RENDERING THREE-DIMENSIONAL SCENES WITH TETRAHEDRAL MESHES

A DISSERTATION SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

By
Aytek Aman
July 2022

RENDERING THREE-DIMENSIONAL SCENES WITH TETRA-
HEDRAL MESHES

By Aytek Aman

July 2022

We certify that we have read this dissertation and that in our opinion it is fully
adequate, in scope and in quality, as a dissertation for the degree of Doctor of
Philosophy.

Uğur Güdükbay(Advisor)

Özgür Ulusoy

Özcan Ozturk

Ahmet Oğuz Akyüz

Yusuf Sahillioğlu

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ii

ABSTRACT

RENDERING THREE-DIMENSIONAL SCENES WITH TETRAHEDRAL MESHES

Aytek Aman

Ph.D. in Computer Engineering

Advisor: Uğur Güdükbay

July 2022

We propose compact and efficient tetrahedral mesh representations to improve the ray-tracing performance. We reorder tetrahedral mesh data using a space-filling curve to improve cache locality. Most importantly, we propose efficient ray traversal algorithms. We provide details of the regular ray tracing operations on tetrahedral meshes and the Graphics Processing Unit (GPU) implementation of our traversal method. We demonstrate our findings through a set of comprehensive experiments. Our method outperforms existing tetrahedral mesh-based traversal methods and yields comparable results to the traversal methods based on the state-of-the-art acceleration structures such as k-dimensional (k-d) tree and Bounding Volume Hierarchy (BVH) in terms of speed. Storage-wise, our method uses less memory than its tetrahedral mesh-based counterparts, thus allowing larger scenes to be rendered on the GPU. We also describe additional applications of our technique specifically for volume rendering, two-level hybrid acceleration structures for animation purposes, and point queries in two-dimensional (2-D) and three-dimensional (3-D) triangulations. Finally, we present a practical method to tetrahedralize very large scenes.

Keywords: ray tracing, ray-surface intersection, acceleration structure, tetrahedral mesh, Bounding Volume Hierarchy (BVH), k-dimensional (k-d) tree.

ÖZET

ÜÇ BOYUTLU SAHNELERİN DÖRTYÜZLÜ ÖRGÜLER İLE GÖRSELLEŞTİRİLMESİ

Aytek Aman
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Uğur Güdükbay
Temmuz 2022

Işın izleme performansını artırmak amacıyla etkin ve kompak dörtyüzlü örgü yöntemleri öneriyoruz. Dörtyüzlü örgü bilgisini, önbellek verimini artırmak amacıyla uzay eğrisi kullanarak sıralıyoruz. En önemlisi, çok hızlı çalışan ışın takip yöntemleri sunuyoruz. Dörtyüzlü örgüleri ışın izleme işlemlerinde kullanabilmek için gerekli yöntemlerin detaylarını verip bu yöntemlerin grafik işlemci birimlerinde nasıl kullanılacağını tarif ediyoruz. Bulguları ve sonuçları kapsamlı deneylerle gösteriyoruz. Geliştirdiğimiz yöntemler, mevcut dörtyüzlü gezim yöntemlerinden daha hızlı çalışmaktadır. Bu yöntemler, aynı zamanda yaygın olarak kullanılan k-d ağacı ve sınırlayıcı hacim hiyerarşisi gibi ışın izleme hızlandırıcı yapılarına yakın bir performans sergilemektedir. Geliştirdiğimiz yöntemler, diğer dörtyüzlü tabanlı yöntemlere göre daha az bellek kullanmaktadır; böylelikle grafik işlemci ünitesi gibi belleğin kısıtlı olduğu yerlerde daha büyük sahnelerin görselleştirilmesine olanak sağlamaktadır. Bunlarla birlikte, yöntemlerimizin hacim görselleştirilmesi, animasyon uygulamaları için iki seviyeli hibrit hızlandırıcı yapılarına uyarlanması ve iki ve üç boyutlu üçgenlemelerde noktaların sorgularının yapılması gibi uygulamalarda nasıl kullanılabileceğini anlatıyoruz. Son olarak, çok büyük sahnelerin dörtyüzlemesinin yapılmasını sağlayan pratik bir yöntem sunuyoruz.

