6 Integration to Mitsuba . . . 35
3.7 Rendering and Performance . . . 38
3.8 Packaging to Docker . . . 38

4 Results **39**
4.1 Test scenarios . . . 39
4.2 Case comparison . . . 43
4.3 Benchmarks . . . 43
4.4 CUDA image comparison . . . 51

Discussion **53**

Conclusion **54**

Bibliography **55**

List of Figures **57**

List of Tables **58**

A Attachments **59**

A.1 Electronic Project Attachment 59

A.2 User Documentation 59

A.3 Prerequisites Versioning 59

Introduction

The vast majority of visuals and image productions had at some point started incorporating voxel data as well as 3D textures into their work pipeline as an industry standard. Feature use of volumetric textures can usually be illustrated by their use in a wide variety of cases. That is, from visual effects that utilize dynamic temporal data for volume, smoke, or fluid simulations, to scientific computer-assisted medical imagery.

The reasoning behind their use is twofold, one being the switch from the rasterization pipeline to the physically based light transport, or in other words, path tracing [1]. Additionally, voxelized representation also allows for a better definition of the object's internal structure as compared to hard-surface modeling. By enabling a move away from the standardized shader-based render pipelines, they simplify both in-media and surface light calculation, as it is handled by physically based rendering instead. As a result of their use, volumetric representation allows for an easier setup phase for industrial rendering by removing the need for the presence of scene and shader-based tweaking iterations in productions. However, despite introducing improvements to the rendering pipeline by removing the shader-based global illumination and light scene tweaking iterations, this use of volumetric representation is not without its drawbacks, the main one being its memory footprint [2].

Memory footprint as an issue Traditionally, the representation of volume data, or in some cases also called volume textures, has been done in the form of uniform regular 3D grids, being simple to use and easy to access/modify. However, several major issues with regular data grids spurred the movement towards the use of sparse data structures. The main reason is the badly scaling memory requirement of popularized dense grids, as they scale proportionally with the volume of their embedded space.

This issue of increased memory requirement for volume representations is exacerbated further by the use cases of voxels in even small or moderate-sized numeric volume or fluid simulations. Animated volumes add changes in an object's 3D shape as a time-varying factor as well as a possibility of shifting the domain of the grid. Both result in an increase in the feature demands in the data structures that encode these animated voxels. While generally, it is possible to represent such cases with simple dense grids, the combination of memory inefficiency as well as the sheer amount of information either limits the resolution of embedded space or raises a question of data packing efficiency. Which asks for the use of sparse grid representation as a potential solution to this memory efficiency struggle.

The authors of OpenVDB claim VDB not to be a ”silver bullet” to all of the existing structure and memory issues of spatial data representation. Their claims aside, the OpenVDB is still an easily customizable data structure with a high degree of flexibility that allows for the rendering of spatial grids with an almost infinite spatial domain.

As the dense data structures tend to struggle with memory efficiency, the OpenVDB's hierarchical topology supposedly solves that. The proposed VDB data structure solves some of the issues found in these simple sparse data structures

for the representation of 3D textures by introducing a hierarchical topology and encoding most of the information using a construction-time fixed tree structure. The only downside to it is that it, just like many other sparse data structures, utilizes pointers for added flexibility, making it hard to use natively on GPU due to SIMD-like instructions requiring sequential block memory access for a GPU-side parallelized set of instructions.

NanoVDB as GPU optimized sparse data structure Despite the path tracing being possible to be modeled on a CPU environment, it is, without doubt, a slower approach than vectorized GPU or optimized SIMD, like ones using LLVM for cases of high-performance computing. Unfortunately, despite both the popularity of OpenVDB and its relative success as well as the widespreadness of the volume representation in the industry, most approaches to GPU rendering still utilize the intermediate representation grid format. That intermediate format is usually in the form of dense grids, which, as it was introduced, have poor memory efficiency and thus badly scale in their applications to GPU-side rendering. With recent developments, a new variant of NVIDIA-developed OpenVDB compatible structure was added, called NanoVDB. It is proposed as a fast and optimized solution with a smaller memory footprint compared to OpenVDB, albeit with reduced functionality. The goal of this thesis is the integration of this improved GPU-ready sparse volumetric data structure into a vectorized JIT-compiled research renderer of Mitsuba 3. To that end, this thesis aims to provide a prototype implementation of the named NanoVDB grid plugin as a CUDA and LLVM natively supporting solution due to its integration into a templated and vectorized DrJIT + Mitsuba project.

Outline Structurally the thesis is split into four chapters, with additional space for discussion and conclusion. Starting with the background chapter 1, our thesis attempts to illustrate the problem with memory efficiency and builds a case of VDB as a solution to these issues. Additionally, this thesis goes into some detail to provide the reader with sufficient background linked to the problem of volume representation using previous variants of data structures used for voxels. To achieve that, some of the known data structures are compared by their relevant features. Separately, a high-level description of their memory efficiency is briefly added. However, as the main goal of this thesis is to provide access to the sparse volumetric data within a NanoVDB format through the development of a prototype Mitsuba 3 volume plugin, it is added in chapter 3. To achieve it, the results chapter 4 shows that the reached performance values are comparable to the results of existing dense grids rendered with the simple Mitsuba volume grid plugin. To illustrate that, a quantitative comparison of produced path-traced images is conducted, utilizing performance benchmarks on speed and accuracy. For accuracy, a set of measurements is conducted with several image similarity methods as well as attached benchmarks of the rendering time and memory.

Furthermore, the Discussion 4.4 and Conclusion chapters follow the main content by reiterating the findings. These also include some of the peculiarities that were of interest or deserve a mention. The discussion mainly describes some of the challenges that were encountered in the process of implementation as well as some of the shortcomings.

1 Background

This chapter aims to give the reader a brief description of concepts relevant to the thesis in the following chapters. Sections focus on data structures used and provide a brief overview of differentiable rendering in Mitsuba using Enoki/DrJIT.

1.1 Overview

The motivation of this thesis is to allow vectorized access to multiple graphics APIs for sparse volumetric data, specifically targeting a Mitsuba 3 research renderer. The end goal of this work is the production of a deliverable volumetric plugin compatible with the aforementioned research renderer that would be able to process files of the NanoVDB extension.

To that end, this chapter describes the nature of the memory consumption problem with the use of dense grid representations and their memory bottleneck. The goal of this chapter is to describe relevant data structures that were or are currently used in the industry for the representation of volumetric data. Some of these include regular uniform grids, Sparse Voxel Octrees (SVO), as well as other kinds of sparse data structures.

However, not all sparse structures are well-suited to represent huge amounts of data, be it in the context of rendering an object with a big resolution or a moderate-sized animated volume. Another good example of this fact is the results achieved by a sparse data structure, octrees in Meagher, in which the downside to their approach was also emphasized as its inefficient memory utilization. In this case, it is because the memory footprint is directly proportional to the surface area of the enclosed object [3].

OpenVDB's claim supports that because, to the best of their knowledge, none of the dense nor simple octree representations were successfully used for animations or simulations.

Actualizing this information in this chapter also serves to illustrate why the use of sparse data structures becomes the norm as well as the factual industry standard, especially with the introduction of Sparse Voxel Octrees by Nvidia [4] as well as open source OpenVDB by Ken Museth in 2013.

Later sections introduce the main focus of this thesis, sparse structures known as OpenVDB and NanoVDB. Both of them are introduced briefly and on a surface level, covering most of the relevant features. Separately, our Renderers section 1.3 describes DrJIT, a Just-In-Time Compiler for Differentiable Rendering as well as two versions of a Mitsuba renderer. Paragraphs inside it emphasize the modularity of Mitsuba, acting as a gradual introduction to the renderer's architecture being a highly modifiable plugin-based solution.

1.2 Data structures

This section explores and provides a descriptive case of relevant data structures, starting with regular grids and one of the first sparse data structures, octrees.

Additionally, other included subsections provide descriptions of OpenVDB features and how NanoVDB iterates on OpenVDB as its successor.

1.2.1 Regular grids

Probably, the simplest and most uncompressed version of the volume representation that can be easily perceived is the Cartesian Coordinate (CC) regular grid, analogous to a 3-dimensional array with index order *ijk* in programming. It has major Cartesian axes X, Y, and Z, most of which assume the mapping of each equally spaced, discrete coordinate value in a sorted XYZ order to a separate voxel value. However, the resolution may be different in distinct cases [5] [6]. While the approach is widely known and used, it is also recognized to be quite inefficient in terms of memory usage because it lacks proper compression. Because the data is uncompressed, various approaches prefer another version, a body-centered cubic grid (BCC), as a way to save around 30% in memory due to sampling specifics. Despite that, and also due to the GPU's inability to natively generalize to the BCC grid's linear interpolation, most approaches still continue widely using the CC regular grids as a favored representation format. [7]

Utilization of regular uniform 3D grids in that uncompressed form clearly has several advantages and downsides. One of the more prominent benefits of using such a structure for volumes is the speed and simplicity of random memory access. The downsides, however, include a huge memory footprint that poorly scales in comparison to sparse data structures such as octrees 1.2.2, hierarchical OpenVDB 1.2.3, or its successor NanoVDB 1.2.4. As a very extreme example of this inefficiency, a sparse data structure like OpenVDB allows for a grid domain to be virtually infinite. In contrast, the regular dense grid has some strict grid domain size limits. That is due to the memory footprint scaling proportionally to the volume of the embedding domain. Additionally, as a result of the gap between processing power and memory performance steadily growing for several decades now, this is also becoming an issue in the use of dense grids for more sizeable volumetric data [8].

Evidently, for that to be an immediate concern with rendering, even in the case of regular dense grids, the object in question has to be of sufficient grid size. One of the most well-known examples of such data would be a Disney cloud dataset, around 2.7 GB in the form of sparse OpenVDB format. During conversion from dense grids to a sparse data structure, they are typically enclosed in an approximated bounding box with floating-point values that, during conversion, require a cutoff tolerance parameter. This is done to indicate the active voxels for sparse representations because of optimizations that come from the hierarchical data structures in their topology. When converting such clouds from sparse to dense, however, there is no such cutoff, as the data is either way copied into the densely allocated space. That space is most likely serialized in memory, as opposed to indirection through pointers used in most CPU sparse structures. As such, it is hard to estimate the exact location of each voxel for such a float cloud, making it challenging to calculate its accurate size in the precise dense voxelized equivalent. Significantly, because the resulting data is typically subjected to transfers over the CPU-GPU bus, in the case of GPU-side renders, the use of dense grid representation would usually result in a memory bottleneck due to their

enormous size [9]. Thus, despite us not knowing the exact amount of memory its representation would take in the form of a dense grid, that still makes the memory efficiency argument conceptually quite simple.

Both image and effects productions usually require some massive scene to be rendered that would include one of the effects as smoke, clouds, or even sometimes fuzzy but complexly shaped objects in the distance. While this was an issue even for voxel data of size 512 MB in the year 2009, current generation GPUs allow for a memory size of 6-8 GB as a standard VRAM capacity [10]. This fact of GPU memory limits having been increased over several years makes it a bit of a challenge to find a relevant isolated volume. At least one of a size that its dense representation would not fit into the GPU memory directly. However, single file cases aside, even for samples of small to moderate size that have temporal variance data, this memory issue again becomes important since they have to record the file in the animation frames, which is usually the case for animated or simulated volumes [9].

While for the sake of the scope of this paper, this discussion will not delve much deeper than this into the regular grids, it was important to mention them. Our approach uses dense grid representations frequently in the rendering process for producing the baseline reference images. That is because producing a reference image in Mitsuba 3 renderer requires using a volume grid plugin, which requires a dense grid representation either in the VOL format or Python dense array. Additionally, because the methodology initially includes all original grids in the form of OpenVDB files as a starting point, there is a need to convert them to dense representation. This procedure is included in all comparison steps because of the need to produce a baseline in memory and render time benchmarking. The conversion process is described in subsection 2.4.1.

1.2.2 Octrees

This particular data structure is widely known for both space partitioning and voxel data representation. First proposed to be used by Donald Meagher [3] as a direct successor of quadtrees for 2D representation, octrees can be generalized as hierarchical N-dimensional binary trees, specifically in the case of a 3D object representation. The topology of octrees meant that the node structure would always have a fixed number of children, exactly eight. Thus, through that hierarchical partition, they were able to represent the object comparatively a bit more efficiently than just dense grids. Despite there being several available memory packing options, most octrees are known for the breadth-first traversal, which enables ease of access, better cache efficiency, and predictable access patterns. However, as the authors note, still its main downside was its relatively bad scaling memory requirement.

While the use of octrees for voxel data has been popularized by works like that of Crassin (2009), these early works also include inefficient block-sized leaf nodes [10]. While this coincidentally makes it quite efficient in reading mip-mapped 3D textures, this also wastes a lot of memory. For example, in cases where the leaves do not contain true volume data, early octrees had leaves that typically contained 16^3 or 32^3 voxels, depending on the resolution.

With some clever pointer optimizations, there is another version of octrees,

namely Sparse Voxel Octrees, that enables octrees to be used for the volume representation [4]. This particular optimization adds that nodes always represent a voxelized space, with further partitioning being represented by the presence of children inside of a parent node. This structure encodes both the parent node and most of its children, minimally including the topology information in its 64-bit child descriptor that includes a 15-bit pointer to a first child and two 8-bit masks that indicate if the child is valid or is a leaf. The paper only includes a brief mention of this masking structure because of its light resemblance to the OpenVDB hierarchical structure.

1.2.3 OpenVDB

OpenVDB is an open-sourced variant of a data structure, also known as VDB, which was under closed development until 2012 in the Dreamworks Animation by Ken Museth [9]. The name is short for Volumetric, Dynamic grid that shares several characteristics with B+ trees. The file format itself has been initially used behind closed doors at Dreamworks for animated movies. Already, after VDB became open-source, it became more widespread with its more diverse use cases, such as representing background objects of complex structures that are not required to be rendered in detail.

Aside from OpenVDB, several other data structures precede it, finding their use in compactly representing level set and volume data, including octrees and Sparse Voxel Octrees, but as the authors of the VDB paper suggest, none of the aforementioned structures have been practically applied to animated volumes or simulated voxels [9].

The authors of the VDB approach address the data sparsity in a way that on a surface level, structurally resembles B+ trees. Specifically, it also employs cache coherency, similar to the B+ trees. Additionally, as a consequence of the resemblance, because of its topology creating multiple internal nodes with a high branching factor, it allows for sparsity of stored data.

In order to understand a bit more about OpenVDB, we focus on what are the B+ trees as they have a lot in common with VDB. Then, we provide a brief description of both its structure and features that differentiate OpenVDB from other sparse data structures and the problems it solves compared to its counterparts using those features.

Starting with B+ trees, although their definition is not documented to a standard in any paper, they are brought up as a B-tree variant in a publication by Bayer and McCreight [11]. Compared to the more widespread binary trees, both B-trees and B+ trees are used mainly in the application of reading from memory in blocks because of better structure and data alignment that allows for faster and more efficient sequential queries. That makes B+ trees a perfect choice for implementations of relational databases and some non-relational ones.