Anahtar sözcükler: Işın izleme, ışın-yüzey kesişimi, hızlandırma yapıları, dörtyüzlü örgü, sınırlayıcı hacim hiyerarşisi, k-d ağacı.

Acknowledgement

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Rendering is the process of generating a synthetic image of a scene. The scene refers to the collection of 3-D geometry, materials, light sources, and camera parameters. A rendering task is typically performed using the following methods.

- *Rasterization:* Rasterization works by projecting the primitives to the screen. The per-pixel depth buffer is used to determine the visibility of the pixels. Rasterization is used in many real-time applications due to its low computational cost. Rasterization is an object-space rendering method [2] since all primitives (objects) are projected to the screen and then rendered after visibility determination.

- *Ray tracing:* Ray tracing [3] works by tracing rays from the camera and accumulating the color information from the objects that are hit by these rays. When a ray hits a surface, other rays can be cast on the scene to produce effects such as reflections, refractions, and shadows (both soft and hard). Ray-tracing using only primary rays (rays originating from the camera) is called *ray casting* [4]. In ray casting, recursive rays for reflection and refraction are not used. Ray tracing is an image-space rendering method [2] since color information is calculated for each pixel using rays.

Figure 1.1: Images rendered using ray tracing [5] (left) and radiosity [6] (right).

1.1 Acceleration Structures for Ray Tracing

The core operation in the ray tracing algorithm is finding the closest ray-surface intersection. Ray-surface intersection calculations may take more than 95% of the computation time [7]. For this reason, the run-time efficiency of the intersection calculations dictates the ray-tracing algorithm's run-time efficiency. In order to speed up intersection calculations, the most common approach is to partition the scene so that the triangles are enclosed in different volumes. During ray-traversal, ray-triangle intersections tests can be avoided if the enclosing volume for a triangle does not intersect with the ray. Regular grids, octrees, k-d trees, and Bounding Volume Hierarchies (BVH) are commonly used for partitioning the scene. k-d trees and BVHs are the most preferred space partitioning structures for ray tracing, thanks to the recent advancements in the construction and traversal methods. k-d trees and BVHs are considered state of the art for fast ray tracing.

1.2 Tetrahedral Meshes as Acceleration Structures

Unlike common approaches, Lagae et al. [1] use tetrahedral meshes to render typical 3-D scenes. They tetrahedralize the space between objects in a constrained

manner where triangles in the scene geometry align with the triangles of the tetrahedral mesh. Figure 1.2 illustrates this idea.
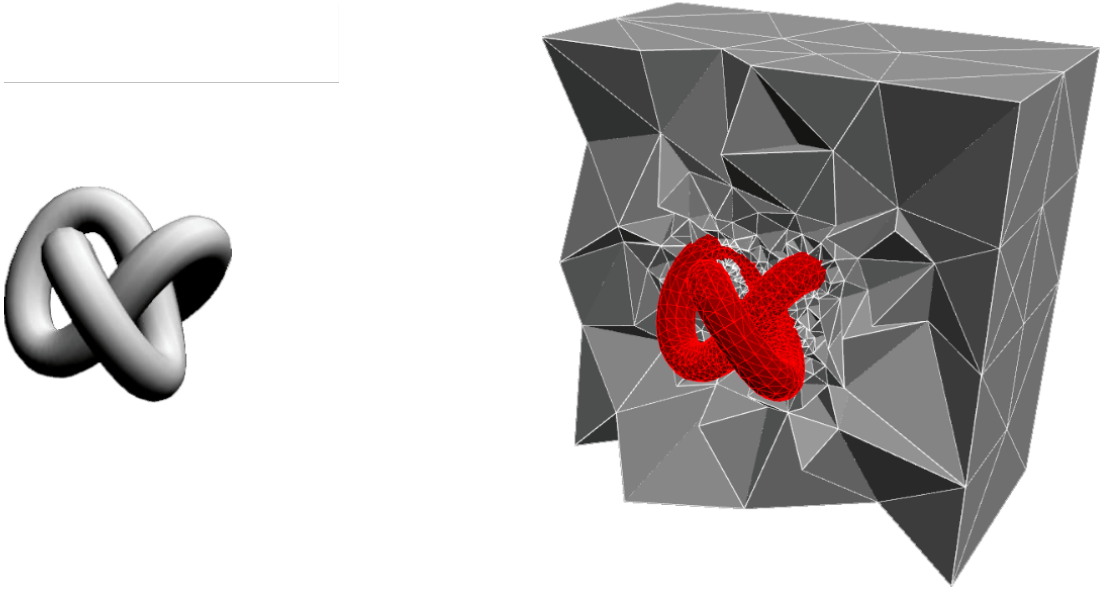
Figure 1.2: Tetrahedralization process: A torus knot (left). Tetrahedralization of the torus knot and empty space around it (right).