Structurally B+ trees resemble B-trees as both are M-ary trees, meaning that each node can have up to m children. Unlike binary trees, which are a specific case of M-ary trees with m equal to 2, B+ trees and B-trees can have a higher branching factor of m. Both B-trees and B+ trees are balanced trees, ensuring that all leaf nodes are at the same depth, and they maintain a balanced structure by ensuring that the number of children of each node ranges from $\left\lceil \frac{m}{2} \right\rceil$ to m. In

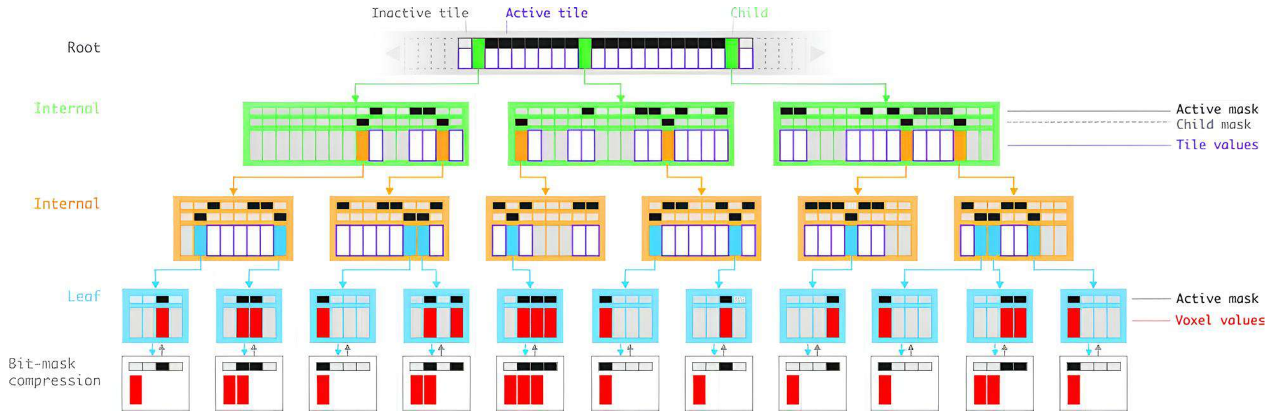


Figure 1.1 1D structure example (simplified) of voxel data packing in VDB tree. Going from top to bottom, there is a hashmap Root node, two Internal nodes, and block-size Leaf nodes. Both internal nodes contain two bit-masks that are integral parts of VDB tree traversal. Two bit-masks are the active and topology masks with mValueMask and mChildMask for the internal node structure, respectively. Voxels are represented in leaves in red. White nodes at the bottom are the compressed leaf blocks with only active (interesting) values present. All nodes have their own fixed branching factor, aside from the root node. It is noteworthy that the fixed branching factor is 2^N because OpenVDB tree traversal depends on fast bit operations.

B+ trees, all the actual data records are stored at the leaf level, and internal nodes only store keys to guide the search process, with leaf nodes linked to facilitate efficient range queries and sequential access.

Conveniently for our explanation, what the VDB has in common with B+ trees is the use of the high branching factor in all nodes of the tree, automatic balancing on the construction stage, and the fact the VDB is also an acyclic connected graph with its basic structure including a root, several internal nodes and leaf level nodes. What is different from B+ trees, however, is the absence of linked list connections on the leaves as well as varying but large branching factors of internal nodes, restricted to powers of two. These factors make OpenVDB a wide and shallow data structure that is suitable for storing vast amounts of data. Additionally, it also enables the storage of data on a close-to-infinite 3D grid when applied to volumes. Compared to the B+ trees, however, the VDB can encode the spatial coordinate information of the grid in each of its nodes, and not only its leaves [9].

VDB concepts To provide a clearer level of understanding of the features that VDB offers, there are several specialized concepts in VDB that must be described beforehand. These concepts also provide a relevant background for Chapter 3 because of the increased grasp of implementation details. As a standard, the data recorded into VDB is represented by a templated data type *Value*, accompanied by the *Topology* of named *Value*. The value represents the location of the node by storing its spatial index (x, y, z) as Cartesian coordinates. Values can be either *active* or *inactive*, with the activeness indicators stored in a node's binary mask. The value being *active*, as the paper puts it, represents its rate of "importance" and "interest." In the context of *Voxels*, the smallest spatial elements that can be recorded to VDB, the active voxel would represent a value that is different

from a background value, making it an object of "interest." In contrast to voxels, there are *Tiles* that represent areas of index space with the same value. Tiles are crucial in VDB for both memory efficiency and sparsity of spatial representation since a proper tile placement allows for representation of the entire subdomain of node's children with a single value [9].

Starting with Leaf nodes, they contain a compile time-defined, fixed amount of voxels, with a typical block size being 8x8x8. Leaf nodes have their enclosed voxel space encoded in a direct access table mLeafDat, while topology information is retained using a bitmask mValueMask. Direct Access Table in the context of VDB indicates an array of elements with a worst-case access complexity of O(1) [9]. What is important for later is that the mLeafDat is represented as a C++ union with its voxel values being able to be streamed out-of-core without additional memory overhead. This will become important in later sections as we address the voxel data by offsets in NanoVDB.

The structure of the Internal nodes heavily resembles that of leaf nodes, with the addition of mInternalDat recording not only the node values but also pointers to its children in a Direct Access Table. Another important change is an added bit mask, mChildMask, that allows for recording topology information. While both Leaf nodes and Internal nodes have compile-time-defined fixed branching factors, VDB allows for several levels of Internal nodes. In fact, the internal nodes comprise the main depth of the tree in OpenVDB. Additionally, while the branching factor is fixed for an Internal node, it is possible to have different node levels with a different number of children, just as shown in Figure 1.1, leading to a highly flexible data structure. In practice, however, the structure of most OpenVDB trees is quite fixed, as can be observed from the configuration files written for VDB reads; the structure is typically Tree_float_5_4_3, resulting in:

- Single root node (1 x 8)
- Several internal upper-level nodes (typically 8 x 32^3)
- Several thousand internal lower-level nodes (K x 16^3)
- Multitude of leaf nodes, typically in hundreds of millions (M x 8^3).

Lastly, when describing the VDB Root node, it is important to note that it is the only node type that does not have a Direct Access Table as its basis. A reason for this is that if it did have a strictly bound branching factor, the data structure itself would not be that much different from an m-tree with modifications. Since there would be a limit on the number of potential discrete spatial blocks that its children cover, the potentially infinite feature of the embedding space of VDB would not be possible. Despite the random access to a hashmap as well as a hash generation step both making the structure slower, there are bottom-up optimizing algorithms in place that amortize the cost of search traversals that start from the root node.

To better illustrate the explanation of the hierarchical structure of OpenVDB, Figure 1.1 is useful. Of utmost importance is the shallow and fixed depth of the tree, accompanied by the large fixed and predefined at construction, a set of branching factors for each level of the tree's depth, and an unbounded Root node. Both Internal nodes are essentially the same in structure and include, as such,

two bit-masks, one for active values and the other for hierarchical information about its children nodes, called mValueMask and mChildMask. Noteworthy is that the Root node has a hashmap as its choice optimization data structure.

1.2.4 NanoVDB

Despite all the advantages that OpenVDB offers, its main downside is the inability to be used natively on GPU. To be precise, as the authors put it, first of all, it was purposed for CPU use, and secondly, it is not designed to work on GPU due to its structure's heavy utilization of pointers. Because of that, the data locality inside of the structure is also much worse when compared to NanoVDB. To that end, in contrast to OpenVDB being a highly customizable structure with fast random access and in itself representing a wide and shallow B+-resembling tree, NanoVDB loses the customizability due to assuming the topology of the internal nodes structure to be static [12].

NanoVDB had adopted the majority of OpenVDB's best features, including its hierarchical structure, bottom-up recorded traversal of the tree and already integrated templating of standard types. Initially developed by Nvidia, it was integrated into OpenVDB as a way to add GPU support with a data structure that is fully compatible with the existing OpenVDB core.

Thanks to this and the way NanoVDB is implemented as a flat, contiguous memory buffer, not only does NanoVDB work on both CPUs and GPUs with a wide list of supported architectures, but it is also supported by several graphics APIs widely used in image production and game industry, like CUDA, OpenGL, DirectX, and HLSL, just to name a few. Thus, making it a perfect choice for future development as well as a focus of our thesis.

When packaged as a product, NanoVDB contains a reduced set of instruments available compared to its predecessor. As the biggest downside of this, NanoVDB does not have tools for modifying an already serialized grid; that is not to say it is impossible to modify a single or a series of values themselves. As a restriction, because of this, it focuses mainly on implementations of grids on static trees.

Essentially, what NanoVDB itself represents is a C++11 standard header-only library with a significantly reduced functionality of standard OpenVDB. At the time of writing our work, the NanoVDB is also accompanied by two additional header-only implementations, a CNanoVDB and PNanoVDB which are both minimized C99 standard usable implementations of core data structures and access methods, although when compared to CNanoVDB, PNanoVDB provides better coverage of original functions from NanoVDB.

1.3 Renderers

1.3.1 Mitsuba2

Li et al. first introduced the differentiable rendering approach and the accompanying challenges in 2018. [13] Challenges include having to manually differentiate every existing expression with the addition of a new model or algorithm to a differential rendering system. Mitsuba 2 addressed these issues by standardizing a set of types and disjointing the algorithms and various rendering methods by ways

of abstraction. This allows Mitsuba to be highly customizable and re-targetable to various applications. This thesis utilizes this render code generalization to customize the rendering system in our interactions with Mitsuba 3 as its direct successor, allowing native support of multiple backends.

This generalization of render algorithms is achieved by Mitsuba 2 being supplied in two different projects, Enoki, which is a template library, that handles vectorization and JIT compilation. The second project is the compilation of multiple approaches culminating in a differentiable render system, Mitsuba 2, which is a complete rewrite of the previous version, only borrowing the Mitsuba 0.6 scene description style.

Mitsuba 2 includes multiple features, however, for the scope of this thesis only several of them are of relevance. Starting with, template metaprogramming is a feature that gets heavily used in the Mitsuba renderer itself, with most of its features specified using target-specific abstracted types, like *Float* and *Spectrum* templates. Due to optimization on target-specific backends, like CUDA for support of vectorized computation and high-performance computation, the unused branches are pruned at compile-time utilizing *constexpr* evaluated code blocks. The last feature is the JIT evaluation, which includes Nvidia Parallel Thread Execution (PTX) intermediate representation code generation, branch pruning, and optimizations, as well as JIT GPU kernel optimizations. The emergence of such an approach was motivated by the unpredictability of vectorization when applied to physically-based rendering due to rendering code being able to jump to any part of the renderer [13].

Due to these Enoki and Mitsuba 2 features, the entire system can be characterized as a compile-time branched system that includes a Just-In-Time compiler, similar in application range to DrJIT, which is included in the later discussion of the following section.

1.3.2 DrJit

DrJIT stands for a Just-In-Time Compiler for Differentiable Rendering, which focuses on physically-based rendering by producing and aggressively optimizing the computation graphs of high-level simulation code as well as helping in simplifying the differentiable rendering algorithms. While the scope of this thesis does not require the coverage of differentiable rendering, the kernel evaluation and optimization are most interesting to us.

Modern-day effective computing requires the elevation of features like SIMD instruction sets and GPU compute shaders, depending on the host device area of the application. Tracing is the production of a trace, or a large computation graph, during the execution of a rendering algorithm. A traced graph typically includes a large computation graph of ”arithmetic, loops, ray tracing operations, and polymorphic calls” that later gets evaluated onto a host-backend’s large data-parallel kernel. DrJIT, with its roots in Enoki, already had support for its own kernel evaluation on GPUs and optimized PTX instruction set in the form of IR code. While the creation of traced kernels was done in Enoki as well, it only supported GPU with its CUDA backend. DrJIT, however, includes a recent addition of LLVM IR support, which enables highly parallelized CPU vectorized computing.

This traced computation graph, in essence, describes the exchange of information between the algorithm and scene objects, with scene objects including the object representation, light, sensors, and many other standard render primitives. A key difference between DrJIT and Enoki, however, is that the computation graph size is much larger as it includes a larger set of operations to evaluate before generating an unevaluated image in the form of a trace [14].

1.3.3 Mitsuba3

Mitsuba 3 directly builds on top of Mitsuba 2 with its plugin-based solution as well as the scene descriptor language support from Mitsuba 0.6, with its approach to rendering primitives being defined as scene objects.

Due to Mitsuba 3 evaluating its computation through a vectorized scene tracing in DrJIT, it also inherently supports both backends, CUDA and LLVM. These are recognized as variants in Mitsuba and are required to be separately specified with build flags to be compiled and built. Additionally, the Mitsuba 3 is inherently language-agnostic, as it is written on top of DrJIT, and most API calls having an equivalent in both Python and C++.

Separately, the Mitsuba 3 is once again a plugin-based system, with most objects and algorithms being directly interchangeable within the variant and defined through inherited class plugins, as shown in section 3.2. In this thesis, we make use of that fact to develop our own plugin solution for NanoVDB-supported volume path-tracing.

2 Methodology

This chapter provides a high-level overview of our work and main processes, including conversion, plugin implementation, rendering, and comparisons. Furthermore, it also covers most of the decisions involved in the implementation of the plugin itself. It also briefly summarizes both acquisition and comparison processes for the resulting rendered images. It is important to note that most of what this chapter covers is non-technical details of our work and some of the shortcomings, which will be additionally described in chapter 4.4 with the rest of the discussion.

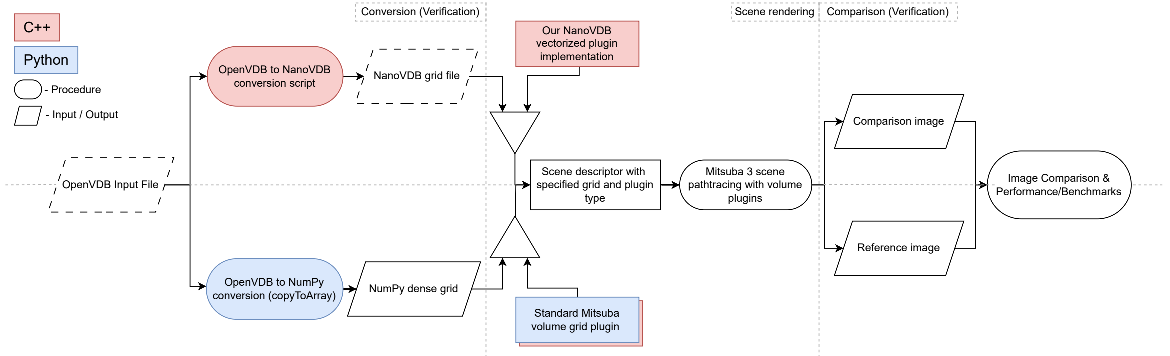


Figure 2.1 A schematic flow diagram for illustrating the conducted work and its general structure. This diagram outlines major sections of our process on a high level. The work is split into several steps, out of which distinguishable are major three. Starting from left to right, the VDB conversion stage processes inputs of OpenVDB to both sparse NanoVDB and dense grid NumPy. Secondly, there is a Mitsuba plugin involved in the rendering stage that loads the relevant converted file format into the plugin and outputs the rendered image. Lastly, in the comparison stage, both accuracy and performance are measured using image metrics as well as benchmarks. Additionally, the dashed lines indicate sparse data while solid lines indicate dense volumes.