Once the empty space between the objects is tetrahedralized, it is possible to calculate ray-triangle intersections by traversing the tetrahedral mesh, where ray traversal can be terminated if a ray passes through a constrained face (a face that belongs to a scene geometry). Figure 1.3 illustrates this idea in 2-D inside a triangulation.
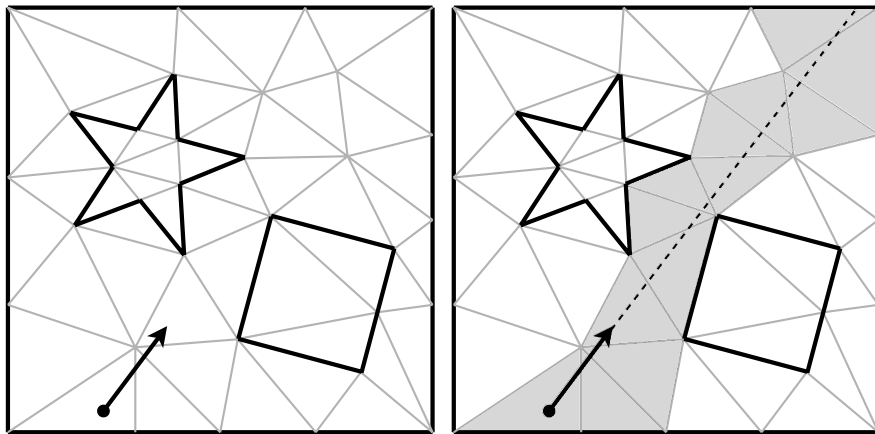
Figure 1.3: Ray tracing using a triangulation (adapted from Lagae et al [1]).

This approach has the following advantages:

A unified data structure for global illumination: In constrained tetrahedralizations, scene geometry coincides with the acceleration structure. This structure can improve the rendering performance since all related data to compute illumination resides in one unified structure.

Handling deforming geometry: Tetrahedral meshes allow some deformations to be applied without causing any changes in the topology. Therefore, the structure does not need to be rebuilt for such deformations.

Level of detail: Level-of-detail systems use similar representations with tetrahedralizations. Therefore, implementing a level of detail system in the tetrahedral mesh can be trivial.

Ray tracing on the GPU: Traversal algorithm for tetrahedral mesh does not require a stack. Thus tetrahedral mesh traversal methods are GPU friendly.

Adaptive and non-hierarchical structure: Tetrahedral mesh-based accelerator can adapt to scene geometry as in popular tree-based approaches, therefore not suffering from a *teapot in a stadium problem*. Since the structure is not hierarchical (merely a graph structure), traversal methods become simpler and can be made more efficient.

We use tetrahedral meshes in new and efficient ways to improve the rendering performance. We focus on the following topics.

Fast ray traversal: We propose a fast ray-traversal method where calculations are performed on a new coordinate system defined by the ray. The resulting implementation is efficient and requires only a few floating-point operations per tetrahedron.

Preserving vertex attributes: When a tetrahedral mesh is constructed out of scene geometry, triangles in the input mesh are directly embedded in the tetrahedralization. Efficiently preserving vertex attributes such as normals

4

and texture coordinates is essential to performing typical rendering operations. We investigate the approaches to preserve such attributes efficiently.

Global illumination methods: We adapt global illumination methods to tetrahedral meshes to improve the run-time performance of these methods. For this purpose, we will show how secondary rays can be traced efficiently using this approach.

Handling dynamic scenes: Preserving the structure of the tetrahedral mesh is essential to rendering animated scenes. When an object in the scene has rigid-body motion, the tetrahedral mesh may need to be updated. We will provide efficient methods to perform these updates in real-time.

Cache-efficient data structures: During traversal, a significant amount of time is spent fetching new data from memory. In order to improve the run-time performance of the ray traversal, we present new cache-efficient data structures for tetrahedral meshes.

Parallelizations on the CPU and GPU: We implement the parallelized version of these approaches on the CPU and GPU. For the GPU-based implementation, we present methods well-suited to GPU memory architectures.

Through experiments, we demonstrate that our compact structure and efficient traversal methods can provide considerable speed-up compared to the other tetrahedral mesh based traversal methods and comparable performance w.r.t the popular hierarchical acceleration structures. We also show that our compact and efficient structure could be adapted to other domains. We show that it can be used to render volumetric data where input can be convex or non-convex. We also adapt our method to the two-level accelerators, providing a way to render a dynamic scene using a tetrahedralization-BVH hybrid structure. We then describe two techniques to handle point-in-triangulation queries using our compact representation. Finally, we describe a practical approach to out-of-core tetrahedralization of huge meshes.

1.3 Publications

The following papers are published based on the methods and data representations developed within the context of this thesis work.

- *Fast Tetrahedral Mesh Traversal for Ray Tracing* [8]: This work describes the early work performed on compact tetrahedral mesh-based accelerators for ray tracing. This work was presented in *The Conference on High-Performance Graphics (HPG '17)* in Los Angeles, CA, in 2017.

- *Multi-level tetrahedralization-based accelerator for ray-tracing animated scenes* [9]: This work describes a two-level accelerator (tet-mesh and BVH hybrid) suited for ray-tracing dynamic scenes. This work was published in the journal *Computer Animation and Virtual Worlds* in 2021.

- *Compact tetrahedralization-based acceleration structures for ray tracing* [10]: This work describes compact tetrahedral mesh-based accelerators and efficient traversal methods suited for ray-tracing. This work is accepted for publication in *Journal of Visualization* in 2022.

- *Ray-traced Shell Traversal of Tetrahedral Meshes for Direct Volume Visualization* [11]: This work describes a volume rendering method using our compact tetrahedral mesh-based accelerators and efficient traversal methods. This work is presented in *IEEE Visualization Conference (IEEE VIS 21)* and appeared as a short paper in 2021.

- *Out-of-core Constrained Delaunay Tetrahedralizations for Large Scenes* [12]: This work describes a practical approach to out-of-core tetrahedralizations of huge meshes. This work was presented in *Numerical Geometry, Grid Generation and Scientific Computing (NUMGRID 2020)* and appeared in conference proceedings published in 2021.

1.4   Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 gives a detailed summary of the related work on ray tracing accelerators, tetrahedral mesh construction, and traversal and volume rendering methods. Chapter 3 describes the proposed approach, an efficient and compact tetrahedral mesh representation, and a fast traversal method for this structure. We also describe tetrahedral mesh-based accelerator construction in detail and provide valuable techniques and optimization to adapt it to a typical ray-tracing pipeline. Chapter 4 describes the application of our methods to the other domains. These are volume rendering, two-level hybrid accelerators (tetrahedral mesh - BVH hybrid), and point location queries in triangulation. This chapter also describes a practical out-of-core tetrahedralization method for huge meshes. Chapter 5 illustrates the impact of the results, highlights the limitation of our approach, and provides potential future work directions. Appendix A describes the ray-tracing toolkit developed to support this research work in detail.

Chapter 2

Background and Related Work

The most common approach to speed up ray-surface intersection calculations in ray tracing is to use spatial subdivision structures that partition the scene to enclose the polygons in different volumes. Ray tracing algorithms can avoid ray-triangle intersection tests if the enclosing volume for a triangle does not intersect with the ray. Popular acceleration structures are regular grids, octrees, Bounding Volume Hierarchy (BVH), and k-dimensional (k-d) tree. BVH and k-d tree are the most preferred acceleration structures for ray tracing, thanks to the recent advancements in the construction and traversal methods.