2.1 Work outline

As illustrated in Figure 2.1 by a vertical split, this thesis can be symbolically separated into two major parts. One above involves our developed code, while the below interacts only on the Python side using existing API calls, either to Mitsuba plugins or OpenVDB converters. The exact configurations for replicating the complete setup are provided in the chapter 3 together with selected code samples. Just as illustrated, both processes receive the input in the form of a sparse volume file in the OpenVDB format. A rough common outline for these distinct sections can be described because of the shared similarities between the majority of steps for these two development parts.

It is imperative to note that the order in which this chapter is structured does not directly reflect the sequence of conducted work. Because a flow schema of the general process is provided here in the first section, it becomes possible to discuss the methodology steps in a way, that in our belief, would most benefit the reader's understanding.

Because of that, instead of VDB conversion, this chapter begins with section 2.2 that describes the differences in directly working with volumetric formats. That information becomes useful in the following sections, including one describing the implementation of the volumetric plugin. As the background for the data structures that are used in the thesis was already provided, this should be enough to justify not reintroducing the topic again in this chapter. Instead, the focus is shifted to a discussion of the peculiarities as well as distinguishing features of working with these voxel formats. That discussion also includes the limitations that follow the choices of data structures. Additionally, these feature constraints of some selected structures also define some of the restrictions imposed on the implementation itself.

Right after introducing the differences and some of the restrictions that come with our selected data structures, the section 2.3 follows. Its purpose is to provide an overview of the high-level implementation sequence for the production of the finalized NanoVDB plugin. The actual implementation with code samples will be included in chapter 3.

After that, section 2.4 attempts to gradually lead the discussion to our rendering and comparison stages, which are unified under it. Thus, verification is mainly there to provide a description of image production by path tracing the scene descriptors in the Mitsuba 3 renderer. Later, it is finalized by comparing their outcomes using various comparison methods and rendering time and memory benchmarks.

2.2 Comparing VDB structures and Dense grid

Before, the section 1.2 only gives a general overview of features for these VDB-typed data structures as well as some types of dense grids. That module also gives minimal context for how they might be used to represent volumetric grid data.

In contrast, the following description attempts to highlight the use limitations that arise from conversion from OpenVDB to Numpy dense and OpenVDB to NanoVDB. In light of that, the following paragraphs also serve as an introduction to the conversion topic of VDB, which will be discussed more in section 2.4.1.

This part focuses on the comparison of those same data structures mainly in the context of their use in plugin implementation. It additionally discusses a number of restrictions they introduce when utilized in the scope of this thesis.

The provided Table 2.1 is a short table of feature comparison that allows us to illustrate how the conversion from OpenVDB to other data structures affects the data, including data accessibility as well as introduces constraints in the conversion process itself, which we cover further in the section 2.4.1. It additionally allows us to argue for the choices of data structures and implementation headers done in the following section 2.3.

Starting from the top of the list, there is a question of voxel data structures being suitable for their use in GPU-side rendering. As it was already pointed out, out of all described structures, either due to the inefficient memory footprint of dense grids or due to the utilization of pointers in the topology of a volumetric structure for OpenVDB, only NanoVDB is suited for working on GPU. However, it does suffer in other aspects. Such shortcomings include being unable to write

	Dense Numpy	**Open VDB**	**Nano VDB**	**PNano VDB**
Optimized* for GPU	No	No	**Yes**	**Yes**
OpenVDB func. coverage	n/a	Full	**Min***	**Core***
Can values be modified	Yes	Yes	Yes	Yes
Can topology be modified	Yes	Yes	**No**	**No**
Depends on pointers	n/a	**Yes**	**No**	**No**
Multiple grids in a file	No	Yes	Yes*	No
Memory effective for sparse	No	Yes	Yes	Yes

Table 2.1 Selected feature comparison table for data structures that are used in the thesis. Column-wise, highlighted in bold, are the minority outliers in each data structure column, done only for increased visibility.

data into an already initialized grid, as well as assuming the grid structure to be static, as opposed to fixed at construction. Additionally, that means supporting a smaller set of instructions than the original OpenVDB, but for our scope, it is an advantage, as it significantly limits our code testing stage.

Separately, there is support for multiple grids being composed in one file. From the construction and API documentation of OpenVDB, it is known that both OpenVDB and NanoVDB support the addition of multiple grids to a single file. However, this is not the case for the dense voxel grids, as the data structure would quickly grow out of proportion, resulting in an even worse memory footprint scaling. Furthermore, as was pointed out in the OpenVDB paper, there is no known evidence if they were ever used for animated volumes, which is one of the main reasons for including multiple grids in a single file [9]. We point that out as one of the reasons because aside from path tracing in a scene file of multiple objects, which can also be achieved by simply increasing the number of referenced single grid files, other goals can potentially be achieved in the frame compositing.

Because both dense and sparse data structure formats are used in our comparison stage, we argue in comparing them in rendering performance in speed as well as memory. Figure 2.1 roughly allows us to summarize dense grids as a structure that is not designed for holding sparse volumetric data with dynamic topology on a massive scale grid, which is, inversely, a specialization of VDB [12]. Furthermore, section 1.2 in combination with the figure 2.1 helps us illustrate that in the case of 3D representations of large-scale data, a dense grid fails to fit the dataset into memory effectively. This problem is even more exacerbated in the case of GPU memory, as the inefficiency means that the dataset will completely fail to fit into the limited memory. That serves as support for one of the major reasons for this thesis, the utilization of VDB structures on GPU. Additionally, that serves as a special case in the comparison stage, representing data that always requires some sparse representation to allow storing it in GPU memory.

2.3 Mitsuba 3 plugin implementation

This section describes in high level terms our general approach to the implementation of deliverable plugin, from the first steps of accessing the grid handles and reading out float monochromatic values, to rewriting PNanoVDB core using both DrJit types and vectorized API. This part of thesis also includes a few paragraphs on the setup and implementation of Unit Tests that accompany all of implemented methods.

2.3.1 Accessing VDB values

Accessing the float values of either OpenVDB or NanoVDB is done using a set of required grid pointers and value read accessors that are obtained before the call to a type-templated read method. Our goal for this part was to ensure consistent results were obtained across files in both formats. As a first step, the simple data conversion setup was completed, which allowed the conversion of all input files from OpenVDB to NanoVDB format. In the same procedure, a series of random reads ensures that the results stay consistent across all files both before and after conversion.

Combined with our initial render, this step helps us in the long term by providing a baseline image for benchmarks and comparison. As for the VDB values themselves, there are several possibilities for acquiring the data from the grid. For both OpenVDB and NanoVDB, it is required to obtain a read accessor for the relevant grid type and then try addressing the read by the Cartesian coordinates of the queried voxel. In contrast, in PNanoVDB, as the data structure does not rely on multiple tree-level pointers, the API call requires obtaining multiple dummy grid handles. These are designated for several distinct levels of the tree, including for root, upper, and lower internal nodes, and later addressing of the value by its 3D coordinates.

2.3.2 Rewriting the NanoVDB to DrJIT

As it may seem from a quick glance, the simplicity of NanoVDB's calls is better suited for the end-user. However, for the thesis goals of implementing the plugin with NanoVDB support, the PNanoVDB header was a better choice for multiple reasons.

One of the reasons in favor of PNanoVDB is a much heavier use of templating in the NanoVDB codebase. What was initially predicted to become one of the main issues with NanoVDB during translation to DrJIT types was a type conflict when replacing the standard types with vectorized alternatives defined in DrJIT. However, this step to vectorize and rewrite all types and at least the stripped core of NanoVDB methods was a strict requirement in all cases. Mainly because this reduction of core code was unaffected by the choice of header library, the PNanoVDB header was selected as a final choice of implementation basis. Because NanoVDB itself is a cut-down version of OpenVDB, without including standard or any other libraries as dependencies, the header-only trimmed-down file of PNanoVDB takes it a step further. It is, in essence, a very slim header file with around 8200 lines of code, without any external dependencies. However, with the main goal being the delivery of the translated PNanoVDB code itself

and its integration into Mitsuba, it was necessary to trim it even further. This was possible because of the inclusion of multiple graphics APIs, including HLSL, GLSL, DirectX, and CUDA. The following steps were direct translations of existing standard-type replacements in NanoVDB. Following that was a minimally invasive vectorization of a method stack that comprised a complete call to a Cartesian coordinate read of a value stored in a queried voxel.

There were several options to achieve results that would be comparable to one another. One option was to attempt re-implementing all of the methods, including both core and helper, from the NanoVDB header. Another option was to utilize a cut-down version in either of the two, PnanoVDB or CNanoVDB headers. As both mainly contain the only core with some additional functionality coverage, we opted for the PNanoVDB as it was simpler to understand, provided more coverage, and was thus easier to end-test using a testing framework. Additionally, PNanoVDB was simpler to test because of the granularity of calls in the end API coordinate read. This factor allowed us to unit test different levels of implementation separately, without additional templating, ultimately ensuring the working capability of the entire module.

2.3.3 Framework unit testing

The unit testing phase utilizes DocTest, a lightweight C++ only testing framework for testing and asserting our code implementation's quality and reliability when implementing the NanoVDB to DrJIT. Following the TDD principles, test cases are defined for each function with an extensive amount of random querying subtests. Fortunately, most of the data that is returned from a vectorized method variant can be directly compared to an original PNanoVDB counterpart. Additionally, what DocTest helps with as a framework is an almost painless DrJIT test fixture setup with initialization and flag setting for debugging purposes. Separately, it defines test fixtures, encapsulating the setup and each test, allowing for avoidance of repetition of boilerplate code in setup phases for each fixture. Because of the number of functions that were required to be tested, the DocTest proved to be an irreplaceable part of our process, allowing us to ensure working order in all of the translated vectorized functions. Furthermore, it allows the testing of the complete plugin calls as a system-wide test by also providing an easy-to-use environment to compare and assert data equality to PNanoVDB results.

2.4 Verification

This section is mainly concerned with rendering for the purposes of producing an image from the file with OpenVDB or NanoVDB format, be it initial render in Mitsuba 3 by utilizing the built-in volume grid plugin, or the image produced as a result of the developed deliverable. In addition it describes general approach with a bench-marking process for obtained results and separately, for measuring performance metrics.

However, as can be observed in Figure 2.1, before the discussion of image production by path tracing, there is still a need to describe a bridge between the original inputs and the inputs that are fed into the relevant Mitsuba plugins. In

our case, it includes a description of the conversion process for OpenVDB format to both Dense NumPy and NanoVDB.

Because of that, this section first covers the VDB conversion process for both VDB cases in section 2.4.1. This includes conversion to uniform dense NumPy grid in Python as well as conversion to NanoVDB format in C++. Because section 2.2 briefly covers the differences in features between our original and target representations, we provide a description of the conversion process accounting for those restrictions.

2.4.1 VDB format conversion

Interestingly, despite how widespread the use of the format is, currently, there is no way of natively utilizing the sparsity of VDB in the Mitsuba 3 renderer. Specifically, the existing Mitsuba standard volume plugin requires for dense grid representation of the target volume instead of a sparse one. Thus, the only available way of utilizing the data is mediating it to the renderer by pre-processing it to dense grids. After processing, the data is transformed to the VOL simple binary exchange format that, in essence, represents a spatially encoded dense uniform grid.

However, this thesis uses another conversion process for these files. The reason is that the Mitsuba 3 procedure of converting OpenVDB to the VOL uniform grid format exists only in the form of an outdated and relatively unsupported conversion tool. Despite it still fitting our purposes for testing and an initial rendering stage, the conversion step utilizes the NumPy dense grid instead. The reason behind that decision was better documentation coverage by OpenVDB as well as its relatively quick availability.

As already shown in section 2.2, the dense grids do not have support for more than one grid to be recorded, including NumPy dense representation and VOL format. To summarize the effect this choice has on the thesis, this issue is relatively minor, as the same effect of multiple grids can be achieved in increasing the number of files in the scene. Despite that it still restricts our own comparison images to be produced with single grid files only. That is because our scene descriptors only have a single file in an attempt to decrease potential rendering errors.

Another minor issue was the misalignment of the resulting grid during conversion. That was resolved by ensuring the resolution of the declared NumPy grid matches that of the grid's bounding box resolution. The issue was noticed during the initial Python call for conversion to NumPy, as the Python grid itself has to be declared with the required resolution. During the conversion process, the resulting zero coordinate API call results in a bounding box shift and only a partial rendering of the original volume. This was linked to a re-centering that is done inside of the method call relative to the supplied zero coordinate parameter. Because of that, the only way to ensure one-to-one alignment of the correctly sized resulting grid is to utilize the meta-packed information for specifying the NumPy grid's new initialized zero as the minimum bound of the grid's bounding box in OpenVDB.

We have pointed out our limitation to single grid files in dense grid representations. For the NanoVDB conversion we have written another converting script.

The reason this conversion tool was required was because there are almost no existing and ready-to-use NanoVDB files in common access. A separate justification for this tool's existence was the added possibility of casting the grid pointer type to another grid pointer type. For the sake of simplicity as well as for better coverage of our own test cases, we limited from the start that we were working with a single monochromatic, value-only, float-typed grid representation.

Due to the initially described restriction on dense grids, we also constrain this conversion process. Thus, in the conversion script from OpenVDB to NanoVDB, we also use a single grid file output. Despite the API allowing writes of multiple grids to the single file, we choose to adhere to this choice for consistency and simplicity of the resulting script. For that reason, both our conversion methods and our process involves as source only one file with isolated single grid volumes and has clearly defined bounding boxes.

In order to ensure that the conversion went without major issues, our script also tests for equality a fixed number of random sampled coordinates in both pre-conversion and converted grids. As an output of our NanoVDB converter, we are able to receive a resulting file with .nvdb format that contains data similar in local coordinates and values as our original grid.

2.4.2 Baseline images

Due to our ultimate goal with this thesis being the production of a Mitsuba3 compatible plugin that would be able to rival the existing plugin for dense grid rendering, we need a baseline image to compare to.

The initial render and development were done using the OpenVDB Stanford bunny cloud model, as provided in the samples section on the OpenVDB website. All other models were also borrowed from OpenVDB for initial dense grid rendering and later NanoVDB conversion.

Because the Mitsuba3 plugin does not natively support OpenVDB or NanoVDB, we had to convert our borrowed datasets into a dense grid first, just as outlined in section 2.4.1. As per available instructions on the OpenVDB documentation page, the bunny was converted using the provided API call into a NumPy dense grid and rendered using a very small number of samples due to the hardware limitations of a target machine.

Ideally, the image was to be of higher resolution, but due to time constraints and hardware issues, the increase in target resolution was only achieved by using a computation cluster for rendering. An additional constraint was our scene iterations while developing the plugin because of fine-tuning the scene to work in both of our environments as well as adjusting for NumPy bounding box correction during the conversion process.

The initial rendering was produced on a laptop series RTX 2060 GPU with 6 GB VRAM by using the setup described in section 3.1. The image was produced in the resolution 256x256 pixels per image with a spp equal to 2, which is a very small count of samples per pixel (spp). The cluster-produced images, however, allowed for much higher computational throughput, so the final renders were produced using a resolution of 1024x1024 pixels square image and a converging spp of 1024 for all images. Additional details on results recording are described in chapter 4.