A recent alternative to accelerate ray-surface intersection calculations is to use tetrahedralizations. A tetrahedral mesh is a three-dimensional (3D) structure that partitions the 3D space into tetrahedra. Constrained tetrahedralizations are a special case of tetrahedralizations that consider the input geometry. In the resulting mesh, the components of the input geometry, such as faces, line segments, and points, are preserved. Similar to their two-dimensional (2D) counterparts, tetrahedralizations can be constructed in such a way that they exhibit Delaunay property; i.e., tetrahedra are close to regular. There are three types of constrained tetrahedralizations: *Conforming Delaunay Tetrahedralization*, *Constrained Delaunay Tetrahedralization*, and *Quality Delaunay Tetrahedralization* [1].

Lagae and Dutré [1] use constrained tetrahedral meshes for rendering typical 3D scenes. They tetrahedralize the space between objects in a constrained manner where the triangles in the scene geometry align with the triangles of the tetrahedral mesh. Then, they calculate ray-triangle intersections by traversing the tetrahedral mesh. Because a tetrahedral mesh is not a hierarchical structure, ray-surface intersections are calculated mainly by traversing a few tetrahedra. Besides, this approach has the advantages of providing a unified data structure for global illumination, handling deforming geometry if the topology (connectivity) of the mesh does not change, quickly applying level-of-detail approaches, and ray tracing on the Graphics Processing Unit (GPU). Despite these advantages, the state-of-the-art traversal methods for tetrahedral meshes, such as Scalar Triple Product (ScTP), are still a magnitude or two slower than the k-d tree-based traversal, as Lagae and Dutré [1] state. We aim to improve the performance of the tetrahedral mesh-based traversal for ray-tracing as follows.

- We propose a compact tetrahedral mesh representation to improve cache locality and utilize memory alignment.

- We sort tetrahedral mesh data (tetrahedra and points) using a space-filling curve to improve cache locality.

- We propose an efficient tetrahedral mesh traversal algorithm using a modified basis that reduces the cost of point projection, frequently used during traversal.

- We utilize the GPU to speed up the ray-surface intersection calculations.

We also propose a technique to associate vertex attributes, such as normals and texture coordinates, with the tetrahedral mesh data. Through experiments, we observe that our method performs better than the existing tetrahedral mesh-based traversal methods in terms of computational cost. In certain scenes, especially the scenes with challenging geometry where there are long, extended triangles, we observe a better rendering performance than the k-d tree and BVH implementations of the *pbrt-v3* [13]. Although this method cannot replace and improve

upon the state-of-the-art accelerators (such as BVH and k-d tree) because of its disadvantages in its current form, its orthogonal strengths compared to the alternatives make it valuable and promising. This is especially important for aggregate structures where accelerators with different advantages can be combined to have the best of both worlds.

2.1 Acceleration Structures

Acceleration structures reduce the number of ray-primitive intersection tests by spatially organizing the primitives in the scene. The following sections summarize the research on such structures as grids, k-d trees, and BVHs.

2.1.1 Grids

A regular grid, first proposed by Fujimoto et al. [14], partitions the 3D scene into equally-sized boxes where each box keeps a list of triangles. During traversal, some well-known algorithms, such as the three-dimensional digital differential analyzer (3D DDA), can be used to determine the boxes that intersect with the ray quickly. One major disadvantage of the regular grid is its non-adaptive structure. Most grid cells may not contain any triangles, while some grid cells may have many triangles, increasing the average traversal cost. The regular structure also implies that empty space must be traversed rather inefficiently as many cells need to be visited. Cohen and Zhi [15] apply a city-block-based distance transform to the regular grid. Therefore, each grid cell is assigned a value that indicates the distance to the closest non-empty grid cell. That way, during traversal, cells can be skipped based on these distance values to speed up the grid traversal. Alphan and Isler [16] employ direction-dependent distance transform and a GPU-friendly traversal algorithm with minimal branching to speed up the regular grid-based ray tracing and volume rendering. Lagae and Dutre [17] propose two grid-based accelerators: The first one, *compact grid*, aims to minimize the memory usage by having exactly one index for a cell and a primitive along with a linear time build