2.4.3 Setting up a comparison scene

In order to achieve any significant result aside from the base render, there is a need to focus on two major objectives. The first objective is creating a plugin itself, which is described in detail in chapter 3. Following that, the second goal is to produce a reliable and reproducible example scene that would be nearly identical to the rendered baseline image. This standard scene file is necessary because otherwise, it is hard to produce relevant and consistent comparison data otherwise.

To give the reader a better understanding of scene descriptors, the main elements of the scene file that should be present are described. This scene file is also required for minimal reproduction of our results as well as for minimal rendering of the scene in Mitsuba. While a Mitsuba scene description can be as simple as placing the object file inside of a space, the bare minimum a scene requires is a model to be rendered, a single light source, and a single sensor representing a camera view to be rendered. Additionally, as Mitsuba is a plugin-based renderer, most of the plugins have their internal parameters available for tweaking. All of the standard and in-plugin parameters can be exposed and modified, making it relatively easy to set up a scene. Our scene iterations also use that process for setting up reference and final renders. The entire process mainly focused on modifications and applying correct transformations to the centered volume file. With these iterations, a primary goal was to try and match the resulting world transformations of queried 3D Cartesian coordinates to their correct positions.

The resulting comparison scene was set up using the centered unit cube with an empty material for the enclosing cube and a volume with a fixed scaling centering transform placed inside of it. Additionally, for consistency, a set of fixed-angle increment rotating sensors with an elevated angle is provided for all reference sets of images in a manner similar to a turntable rendering, as described in chapter 4.

2.4.4 Implemented plugin render

Our plugin implementation takes the code of NanoVDB as its reference and combines the existing minimal functionality required for the grid volume Mitsuba 3 plugin to work. That minimal set of functions includes a call to a texture evaluation method that directly samples and interpolates the values of a plugin class's member volumetric texture in the simple binary exchange format VOL. In our implementation, that volumetric texture sampling call is replaced with a read to a NanoVDB file, with removed interpolation because of a raw integer-rounded coordinate read, which then reads the value for the transformed world space coordinates already in the local coordinate space for the file.

In our experiments, we have limited ourselves to a call to a monochromatic interpolate function that originally samples the 3D texture based on local Point3D coordinates, affected by grid transforms from the scene descriptor. For the entire stage of development, the sample count was kept low to increase the feedback effectiveness and produce results faster. In the evaluation and performance sections of our work, the sample count was kept close to 1024 with an additional increase in the rendering image's resolution to achieve render convergence and reduce image noise.

For the result that was pursued, our plugin utilizes the same scene descriptor

set up for the comparison scene. That helps once again in the case of direct resulting image comparison as well as for clearing up the resulting combinations of transformations to be exactly the same in both cases. This is additionally supported by the accuracy measurements. Just as was outlined in chapter 2, while the development for the plugin was done in C++, the rendering of the volume itself was done using the Mitsuba 3 through Python calls with a Jupyter Notebook environment.

In the final rendering stages of our process, the entire setup is transferred to a Docker container while replicating and combining the results achieved in the previous steps with both plugin implementation and rendering Jupyter Notebook setup. The finalized rendering is done in a much higher sample count and increased resolution in order to achieve a meaningful comparison in our following section 4.4

2.4.5 Performance and Benchmarks

For the comparison of the results in terms of accuracy, the MSE and SSIM are used as our primary image comparison metrics. These are conducted using renders of the same scene description, with the only difference being the use of a plugin. All images are rendered with 1024 spp as the resulting number of samples per pixel to achieve a compromise between the time of rendering and image quality achieving convergence.

The performance results on both speed of rendering as well as memory utilization are done using Python *Time* library for recording rendering time in between the Mitsuba render calls. The memory measurement is parsed from the Mitsuba *whos_str()* method call that returns a string for all traced variables involved in the call as well as device memory for both CUDA and LLVM variants. A more detailed explanation of the recording process as well as analysis is attached in chapter 4.

3 Implementation details

This chapter describes the necessary steps for recreating the setup. It starts with the complete development environment, including relevant library names and dependencies, but omits exact version history. Because of its relevant simplicity, the chapter provides only a short description of the technology stack used. A huge portion of its focus is devoted to describing the detailed process of the plugin development itself and conducting unit tests that were added to ensure code correctness and adherence to the original NanoVDB results. Additionally, some space is devoted to describing the image production process with path tracing, which will include scene setup, exact rendering settings, and, separately, comparison and recording of benchmarks. The end sections of this chapter additionally describe several challenges encountered in the implementation process.

3.1 Our setup

The initial project setup was completed on a locally virtualized Windows Subsystem for Linux (WSL ver.2) with dedicated 12 GB RAM and 128 GB swap space. Completed Mitsuba plugin builds, including its initial implementation, as well as unit testing utilizing CMake and Ninja generator as a build system. The entire project configuration with relevant versions is packed in our repository to increase the chances of it being reproducible. For reference, both full version information and a quick setup script in attachments A.1 and A.2 are also included.

Before, in the Methodology sections, it is mentioned that our project is separated into our C++ implementation and OpenVDB/Mitsuba used Python side code and that this language separation acts as its rough division. Factually, our project consists of three smaller projects. Two projects are C++ only, one including a DrJIT-based NanoVDB implementation with added unit tests and another a separate Mitsuba 3 build with a developed NanoVDB plugin that acts as a dependency for our rendering. Lastly, there is a Python-only Jupyter notebook containing scene descriptors that links to the Mitsuba build as its dependency, with Mitsuba acting as the project's renderer. The dependency between the two projects is managed by connecting them using the *PYTHONPATH* environment variable and a linker bash script that is configured inside of the Mitsuba build.

The second significant half of the implementation is set up on a computation cluster using Docker containers for easier dependency management. The cluster was used only after several iterations of code optimizations. Its main purpose was the final path-tracing of resulting images, with the finalized parameters being fixed across all renders. To achieve that, the local machine setup of several projects was recreated in the Docker container while retaining most of the original linking by rebuilding the projects from scratch inside Docker and connecting their dependencies. Rendering projects that use the Jupyter notebooks and the Mitsuba plugin implementation are connected by linking the Mitsuba to the Python environment path in Jupyter, similar to the local machine setup.

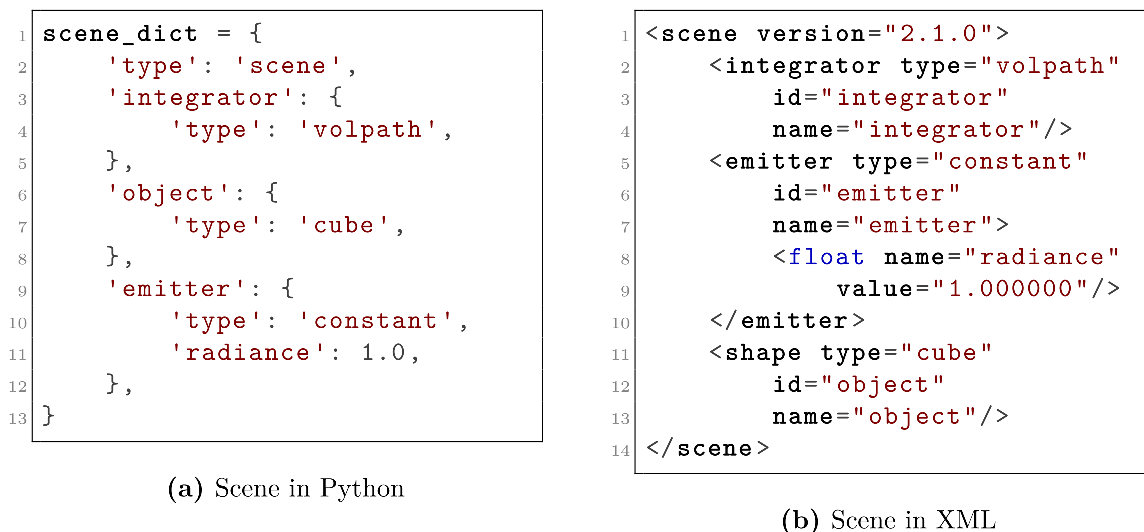
The reason for specifically choosing Docker as our final rendering setup was twofold. First was the hardware limitations of our local machine, which was also the initial reason behind using only a small number of samples during the

plugin implementation phase. Additionally, the resulting images of compute cluster rendering were used to compare the results. The second reason for selecting Docker was that it allowed us to achieve a more precise benchmarking process. Because it allows isolating development containers similar to the environment virtualization. The final reported comparison data was also recorded using the compute clusters. Thus, Docker was an obvious choice there, as the added possibility of isolating memory and execution speed allowed for more precise benchmarking, which was specifically useful for this case.

3.2 Scene descriptors

Earlier, only a brief overview was given of the scene descriptors' main components that are required for rendering the scene, as well as some of the components required by the default volumetric plugin module to ensure its basic functionality. This section can go into deeper details and provide several examples of the code structure of our scene descriptor, as well as describe the exact parameters we use in our plugin against those used in the default volume plugin. This step is relevant to the loading of the plugins at the start of the scene rendering section in our methodology chart, as illustrated in Figure 2.1.

What is interesting from these descriptors is the fact that all the plugins and rendering primitives used in Mitsuba can be classified and inherit the object class. That includes the scene, objects in the scene, light emitters, camera sensors, and many, many more. This can be illustrated in Figure 3.1 as well as Figures 3.2 and 3.3 with the dictionary key *'type'* that signifies the exact plugin to be linked.



(a) Scene in Python

(b) Scene in XML

Figure 3.1 Example of a simple scene in both XML and Python

The standard for scene descriptors is specified in the XML format. All of the scene descriptors can be provided either in the XML format or in the Python dictionary format, depending on the convenience and preference of the end user. It is additionally possible to convert in-between formats using the API calls to Mitsuba's *dict_to_xml()*. In our implementation, we utilize both, with our primary use case being the Python format because of improved readability for the scene iterations itself. Separately we utilize the conversion call from Python

dictionary to XML format for debugging purposes in C++ Mitsuba build because of the faster build times and a removed project linking step.

```
'object': {
    'type': 'cube',
    'bsdf': {'type': 'null'},
    'interior': {
        'type': 'heterogeneous',
        'sigma_t': {
            'type': 'gridvolume',
            "use_grid_bbox": False,
            "filter_type": "nearest",
            "grid": volume_grid,
            # "filename": "scenes-m3/dataVol.vol",
            'to_world': T().scale(3).translate([-0.5, -0.5, -0.5])
        },
        'albedo': 0.999,
    },
    "to_world": T().scale(2)
},
```

Figure 3.2 Use of the standard gridvolume plugin parameters

```
'object': {
    'type': 'cube',
    'bsdf': BSDF,
    'interior': {
        'type': 'heterogeneous',
        'sigma_t': {
            'type': plugin,
            "grid_n": 0,
            "grid_type": "float",
            'vdb_filename': FOLDERNAME+filename+FILEEXT,
            'to_world': T().scale(3).rotate([1., 0., 0.], 90).
               translate([-0.5, -0.5, -0.5]),
        },
        'albedo': 0.999,
        'scale': float_scale,
    },
    "to_world": T().scale(2)
},
```

Figure 3.3 Use of the our grid_pnano plugin parameters

These samples are provided to illustrate and highlight the required parameters in both the standard volumetric plugin as well as our developed NanoVDB plugin, pointing out the differences. As Mitsuba 3 allows for plugging of variables in Python, the VOL input to the plugin can be defined as a variable, which is illustrated in Figure 3.2 by *grid* parameter. It is also possible to supply the input in the form of a filename with VOL extension. What is, however, absent in our implementation is the parameter *filter_type*. Because we are simplifying the function calls to its bare minimum, there is no volumetric texture interpolation, as we only sample single-point data.

As can be illustrated in the provided code samples for setting up the plugin scene descriptors, a set of minimal rendering functionalities for our goals can be ensured by providing object file references, a light source, and a rendering sensor, or in other words, a virtual camera. While it is possible to define only an object in Mitsuba 3, it would not provide the extended customization that our scene asks for. Thus, all objects are to be defined explicitly for increased replication and better comparison results. That is because each file includes standard lighting and transforms for the standard volumetric plugin. Thus by only providing the object grid in the VOL format, the resulting images would not have proper transforms, sometimes making them not visible from the standard camera view angle.

3.3 NanoVDB conversion

While we already described the bulk of our conversion procedure due to its relative simplicity in the Methodology section, this subsection of Implementation is here to provide a detailed example of how the grid pointer casting works, what access options are available to the grid tree data, and what limitations we were imposed with because of using the selected conversion method. This part is related to the conversion section of our methodology in Figure 2.1.

Several concepts need to be introduced once again to describe the inner workings of OpenVDB at a level that can be grasped. As already described multiple times, the VDB is built on top of a B+ tree resembling structure, with the tree holding the data recorded at each ijk index in Cartesian space, when the query is sent for a data read by that index. Trees have different common aliases of form FloatTree or UInt32Tree that in essence are defined in *openvdb.h* as some of the commonly used tree topology type aliases, with the most common being FloatTree, as can be seen in Figure 3.4.

Because this type of addressing to the tree is enclosed by a container class, that is our utilized *Grid* class, we first need to read the file's grid descriptors from the stream and then access them by name, reading the *GridBase* pointer.

Unavoidably, the obtained grid pointer needs to be downcast from a generic grid pointer to a concrete Grid type alias because BaseGrid does not have the generic ReadAccessor methods. To achieve that, a call as seen in Figure 3.4 to convert the grid is made. With the grid pointer casting, there are multiple supported grid types that, in essence, are cast to a concrete Grid class with definitions of the concrete tree value type. That templated method checks for gridType names and performs a static_cast on the shared grid pointer.

On a lower level, this is what happens during a grid read and grid pointer conversion. In our implementation, the simplified basis was, however, provided in the form of a Quick Start page from the OpenVDB documentation. Thus, in order to implement the basic conversion for the NanoVDB format from OpenVDB, the API requires a pointer cast to an OpenVDB grid type. With a later call to a conversion API method that will convert the pointer to a NanoVDB handle.

Because of the absence of topology pointers in NanoVDB, the main structure pointer is contained within a HostBuffer class that contains both buffer size and *data*()* pointer to the raw buffer for the NanoVDB grid.

```
using FloatTree     = tree::Tree4<float,       5, 4, 3>::Type;
using Int32Tree     = tree::Tree4<int32_t,     5, 4, 3>::Type;
using Int64Tree     = tree::Tree4<int64_t,     5, 4, 3>::Type;
...

using FloatGrid     = Grid<FloatTree>;
using Int32Grid     = Grid<Int32Tree>;
using Int64Grid     = Grid<Int64Tree>;
...

// Reading file descriptors from the stream for a given filename
openvdb::io::File file(source);
// Populates the grid descriptors without slow-loading the grid
    itself
file.open();
openvdb::GridBase::Ptr baseGrid;

// Get the gridName on the index 0
...

// Reading and casting a grid pointer to a concrete type
baseGrid = file.readGrid(gridName);
openvdb::FloatGrid::Ptr srcGrid =
    openvdb::gridPtrCast<openvdb::FloatGrid>(baseGrid);
```

Figure 3.4 Tree type defines and grid pointer casting

3.4 Plugin implementation

This section describes the implementation detail of the deliverable plugin, as illustrated by a red rectangle in Figure 2.1.

The nanoVDB itself heavily utilizes all standard types in both 32-bit and 64-bit alignment. Besides, the entire PNanoVDB header is parameterized to be compiled depending on the define macros, as it is a header-only library that supports multiple languages, including graphics programming ones like GLSL and HLSL. Additional code branching is introduced by splitting the methods for x64 and x32 bitness. For simplicity, and because our main focus is the plugin delivery itself, this thesis initially limited itself to the x64 system bitness, with the relatively easy potential addition of x32 systems support in the future.

As was also already pointed out, we selected the file of PNanoVDB.h for the implementation basis as well as a comparison code for which to write unit tests. With the first step being the conversion to DrJIT vectorized types, the task was simplified to figuring out a way to reuse the types from DrJIT in replacing the C++ standard types that were used throughout the header file to the DrJIT variants, with added traced loops and masking where it was necessary. While for most methods, this procedure was as simple as line-by-line translation without any significant modifications, some of the functions required to be modified a bit. Due to the vectorization nature of DrJIT, most of what was rewritten included the functions that had multiplication-addition arithmetic operators of new JIT array types. Additionally, all of the branching *if* cases were replaced by masking.

The only current dependency of our implementation is the NanoVDB's header-only library from the OpenVDB repository, which is only used for the *.nvdb* file

```
// Creating a file object, reading from file and grid descriptors
...

openvdb::SharedPtr<openvdb::FloatGrid> srcGrid =
    openvdb::gridPtrCast<openvdb::FloatGrid>(baseGrid);

// Convert the OpenVDB grid, srcGrid, into a NanoVDB grid handle.
auto handle = nanovdb::createNanoGrid(*srcGrid);

// It is possible to straightaway write the NanoGrid handle to a
   file
nanovdb::io::writeGrid(convertedFilePath, handle);
```

Figure 3.5 Conversion to NanoVDB and writing to file

format IO operations, as well as for storing the PNanoVDB's static grid offset array *pnanovdb_grid_type_constants* for the duplication avoidance, that are initially utilized in the PNanoVDB's implementation. This has to be added either way, as the static array of constant offsets will be extended at some future points by the VDB team, thus making it possible to extend the float-grid-only support of our prototype to cover more grid types.

Challenging to the implementation itself was the *pnanovdb_root_find_tile* method, not only because of issues with vectorization but also because of added complexity when attempting to test it on the LLVM IR instructions produced by JIT tracing, as produced logs resulted in the bug of unknown nature with an undefined *%mask* variable. That forced a temporary workaround and a switch to a separate build for the project of Mitsuba 3. The added complexity was purely coincidental, as the main repository was and is still currently experiencing a switch to a new Python binding library, *nanobind* instead of *pybind11*. To be exact, the project was switched to the version of Mitsuba supplied in the nanobind_mitsuba_bindings branch with relevant submodules. This version repository hash for a DrJIT submodule was also copied to the implementation folder as it was at the stage before integrating into the Mitsuba plugin. The reason for choosing this version was a suggestion to switch to nanobind when contacted for help with the abovementioned LLVM bug. It was given by the author, Wenzel Jakob, as Mitsuba was once again on a continued path towards utilizing nanobind, and supposedly, that branch was more stable.

Unfortunately, right after the switch and a working implementation on the nanobind branch, on the stage of integrating the translated PNanoVDB code into CUDA, it turned out that the CUDA version was not working for some reason on the Docker machine in that branch and the latest master, specifically. We attribute this factor to the currently ongoing transition to nanobind, as it is also supported by issues being submitted for the last several months in relation to CUDA compatibility. This forced the last switch with the rewrite of the same function to the old traced loop version, as shown in Figure 3.7. With the last version presented in this thesis, the base branch with all the modifications added is stable v3.5.2 of Mitsuba and v0.4.6 of DrJIT.

```
PNANOVDB_FORCE_INLINE
pnanovdb_root_tile_handle_t pnanovdb_root_find_tile(
    pnanovdb_grid_type_t grid_type,
    pnanovdb_buf_t buf,
    pnanovdb_root_handle_t root,
    PNANOVDB_IN(pnanovdb_coord_t) ijk)
{
    pnanovdb_uint32_t tile_count =
        pnanovdb_uint32_as_int32(pnanovdb_root_get_tile_count(buf,
            root));
    pnanovdb_root_tile_handle_t tile =
        pnanovdb_root_get_tile_zero(grid_type, root);
    pnanovdb_uint64_t key = pnanovdb_coord_to_key(ijk);
    for (pnanovdb_uint32_t i = 0u; i < tile_count; i++)
    {
        if (pnanovdb_uint64_is_equal(
            key,
            pnanovdb_root_tile_get_key(buf, tile)))
        {
            return tile;
        }
        tile.address =
            pnanovdb_address_offset(
                tile.address,
                PNANOVDB_GRID_TYPE_GET(
                    grid_type,
                    root_tile_size));
    }
    pnanovdb_root_tile_handle_t null_handle =
        { pnanovdb_address_null() };
    return null_handle;
}
```

Figure 3.6 PNanoVDB method pnanovdb_root_find_tile

3.5 Unit testing

Our unit tests utilize a Doctest header-only lightweight framework for testing the implementation of the PNanoVDB functionality to a DrJIT-compatible version. The entire translation process attempts to adhere to the principles of test-driven development.

The overall procedure for the tests was shown in the previous sections; in this one, we want to focus on the specifics of code testing, including test windup and fixture setup.

DrJIT targets two backends, and usually, it is Mitsuba that handles their switching at runtime with the const expression evaluated conditionals added throughout the code. However, in our separate project that only contains DrJIT, the switch is done using a header for *defines.h*, which holds a manually switched define macro for AD and LLVM / CUDA switching. The initially integrated variant of the code also used that before switching to Mitsuba templated types. As the define macro determines the backend, DrJIT has to initialize the selected one using a call to *drjit::init(JitBackend.CUDA)* or alternatively *jit_init(JitBackend.LLVM)*. DocTest helped us avoid firing up the backend initialization through the use of

```
PNANOVDB_FORCE_INLINE drjit_root_tile_handle_t
    drjit_root_find_tile(
        drjit_grid_type_t grid_type,
        drjit_buf_t buf,
        drjit_root_handle_t root,
        PNANOVDB_IN(drjit_coord_t) ijk,
        Mask active)
{
    MI_MASK_ARGUMENT(active);
    UInt32 tile_count = drjit_uint32_as_int32(
       drjit_root_get_tile_count(buf, root, active), active);
    drjit_root_tile_handle_t tile = drjit_root_get_tile_zero(
       grid_type, root, active);
    UInt32 tileFixedOffset = drjit::full<UInt32>(
       PNANOVDB_GRID_TYPE_GET(grid_type, root_tile_size));
    UInt64 coordKey = drjit_coord_to_key(ijk, active);
    UInt64 tileAddressOffset = tile.address.byte_offset;
    UInt32 byte_offset = drjit::full<UInt32>(uint32_t(
       PNANOVDB_ROOT_TILE_OFF_KEY));

    UInt32 i = drjit::full<UInt32>(0);
    Bool foundMask = drjit::full<Bool>(false);
    Bool curMask = drjit::full<Bool>(false);
    Bool modifyMask  = drjit::full<Bool>(false);
    drjit::Loop<Bool>
        loop("Root Find Tile", i,
            coordKey, tile_count, curMask, foundMask, modifyMask,
            tileFixedOffset, tileAddressOffset, byte_offset, buf);

    while(loop(i < tile_count)){
        drjit_root_tile_handle_t dummyTileHandleInLoop =
            { drjit_address_t { drjit::fmadd(i, tileFixedOffset,
                tileAddressOffset) } };
        UInt64 currentKey = drjit_root_tile_get_key(buf,
           dummyTileHandleInLoop, active);
        curMask = drjit_uint64_is_equal(coordKey, currentKey, active
           );
        Bool pos = foundMask.and_(curMask);
        Bool neg = foundMask.not_().and_(curMask);
        modifyMask = pos.or_(neg);
        foundMask = foundMask.or_(curMask);
        tileAddressOffset = drjit::select(
            modifyMask,
            drjit::fmadd(i, tileFixedOffset, tileAddressOffset),
            tileAddressOffset);
        i += 1;
    }

    drjit_root_tile_handle_t null_handle = { drjit_address_null(
       active) };
    tile.address.byte_offset = drjit::select(foundMask,
       tileAddressOffset, null_handle.address.byte_offset);
    return tile;
}
```

Figure 3.7 DrJIT translated NanoVDB method drjit_root_find_tile

fixture setups and teardowns, which are always executed at the start of the testing phase. Similarly, all of the testing code is enclosed with relevant fixture code to reduce code replication.

3.6 Integration to Mitsuba

Our second project involved building our own Mitsuba 3 version with the already-tested translated NanoVDB code added to the renderer. As can be noticed in-code, the unit testing code part was excluded from the renderer builds. This was done in order to not over-encumber the resulting project as well as to ensure the unit tests stayed separate from the final build because the Mitsuba uses their own set of test suites.

The process of integrating NanoVDB DrJit variant translation into the Mitsuba itself was done by minimizing the functional calls for the existing volumetric grid plugin. This procedure enabled the use of a developed plugin natively in connection with already existing scene descriptors. The adjustments to simple grid plugin descriptors were minimal, as only several new parameters were added, and most of the old ones were replaced, as illustrated in examples of section 3.2.

Writing and adjusting the already existing volume plugin function calls for sampling the underlying 3D texture. Instead, as already noted, we opt to only utilize the single read, as the 3D texture would allow us more functionality but less in terms of reliability of our prototype, and it is not covered by the unit tests, as we don't yet have integration tests added to Mitsuba.

A completed DrJIT implementation of the translated NanoVDB code was attempted to be included in the Mitsuba 3 codebase several times. With each attempt, something was either not optimized, outright broken, or produced a CUDA or an LLVM side error. While the previous stage of our development pipeline has helped us immensely, this is a compilation of all major changes done to the code to get to its current stage. This includes various noteworthy fixes and also stuff that was not covered by static branched define macro-ed tests

Templating and Masking

DrJIT has its own templated types, and Mitsuba3 uses the same types, just defining them under a different templated alias. Thus, the inclusion of Mitsuba 3 types required simply adapting our standard templated types from the standalone NanoVDB translated implementation to the already used notation.

Considering masking, there are several macros that Mitsuba 3 provides for writing new plugins that are added for the inclusion of exposed masks that are used in traced graphs for aggressive optimization. However, it wasn't until the second branch switch, from the nanobind to v3.5.2, that this action was needed. In short, for this procedure, we only needed to template all of the functions by enclosing them in a templated class *DrJitImplementation* and then extend them by passing through all method calls, a Mask active parameter. The resulting change looks something like Figure 3.8.

```
PNANOVDB_FORCE_INLINE UInt32Jit drjit_read_uint32(drjit_buf_t buf,
    drjit_address_t address)
{
    return drjit_buf_read_uint32(buf, address.byte_offset);
}
```

(a) Before added templating and masking

```
class DrJitImplementation
public:
    MI_IMPORT_CORE_TYPES();
    ...

    PNANOVDB_FORCE_INLINE UInt32 drjit_read_uint32(drjit_buf_t buf,
        drjit_address_t address, Mask active)
    {
        MI_MASK_ARGUMENT(active);
        return drjit_buf_read_uint32(buf, address.byte_offset,
            active);
    }
```

(b) After added templating and masking

Figure 3.8 Adding templating and masking during Mitsuba integration of PNanoVDB

Fixing unbounded reads

This issue showed up only after the attempt at reading from a bigger file, as the entire initial development was based on the bunny cloud fog volume as a testing sample for both CUDA and LLVM variants. It wasn't until then that mainly, the unit tests fired off that there was a mistake in the read access for an invalid memory offset. Why this occurred was then evident.

Because this is handled by a simple if clause in the original, and masking should have done the same work, the caveat was that DrJIT tracing evaluated both paths in the *select* call. And because the original addressing was still exceeding the allocated bounds, it was causing a segmentation fault error. The solution was to plug that with a preceding select to nullify the offset before a read step is recorded in the traced graph.

Optimizing malloc

Originally, in the first phase of implementation, the version of the prototype that used define macros, has utilized double memory allocation for both UInt32 data and UInt64 data as a dirty way to overcome the failing tests. In the final implementation, instead of a double malloc, we figured out a better way to both satisfy tests and allocate the memory only once. This was done by utilizing a DrJIT load_aligned call that allocates memory on the DrJIT side for our UInt32 data and maps the same memory pointer to the allocated space for the UInt64 data. This change can be seen in commented lines in the call drjit_make_buf as well as Figure 3.9.

```
PNANOVDB_BUF_FORCE_INLINE drjit_buf_t drjit_make_buf(const uint32_t*
    data, uint64_t size_in_words)
{
    uint32_t byteSize = size_in_words;
    uint32_t data32Size = byteSize / 4;

    drjit_buf_t ret;
    // ret.data = drjit::empty<UInt32>(data32Size);
    ret.data = drjit::load_aligned<UInt32>(data, data32Size);
    // ret.data64 = drjit::empty<UInt64>(data32Size << 1);

    if constexpr(IsJIT) {
        // if constexpr(IsLLVM) {
        //     jit_memcpy(JitBackend::LLVM, ret.data.data(), data,
            byteSize);
        //     jit_memcpy(JitBackend::LLVM, ret.data64.data(), data,
             byteSize);
        // } else {
        //     jit_memcpy(JitBackend::CUDA, ret.data.data(), data,
            byteSize);
        //     jit_memcpy(JitBackend::CUDA, ret.data64.data(), data,
             byteSize);
        // }
        ret.data64 = drjit::map<UInt64>(ret.data.data(), data32Size
            / 2);
    }
}
```

Figure 3.9 Optimizing memory allocation

3.7 Rendering and Performance

As already covered in the outline of this chapter, there exists a rendering-only Jupyter Notebook solution that was originally set up in a closed Python virtual environment. The initial setup in the environment was completed using Python 3, and both standard Mitsuba 3 and Jupyter Lab were installed through pip packages. This Jupyter Notebook rendering virtual environment is later manually connected to the Mitsuba plugin builds. As the dependency for mitsuba3 is used, we change the linked version by referencing the relevant Mitsuba build that includes the latest plugin code. This version switch is completed by modifying temporary console environment variables using the *source* CLI command to a *setEnv.sh* script, automatically generated by Mitsuba build. The exact setup replication will be described in the relevant repositories' readme files as well as in Appendix A.2.

3.8 Packaging to Docker

When attempting to record the profiling numbers, we concluded that we would need a more performant system than what was available locally. The first choice, as described in previous sections, was a computation cluster of MFF Charles University, server Mayrau. We rolled out a docker on the version of Ubuntu that checked all our boxes and included CUDA support. This was only available through the image of *nvidia/cuda:11.7.1-devel-ubuntu22.04*, as it allowed for relevant installations of dependency libraries for both OpenVDB and still was within the maximum version that was supported hardware-side, as CUDA v11.7.

The entire setup was moved through GIT repositories and rebuilt on the Docker itself, with dependencies being managed by a few shell scripts linking it all together. More on the performance capture process and Docker setup will be additionally covered in chapter 4

4 Results

This chapter's purpose is to provide description of the results obtained during the rendering and profiling phases, with added details on procedures of benchmarks and image comparison. This part is additionally a post-mortem document of most of the findings, in combination with chapter 4.4 providing a whole picture of both effective results and observations done during implementation. As well as some of the shortcomings that were encountered during the process of results acquisition.

4.1 Test scenarios

This work limits the sections of this chapter to testing on six unique scenarios, not including duplicated ones that are introduced through the generation of multiple dense grid level sets. Selected scenarios include freely available files from OpenVDB of moderate size, with the biggest being *crawler*, which takes up to 10.5GB in dense equivalent. This baseline profiling number is provided by OpenVDB's *vdb_read* tool as a reference point for dense representation on the device. In later sections, both benchmarks, as well as relevant conclusions, are provided based on different comparisons. Thus, to provide a relevant frame of reference for scenario files that will be used throughout, a Table 4.1 is provided below. The recorded data in the table is provided by the same *vdb_read* functionality of the OpenVDB built tool.

The recording process is conducted on the final Docker container setup with LLVM 14 and CUDA 11.7. All resulting files are in the FogVolume format, with most being converted inside the Docker container to FogVolume and later written to NanoVDB format using the VDB conversion tool. As noted already, in the case of baseline images, depending on the comparison purposes, different variants as well as different plugins were used. Mainly, what is important to remember for later is that the baseline always uses the dense grid representation, no matter the variant used.

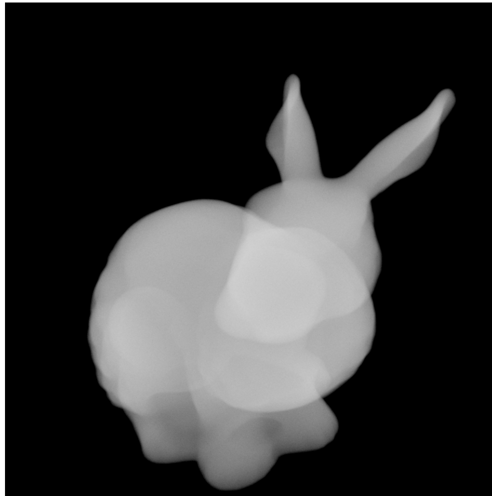
Level sets (armadillo, bunny, crawler, dragon)

Level sets in VDB are represented by three distinct voxel regions, inside, outside, and a thin narrow-band region. Both inside and outside are inactive, with constant signed distances to the object's surface. The narrow-band, typically represented by three voxels on both sides has a signed value signifying distance to the surface, with positive on the outside and negative on the inside.

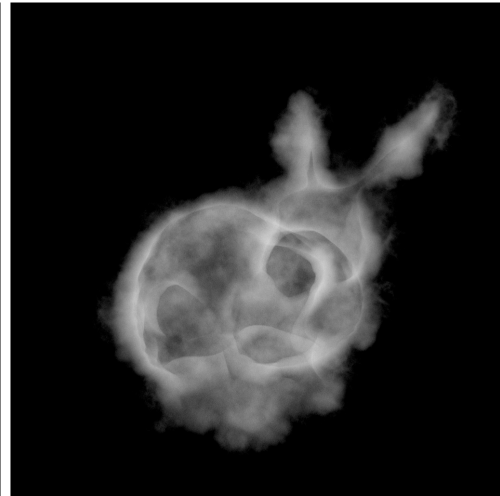
The majority of profiled scenarios, including the case of dense grids, are either generated or obtained in the form of grids with class set as Level Set. However, level sets are structured to represent the signed distance to the closest surface point in each voxel, with signs indicating if the voxel is located inside or outside. Thus, for clearer visualization as illustrated in Figure 4.2 on an example of a *crawler*, it was required to inverse the two halves of narrow-band values, using the *sdfToFogVolume* API method call. This makes it so that negative active voxels inside half of the narrow-band become instead a positive 0-1 ramp and inactive inside voxels become active with a constant approximate value of 1. Additionally,

Test scenario filenames	**Voxel Dim. (XYZ)**	**Structure lower internal and leaf**	**Leaf mem (%)**	**Actual mem (MB)**	**Dense eq.**
armadillo	1270x 1513x 1154	Internal(336x), Leaf(97,988x)	53.6%	213.2	8.2 GB
bunny	622x 615x 483	Internal(76x), Leaf(23,424x)	52%	52.4	704.8 MB
cloudBunny	577x 572x 438	Internal(73x), Leaf(66,212x)	52.4%	139.7	551.4 MB
crawler	2613x 505x 2143	Internal(1,059x), Leaf(558,446x)	56.7%	1177.6	10.53 GB
dense300	301x 301x 301	Internal(56x), Leaf(8,216x)	62.4%	20.6	104.0 MB
dense934	935x 935x 935	Internal(296x), Leaf(82,136x)	38.7%	179.5	3.0 GB
dragon	2017x 905x 1341	Internal(310x), Leaf(99,658x)	53.2%	215.8	9.1 GB
distr300	301x 301x 301	Internal(64x), Leaf(54,872x)	97.1%	116.3	104.0 MB
distr934	935x 935x 935	Internal(512x), Leaf(1,643,032x)	97.2%	3378.1	3.0 GB

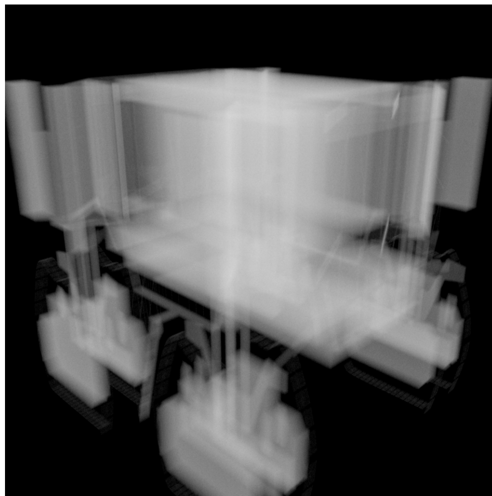
Table 4.1 All of the used test scenario files compared by memory footprint and structure. The structure column is shortened, providing only lower and leaf child counts, with root(1 x 8) and upper(8 x 32^3) already specified in Chapter 1.2.3



(a) Fog volume of bunny level set



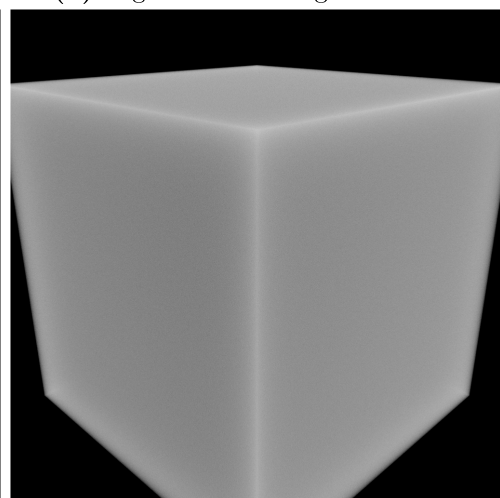
(b) Sparse bunny fog volume



(c) Fog volume of Crawler level set

(d) Fog volume of dragon level set

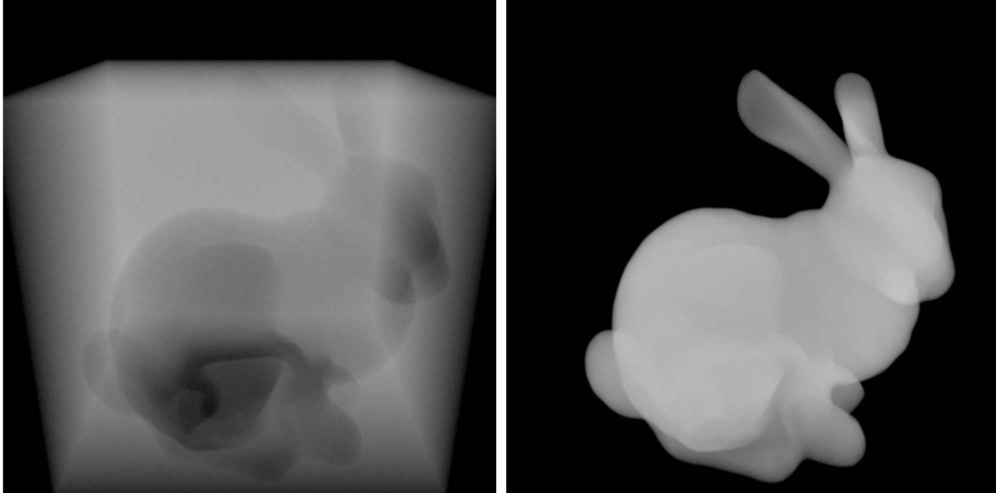
(e) Fog volume of armadillo level set



(f) Dense cube level set

Figure 4.1 Samples of test scenarios that are used.

the active exterior, including background value, becomes inactive with a constant value of 0.



(a) The before of bunny VDB LevelSet **(b)** The after of bunny VDB as FogVolume

Figure 4.2 Example of level set to fog volume conversion.

Fog volume (cloudBunny)

Similar to level sets, fog volumes also have a zero-to-one ramp in their narrow band, separating outside from inside regions and inactive outside voxels of background constant value. However, an inside region is active and has a value of one. Test scenarios include a single true fog volume file of *cloudBunny* with a higher variance in extinction coefficient than compared to converted level sets. This is easily explained by the conversion process for level sets, as all internal values are rewritten to a constant of 1. Due to a lack of true fog volumes in the test cases, this file serves as the only baseline in a truly sparse voxel representation case.

Dense fog volume (denseGrid300, distrGrid300)

In an attempt to get performance values for a densely packed voxel enclosing domain space of the same value, which was set to a standardized 5.0 float value for all dense files, a small executable specifically for creating artificial dense grids was added to one of the projects.

Test scenarios also include two dense fog volumes with varying floats in a random distribution from 4.5 to 5.5 floating point values.

This executable was used to create a set of dense grids, including ones of resolution 300 and 934, as can be seen in Figure 4.1 the voxel cube is filled with the same value. This cube is then transformed to sparse VDB and later to NanoVDB using the VDB conversion script. The bounding box for the generation of relevant volume is centered inside the cube, with zero-ed pivot coordinates. The bounding box spans to a half resolution in all directions, removing the need for additional transformations in the scene descriptors.

4.2 Case comparison

Selected cases are structured in a way that allows comparison of the developed plugin to the *base*, which utilizes dense grid representation, loaded from OpenVDB file format to NumPy dense. To that end, the files in the LevelSet class all undergo value inversion using the process described in section 4.1. The selected cases are generally split into:

- Comparison of both Mitsuba variants to scalar variant.
- Comparing LLVM dense to LLVM with our developed plugin.
- Lastly, CUDA dense compared with CUDA using the NanoVDB plugin.

Base Scalar vs LLVM / CUDA

This case comparison is used to give an approximation of the increase when compared to a scalar method. Unfortunately, as the benchmark values are recorded using the internal call to method *whos_str()*, an accurate memory measurement cannot be provided, as the scalar mode does not allocate memory recorded on the device. Instead, this case comparison uses the dense equivalent from the table 4.1, converted to MiB equivalent, to provide some approximate meaningful comparison baseline, even if not a completely precise one.

Base LLVM vs LLVM

LLVM case comparison is used to provide a reliable measurement baseline and comparison values on one of the Mitsuba variants. This case helps indicate a clear increase or decrease in performance and memory consumption, as opposed to the approximated values for memory used in the base scalar vs LLVM case described earlier.

Base CUDA vs CUDA

Similarly, to the LLVM vs LLVM case, the CUDA vs CUDA plugin case helps illustrate and draw clear conclusions without using approximated data. We are also drawing conclusions mainly on the results of variant-constrained results that were recorded, mainly due to these showing direct comparison within respective backends, as well as because of the motivation for this thesis being the CUDA-availability of data structures, as NanoVDB being made available.

4.3 Benchmarks

All of the benchmark samples were recorded inside of the Dockerized, isolated single Jupyter kernel, individual for each sensor and scene launch to ensure environment isolation and better, more consistent testing results.

Final samples per pixel were set up to be consistent across both baseline rendering as well as comparison images. The value of *spp* was set at 1024, with rendering resolution also being 1024x1024 square sensor for each of 8 sensors.

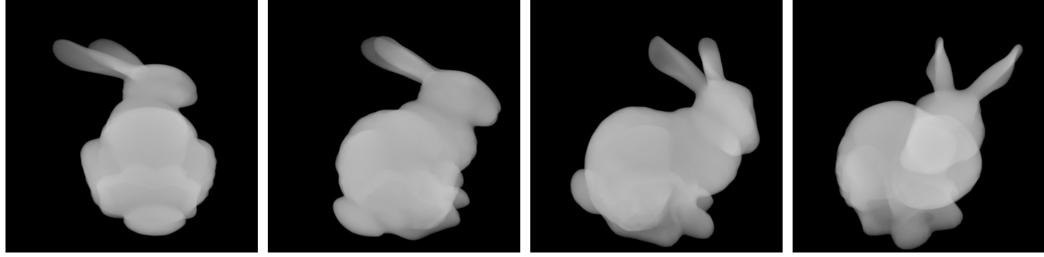


Figure 4.3 Four of turntable shots, illustrating the capture process.

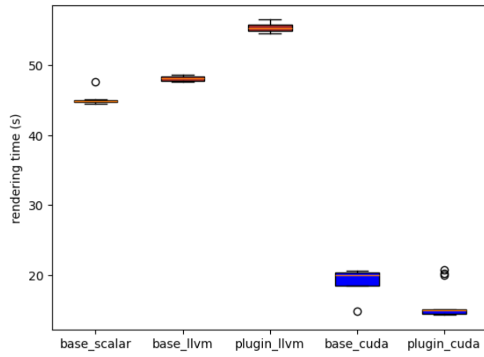
Additionally, all scenes were rendered multiple times (at least 4), with both average and maximum measurements provided in relevant tables. All images have a standardized set of sensors and a pre-set order of these sensors to be recorded, operating similarly to a turntable. This can be clearly illustrated in Figure 4.3

To record performance values of rendering time, a *Time* library inside of the Python kernel was used. To isolate performance to only the Mitsuba rendering phase, the command *mi.render()* alone is enclosed in time tracking calls from *Time* library. Recording of the relevant memory measures was conducted using the *mitsuba.render()* internal debugging call to Python bound *whos_str()* method.

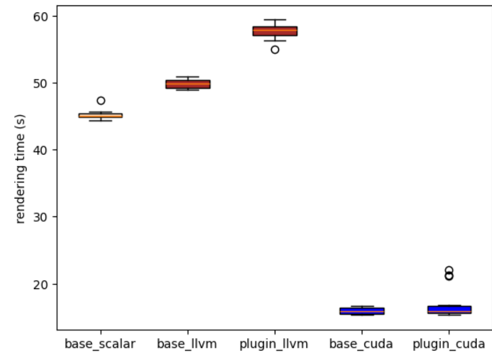
Notably, to verify the file resolution grid limit on dense test scenarios for loads on CUDA/GPU, the *cuda_rgb* variant was attempted instead of *cuda_ad_rgb*, which was initially used for reasons of code compatibility checks. That is because, from one of the recent Mitsuba 3 repository issues, it was known that an additional GPU footprint is added on AD variants, in the form of a large wavefront buffer created that gets stored in device memory. However, there was quite a bit of discrepancy in expected results for memory allocation on the target device as compared to observed ones, on both variants. Thus, in our experiments, the reported results for the baseline of *scalar_rgb* are mainly based on the OpenVDB read equivalent of dense device memory as a reference number, despite it being sometimes a much smaller baseline number.

To support that observation of allocated memory not matching, a separate set of experiments was conducted on the NVIDIA RTX 2080 Ti with 12 GB of memory capacity. It was observed that AD variants on the CUDA backend and files of 3GB equivalent representation were still able to fit into GPU memory, while samples of 6 GB equivalent were already failing to create and maintain a computation kernel. This is further supported by examples of observed results for a file of dense equivalent in 6GB resulting in a Mitsuba reported device memory of 8.002GiB, while monitoring CLI command *nvidia-smi* reports a total memory utilization by python process as 14.512GB. We should mention that the upper GPU limit should never have been theoretically reached even on RTX 3080 Ti and all setup processes in for scalar setups were done specifically to eliminate any additional memory allocation on the device as described in section 3.6.

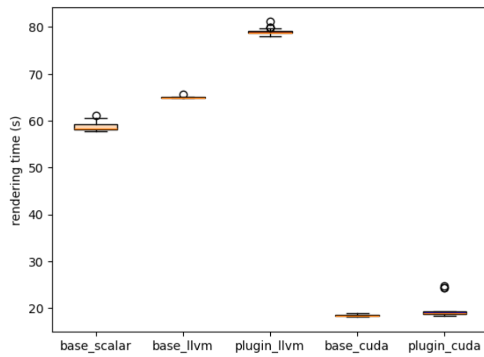
Upon inspection of the results from the following benchmarks for all testing case comparisons, there seems to be a consistent two-time increase in the allocated size on all backends, based on the reported raw file size in NanoVDB format. This is unforeseen because a double memory allocator inside the plugin was removed at the last minute, which was at the start of buffer creation. This matter will need to be discussed further, but unfortunately, due to a tightening time constraint for this thesis deadline, we can only report on it here as a last-moment observation.



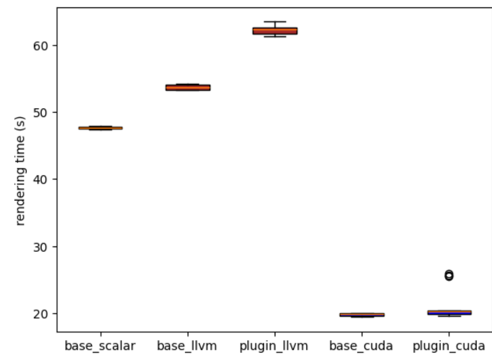
(a) Render time for armadillo.nvdb



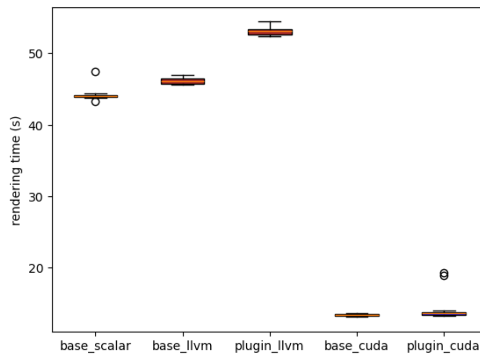
(b) Render time for bunny.nvdb



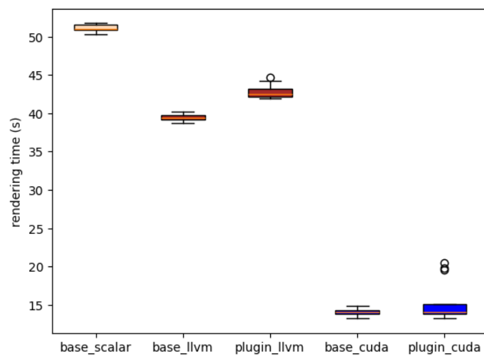
(c) Render time for cloudBunny.nvdb



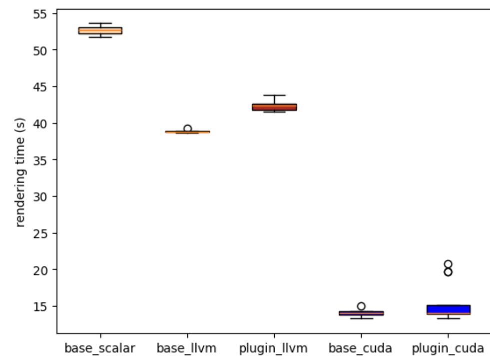
(d) Render time for crawler.nvdb



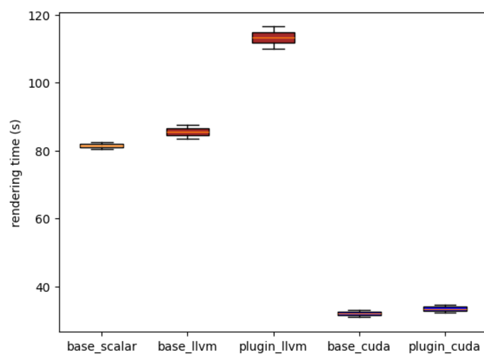
(e) Render time for dragon.nvdb



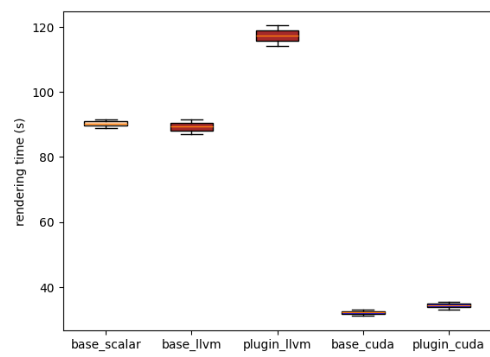
(f) Render time for denseGrid300.nvdb



(g) Render time for denseGrid934.nvdb



(h) Render time for distrGrid300.nvdb



(i) Render time for distrGrid934.nvdb

Figure 4.4 Render time box plots for all base and plugin variants.

Dense Scalar vs Plugin CUDA

File	Base(sec)	cuda Avg.	cuda Max.	Raw Diff	%Diff
armdl	45.11	15.55	20.68	-29.55	-65.52
bunny	45.27	16.79	22.05	-28.48	-62.91
cloud	58.78	19.67	24.77	-39.11	-66.54
crawler	47.67	20.91	25.86	-26.77	-56.15
grid300	51.16	14.92	20.52	-36.23	-70.82
grid934	52.60	15.00	20.75	-37.60	-71.49
dragon	44.25	14.10	19.23	-30.14	-68.12
distr300	81.48	33.35	34.44	-48.13	-59.07
distr934	90.24	34.28	35.41	-55.96	-62.01

Table 4.2 Render time benchmark for Scalar base vs CUDA plugin

File	Base(MiB)	cuda Avg.	cuda Max.	Raw Diff	%Diff
armdl	8459.26	288.10	288.10	-8171.16	-96.59
bunny	704.81	96.07	96.08	-608.74	-86.37
cloud	551.45	288.10	288.10	-263.35	-47.76
crawler	10786.82	2079.74	2079.74	-8707.07	-80.72
grid300	104.03	64.07	64.08	-39.96	-38.41
grid934	3118.08	288.10	288.10	-2829.98	-90.76
dragon	9337.86	288.10	288.10	-9049.76	-96.91
distr300	104.03	160.10	160.10	56.07	53.90
distr934	3118.08	4127.74	4127.74	1009.66	32.38

Table 4.3 Memory benchmark for Scalar base vs CUDA plugin.
The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values, in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase.

Observation

It is easy to observe a significant decrease in render times across all files. While interesting, this metric is not as useful to us because of the comparison case being too drastic, as we are in essence, comparing completely different approaches.

What is, however, useful to us is the memory benchmark results that show a significant drop in device GPU utilization, even when compared to equivalent dense values, as they are cutting off a significant chunk of memory footprint, when compared to the device metric. Interestingly, there is no variation in the reported numbers for memory footprints. What is even more noteworthy is the absence of variation in CUDA-allocated memory, showing that JIT *load_aligned* calls are allocating memory in blocks of fixed size.

Dense Scalar vs Plugin LLVM

File	Base(sec)	llvm Avg.	llvm Max.	Raw Diff	%Diff
armdl	45.11	55.44	56.56	10.34	22.92
bunny	45.27	57.70	59.42	12.43	27.45
cloud	58.78	79.02	81.13	20.24	34.43
crawler	47.67	62.29	63.47	14.62	30.66
grid300	51.16	42.78	44.70	-8.37	-16.37
grid934	52.60	42.33	43.80	-10.27	-19.53
dragon	44.25	53.08	54.42	8.83	19.96
distr300	81.48	113.36	116.58	31.88	39.13
distr934	90.24	117.22	120.36	26.98	29.90

Table 4.4 Render time benchmark for Scalar base vs LLVM plugin

File	Base(MiB)	llvm Avg.	llvm Max.	Raw Diff	%Diff
armdl	8459.26	442.50	442.50	-8016.76	-94.77
bunny	704.81	120.80	120.80	-584.01	-82.86
cloud	551.45	295.60	295.60	-255.85	-46.40
crawler	10786.82	2372.61	2372.61	-8414.21	-78.00
grid300	104.03	57.35	57.35	-46.68	-44.87
grid934	3118.08	375.10	375.10	-2742.98	-87.97
dragon	9337.86	447.70	447.70	-8890.16	-95.21
distr300	104.03	248.70	248.70	144.67	139.07
distr934	3118.08	6771.71	6771.71	3653.63	117.18

Table 4.5 Memory benchmark for Scalar base vs LLVM plugin.
The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values, in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase.

Observation

Interestingly, the LLVM experiences a slowdown in all tested scenarios, except on densely populated artificial grids. This could be attributed to the fact that our dense artificial grids contain only a single value. Additionally, the shape of the object is more simple, meaning the leaf nodes, closer to the outside, represent less detail, resulting in a speed-up when compared to a scalar dense grid. This is supported by the rough calculation based on the number of active voxels that are not in leaf nodes, or by looking up the footprint percentage for the leaf nodes from Table 4.1. In this case, the *dense300* is an outlier from the leaf% statistic. Once again, these render time benchmarks can't be used for conclusive evidence due to using scalar variants, but they still provide meaningful statistics.

Dense LLVM vs Plugin LLVM

File	Base(sec)	llvm Avg.	llvm Max.	Raw Diff	%Diff
armdl	48.15	55.44	56.56	7.30	15.15
bunny	49.89	57.70	59.42	7.81	15.66
cloud	65.06	79.02	81.13	13.96	21.45
crawler	53.72	62.29	63.47	8.57	15.95
grid300	39.43	42.78	44.70	3.35	8.49
grid934	38.87	42.33	43.80	3.46	8.89
dragon	46.19	53.08	54.42	6.88	14.90
distr300	85.59	113.36	116.58	27.76	32.44
distr934	89.25	117.22	120.36	27.98	31.35

Table 4.6 Render time benchmark for LLVM base vs LLVM plugin.

File	Base(MiB)	llvm Avg.	llvm Max.	Raw Diff	%Diff
armdl	8455.17	442.50	442.50	-8012.67	-94.77
bunny	717.10	120.80	120.80	-596.30	-83.15
cloud	564.30	295.60	295.60	-268.70	-47.62
crawler	10772.48	2372.61	2372.61	-8399.87	-77.98
grid300	119.00	57.35	57.35	-61.65	-51.81
grid934	3124.22	375.10	375.10	-2749.12	-87.99
dragon	9331.71	447.70	447.70	-8884.01	-95.20
distr300	119.00	248.70	248.70	129.70	108.99
distr934	3124.22	6771.71	6771.71	3647.49	116.75

Table 4.7 Memory benchmark for LLVM base vs LLVM plugin.
The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase.

Observation

From the benchmarked results it is obvious that LLVM is experiencing a slowdown in all files, with the ones in dense uniform value grids being less prominent, and ones with highly varying bunny cloud more prominent. Additionally, the maximum increase does not seem to be related to the size of the reported file, as *crawler* is bigger than *cloud* both in actual size and dense equivalent, but the reported render time for it is lower on all variants.
Separately, the memory benchmarks perform as expected, with the exception of artificial random distribution dense grids. On other files, the memory footprint becomes noticeably smaller due to NanoVDB packing being more efficient. It occupies a contiguous memory block for its underlying fixed hierarchical structure.

Dense CUDA vs Plugin CUDA

File	Base(sec)	cuda Avg.	cuda Max.	Raw Diff	%Diff
armdl	18.83	15.55	20.68	-3.28	-17.40
bunny	15.97	16.79	22.05	0.83	5.17
cloud	18.41	19.67	24.77	1.26	6.84
crawler	19.78	20.91	25.86	1.13	5.70
grid300	14.03	14.92	20.52	0.89	6.37
grid934	14.08	15.00	20.75	0.91	6.47
dragon	13.35	14.10	19.23	0.75	5.63
distr300	31.93	33.35	34.44	1.42	4.44
distr934	32.12	34.28	35.41	2.16	6.74

Table 4.8 Render time benchmark for CUDA base vs CUDA plugin.

File	Base(MiB)	cuda Avg.	cuda Max.	Raw Diff	%Diff
armdl	16414.72	288.10	288.10	-16126.62	-98.24
bunny	1055.74	96.07	96.08	-959.67	-90.90
cloud	1055.74	288.10	288.10	-767.64	-72.71
crawler	16414.72	2079.74	2079.74	-14334.98	-87.33
grid300	160.10	64.07	64.08	-96.03	-59.98
grid934	4127.74	288.10	288.10	-3839.64	-93.02
dragon	16414.72	288.10	288.10	-16126.62	-98.24
distr300	160.10	160.10	160.10	0.00	0.00
distr934	4127.74	4127.74	4127.74	0.00	0.00

Table 4.9 Memory benchmark for CUDA base vs CUDA plugin.
The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values, in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase.

Observation

The observed results on CUDA in combination with the render time box plots in Figure 4.4 show some slowdown on all files on the CUDA variant, with an outlier improvement over 8 sensor launches only with file *armadillo*. The overall trend is that the rendering time has insignificantly slowed down, with outliers present in less than half of the samples, showing a significant slowdown. On a separate note, the renders seem to take longer for the artificially distributed grids, probably linked to the growing size of their file. Memory benchmarks consistently show improvement in the amount of device memory. Similar to the case of Scalar vs CUDA, the allocated memory is the same on multiple different files, solidifying that the CUDA allocation pattern is indeed block-based.

4.4 CUDA image comparison

This section provides SSIM as well as MSE method image comparison results to illustrate the attained accuracy of the developed plugin. The results show a very high degree of accuracy, with the average MSE values not increasing past 10 units of squared error as well as the average SSIM not exceeding 0.97. However, as the artificial grid results show the MSE being at around 0.00 units throughout the entirety of the testing and result recording phases, we have the right to assume that something does indeed go wrong in the conversion process, either OpenVDB to PNano or OpenVDB to Dense. Upon closer inspection, as shown in Figure 4.5 that pinpoints the multiplied absolute image difference of a worst-case file for *bunny*, it can be seen as approximately 5-10 pixel displacement in the XY axis, which must have resulted from the NumPy dense comparison image conversion process, as we manually set the bounding box. This is not a direct result of a fault in the plugin development, but it will be investigated further before the release.

It is important to note that for the generation of base images for image comparison, only the *cuda_ad_rgb* variant was used. That is because the results needed to be compared to the same available implementation on the GPU backend as it meets the main goal of the thesis. All images were matched pairwise using their sensor IDs and compared using either MSE or SSIM comparison algorithms on the OpenCV library in Python.

File	**MSE average**	**MSE max**	**SSIM average**	**SSIM min**
armdl	2.63	5.07	0.99	0.99
bunny	9.82	21.50	0.99	0.98
cloud	4.57	8.51	0.98	0.97
crawler	8.21	9.68	0.97	0.97
grid300	0.00	0.00	0.99	0.99
grid934	0.00	0.00	0.99	0.99
dragon	2.33	4.06	0.99	0.99
distr300	0.00	0.00	0.99	0.99
distr934	0.00	0.00	0.99	0.98

Table 4.10 MSE and SSIM measurements for all files on CUDA variant

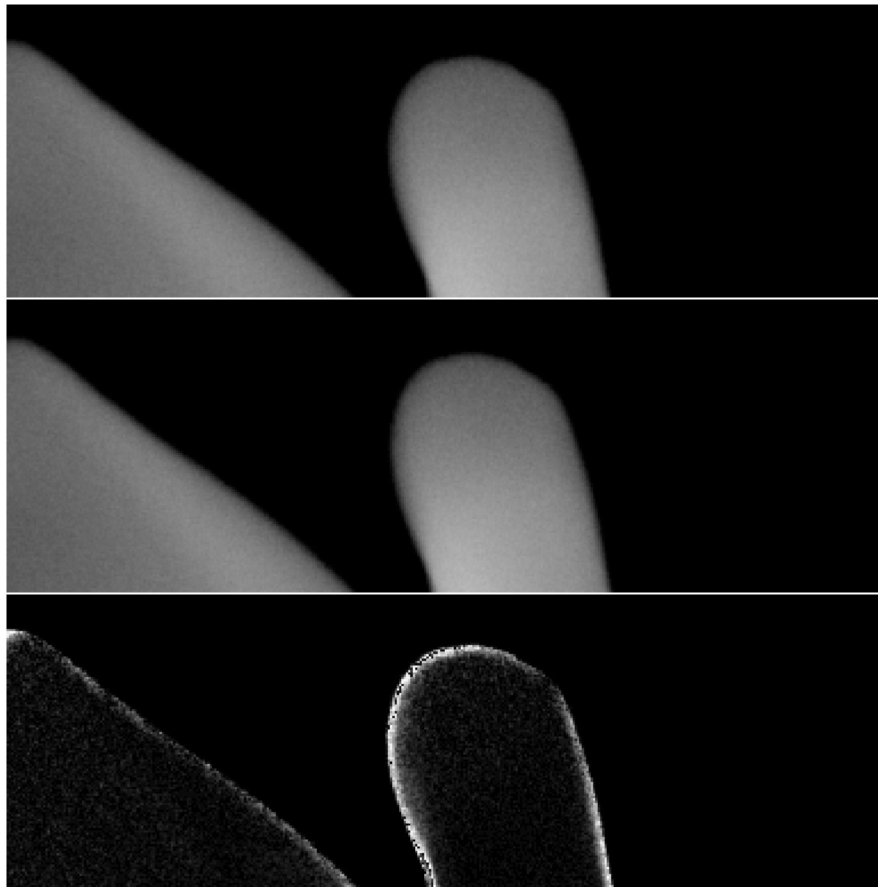


Figure 4.5 Several pixels of displacement difference.

Discussion

Versioning

The compiler settings for Mitsuba 3 specifically require adding a Clang compiler with supported libraries libc++-dev and libc++abi-dev since it does not support GCC's libstdc++. Most of our required packages are installed using Linux's Advanced Package Tool, or apt for short, because of ease of use and increased availability. The only exception from this is the LLVM Clang itself, which is installed from the official repository and checked out at version 17.0.5.

Future work

Due to a failure to include a larger file size, the next step for this is investigating the nature of this behavior and potentially opening issue tickets in the relevant development repositories. As the implementation is limited to only monochromatic float grids right now, the next step would be to add support for colored grids as well as supporting other grid types. The ultimate goal would be the inclusion of all types of grids of all sizes, with the later opening of a merging pull request.

Shortcomings

As one of the shortcomings of this thesis, there is one oversight that forced us to exclude several bigger files from being included in the results. During the experimental setup with the different available GPUs, several files consistently failed in the kernel creation stage. This happened for all image renderings on both CUDA and LLVM variants. That includes our initially included Disney cloud as well as dense representation grids above 1174 resolution. This issue was further observed by experimenting on a local variant of unit-tested PNanoVDB implementation, which led to the conclusion there is something wrong with reads that exceed a certain file size/resolution. An interesting observation was, however, made that the issue was caused by the unchanged variant of the PNanoVDB header file on the read method for UInt64 reads. Extensive testing and both compiler and LLVM version matching did not produce any significant behavior changes for the observed code, neither in the produced deliverable nor on the original header. The best assumption right now on this issue is that a newer version of PNanoVDB changes this behavior, but as this was not fixable within the time limit left on the thesis, our effort was better spent on producing meaningful results.

Separately, due to a tightening time constraint, our current version can only support a monochromatic value read from the NanoVDB file, with some improvements possible in the near future, outside of this thesis's scope. Additionally, because of heavy templating, our version of the NanoVDB plugin does seem to work natively with multiple vectorization backends but is limited to the x32 assumption, which, as it later turns out, is an in-built limitation of the Mitsuba / DrJIT itself.

Conclusion

In this thesis, we provide a brief descriptive survey of the data structures used for volumetric object representation. Using that, we illustrate the actual degree of memory issue's importance. OpenVDB and its NVIDIA-produced derivative structure, NanoVDB, are prioritized as examples of effective sparse hierarchical structures that can provide solutions to some of these memory efficiency problems. Through limiting the scope to a monochromatic, float grid, a vectorized implementation of a header-only PNanoVDB to DrJIT templated types is presented, with the attached described unit testing procedure in DocTest. The unit tests are there to ensure the reliability of the produced results as well as overall code correctness.

Results were rendered through a high amount of samples-per-pixel on a sufficient resolution on a remote compute cluster to ensure the reproducibility and accuracy of the observations as well as limiting of resulting image noise.

Through the collected renderings and benchmarked results on the Docker-isolated Linux system environment setup on both Mitsuba variants of CUDA and LLVM, a relatively low threshold of result dissimilarity is observed. The potential problem introducing the error is identified and will be dealt with as indicated in the future work section of the discussion.

We believe that considering the limiting scope of the initial assumptions, the resulting deliverable in the form of a rendering plugin, that supports NanoVDB format on both CUDA and LLVM backends has been achieved. The results are supported by the attached observations and speed/memory performance benchmarks when compared to the baseline combination of dense grids and the original Mitsuba volumetric grid plugin. The main flaws include the undetermined exact limit on the size of the included grid as well as a question of doubly allocated memory, which was noticed in the last hours before the submission.

Bibliography

1. Fascione, Luca; Hanika, Johannes; Heckenberg, Daniel; Kulla, Christopher; Droske, Marc; Schwarzhaupt, Jorge. Path tracing in production: part 1: modern path tracing. In: *ACM SIGGRAPH 2019 Courses*. Los Angeles, California: Association for Computing Machinery, 2019. SIGGRAPH '19. isbn 9781450363075. Available from doi: 10.1145/3305366.3328079.
2. Christensen, Per; Jarosz, Wojciech. The Path to Path-Traced Movies. *Foundations and Trends® in Computer Graphics and Vision*. 2016, vol. 10, pp. 103–175. Available from doi: 10.1561/0600000073.
3. Meagher, Donald. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*. 1982, vol. 19, no. 2, pp. 129–147. issn 0146-664X. Available from doi: https://doi.org/10.1016/0146-664X(82)90104-6.
4. Laine, Samuli; Karras, Tero. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation*. 2010, vol. 2, no. 6.
5. Theussl, Thomas; Moller, Torsten; Groller, Meister Eduard. Optimal regular volume sampling. In: *Proceedings Visualization, 2001. VIS'01*. IEEE, 2001, pp. 91–546.
6. Gárate, Matías. Voxel Datacubes for 3D Visualization in Blender. *Publications of the Astronomical Society of the Pacific*. 2017, vol. 129, no. 975, p. 058010. Available from doi: 10.1088/1538-3873/129/975/058010.
7. Rodríguez, M. Balsa; Gobbetti, E.; Guitián, J.A. Iglesias; Makhinya, M.; Marton, F.; Pajarola, R.; Suter, S.K. State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. *Computer Graphics Forum*. 2014. issn 1467-8659. Available from doi: 10.1111/cgf.12280.
8. Carvalho, Carlos. The gap between processor and memory speeds. In: *Proc. of IEEE International Conference on Control and Automation*. 2002, vol. 5000, p. 15000. No. 10000.
9. Museth, Ken. VDB: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics (TOG)*. 2013, vol. 32, no. 3, pp. 1–22.
10. Crassin, Cyril; Neyret, Fabrice; Lefebvre, Sylvain; Eisemann, Elmar. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, pp. 15–22.
11. Bayer, R.; Mccreight, E. M. Organization and maintenance of large ordered indexes. *Acta Inf.* 1972, vol. 1, no. 3, pp. 173–189. issn 0001-5903. Available from doi: 10.1007/BF00288683.
12. Museth, Ken. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In: *ACM SIGGRAPH 2021 Talks*. Virtual Event, USA: Association for Computing Machinery, 2021. SIGGRAPH '21. isbn 9781450383738. Available from doi: 10.1145/3450623.3464653.

13. Nimier-David, Merlin; Vicini, Delio; Zeltner, Tizian; Jakob, Wenzel. Mitsuba 2: a retargetable forward and inverse renderer. *ACM Trans. Graph.* 2019, vol. 38, no. 6. ISSN 0730-0301. Available from DOI: 10.1145/3355089.3356498.

14. Jakob, Wenzel; Speierer, Sébastien; Roussel, Nicolas; Vicini, Delio. Dr.Jit: A Just-In-Time Compiler for Differentiable Rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)*. 2022, vol. 41, no. 4. Available from DOI: 10.1145/3528223.3530099.

List of Figures

1.1 1D structure example (simplified) of voxel data packing in VDB tree. Going from top to bottom, there is a hashmap Root node, two Internal nodes, and block-size Leaf nodes. Both internal nodes contain two bit-masks that are integral parts of VDB tree traversal. Two bit-masks are the active and topology masks with mValue-Mask and mChildMask for the internal node structure, respectively. Voxels are represented in leaves in red. White nodes at the bottom are the compressed leaf blocks with only active (interesting) values present. All nodes have their own fixed branching factor, aside from the root node. It is noteworthy that the fixed branching factor is 2^N because OpenVDB tree traversal depends on fast bit operations. 13

2.1 A schematic flow diagram for illustrating the conducted work and its general structure. This diagram outlines major sections of our process on a high level. The work is split into several steps, out of which distinguishable are major three. Starting from left to right, the VDB conversion stage processes inputs of OpenVDB to both sparse NanoVDB and dense grid NumPy. Secondly, there is a Mitsuba plugin involved in the rendering stage that loads the relevant converted file format into the plugin and outputs the rendered image. Lastly, in the comparison stage, both accuracy and performance are measured using image metrics as well as benchmarks. Additionally, the dashed lines indicate sparse data while solid lines indicate dense volumes. 18

3.1 Example of a simple scene in both XML and Python 28
3.2 Use of the standard gridvolume plugin parameters 29
3.3 Use of the our grid_pnano plugin parameters 29
3.4 Tree type defines and grid pointer casting 31
3.5 Conversion to NanoVDB and writing to file 32
3.6 PNanoVDB method pnanovdb_root_find_tile 33
3.7 DrJIT translated NanoVDB method drjit_root_find_tile 34
3.8 Adding templating and masking during Mitsuba integration of PNanoVDB 36
3.9 Optimizing memory allocation 37

4.1 Samples of test scenarios that are used. 41
4.2 Example of level set to fog volume conversion. 42
4.3 Four of turntable shots, illustrating the capture process. 44
4.4 Render time box plots for all base and plugin variants. 46
4.5 Several pixels of displacement difference. 52

List of Tables

2.1 Selected feature comparison table for data structures that are used in the thesis. Column-wise, highlighted in bold, are the minority outliers in each data structure column, done only for increased visibility. 20

4.1 All of the used test scenario files compared by memory footprint and structure. The structure column is shortened, providing only lower and leaf child counts, with root(1 x 8) and upper(8 x 32^3) already specified in Chapter 1.2.3 40
4.2 Render time benchmark for Scalar base vs CUDA plugin 47
4.3 Memory benchmark for Scalar base vs CUDA plugin. The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values, in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase. 47
4.4 Render time benchmark for Scalar base vs LLVM plugin 48
4.5 Memory benchmark for Scalar base vs LLVM plugin. The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values, in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase. 48
4.6 Render time benchmark for LLVM base vs LLVM plugin. 49
4.7 Memory benchmark for LLVM base vs LLVM plugin. The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase. 49
4.8 Render time benchmark for CUDA base vs CUDA plugin. 50
4.9 Memory benchmark for CUDA base vs CUDA plugin. The file column represents shortened names for tested scenario files. The columns represent values and relative comparison for respective values, in MiB, all except the last column %Diff, which represents the percentile difference decrease or increase. 50
4.10 MSE and SSIM measurements for all files on CUDA variant . . . 51

A Attachments

A.1 Electronic Project Attachment

The electronic attachment includes the project folders and two Docker files for quick Docker installation of the entire setup.

- The plugin/ folder contains the relevant files that were added to mitsuba3 as well as a quick install script that places them in the correct folders This includes:
  - nanovdb/ - folder with NanoVDB headers, util, and IO mainly. Can also be taken cleanly from the OpenVDB repository
  - vdb_header/ - folder with impl_drjit.h that holds our implementation
  - grid_pnano.cpp - our added Mitsuba 3 plugin code
  - CMakeLists.txt - modified Cmake file from src/volumes/ folder in Mitsuba
- The nano_jit_prelim - folder with pre-checkout git repository contains converted scripts, DocTest unit_tests, dense generator script.
- The Jupyter_notebooks - folder with all Jupyter notebooks that were used for automated and manual rendering of the results.
- The text in the entries may be of any length.
- Dockerfile - a docker file with all required dependencies and prerequisites listed
- docker-compose.yml - a docker-compose file for easier setup with CUDA

A.2 User Documentation

A small .md file with instructions on replicating the setup, either using the provided Dockerfile in A.1 or checking out the GIT repositories. Due to a large build folder size, we couldn't include them physically, so all attempts to replicate our setup should be built from scratch.

A.3 Prerequisites Versioning

Another small .md file with listed prerequisites that are relevant/required by some of the projects to function correctly during the build process. We advise to use the listed versions.