algorithm. The second one, *hashed grid*, further reduces the memory needed by using a perfect hashing based on row displacement compression proposed by Aho and Ullman [18]. Kalojanov et al. [19] use a two-level nested grid to ray-trace dynamic scenes. They propose a massively parallel GPU-based build algorithm. Two-level nested structure helps them tackle teapot in a stadium problem by providing a more adaptive structure. Kalojanov et al. [20] propose another structure, called *Irregular Grids*, which is a flat, non-hierarchical, grid-like spatial subdivision structure. They optimize this structure by merging cells, which increases the traversal performance. They optimize it further by decoupling cell boundaries which allow them to extend cells over others with no geometry inside. This structure enables the skipping of empty space more efficiently. They also provide a very efficient parallel GPU-based build algorithm.

2.1.2 BVHs

One of the famous structures in the literature is BVHs. A BVH is a collection of hierarchical bounding volumes enclosing the scene's objects. BVHs improve the ray tracing performance by culling the scene geometry using bounding volume intersection tests. Therefore, fewer triangle-ray intersection tests must be performed compared to the brute force entire scene traversal.

Modern BVH construction techniques employ the Surface Area Heuristic (SAH) [21] to construct high-performance acceleration structures. SAH is a heuristic based on the probability of visiting a tree node and further descending from that tree node. By carefully optimizing a tree based on its SAH values, it is possible to create faster accelerators. SAH assumes a uniform distribution of the rays over the scene. Unlike that, Bittner and Havran use [22] Ray Distribution Heuristic (RDS), which uses the ray distribution in the scene to build better performance BVHs. Aila et al. [23] use the end-point overlap heuristic (EPO), which is proposed based on the fact that most rays originate on the surfaces. They improve ray-tracing performance by penalizing overlapping surfaces inside a bounding box.

In the presence of a SAH-like heuristic, at each tree level, a split position should be selected to minimize the surface area of the tree. This process is computationally quite costly as the start and end points of each primitive need to be considered on the split axis. Wald et al. [24] use a technique called *binning* to remedy this problem. Their approach divides the scene is divided into bins that contain the primitives. Then, split values need to be considered only between bins. This process accelerates the construction of the BVH while benefiting from a good SAH-based structure. Ganestam et al. [25] propose a technique called *Bonsai* where high-quality sub-trees are built by doing an extensive search for good structure for tight clusters of geometry. Then, they build the full tree by using these well-optimized treelets.

The state-of-the-art BVHs are constructed using a greedy top-down plane-sweeping algorithm proposed by Goldsmith et al. [26], which is extended by Stitch et al. [27] using spatial splits. Spatial splits work by aiming to reduce wasted space and overlap in the BVH nodes by using more AABBs to enclose the primitives. In this way, AABBs provide tighter bounding volumes. However, this approach results in multiple nodes per primitive and increases the memory footprint for the accelerator. It is also more expensive to build a BVH tree with spatial splits. Karras and Aila [23] also proposed another spatial split-based method where tighter AABBs enclose primitives before the tree construction. Popov et al. [28] extend the SAH cost function by introducing the *overlap penalty* term, aiming to reduce the node overlap. Wodniok et al. [29] use recursive SAH values of temporarily constructed SAH-built BVHs to reduce ray traversal cost further.

Dynamic scenes are generally handled by using two-level BVHs. Wald [30] uses a separate BVH for each object in the scene, and those per-object BVHs are inserted into a top-level BVH. The top-level structure is usually called *Top-level Acceleration Structure (TLAS)* while a per-object structure is called *Bottom-level Acceleration Structure (BLAS)*. Since objects are mostly expected to have rigid body motion, animated scenes could be handled easily by rebuilding the TLAS only as BLASes are not needed to be rebuilt. One problem with two-level hierarchies is the overlap of BLASes, which reduces the efficiency of ray-tracing as overlapping trees need to be traversed together. To alleviate this issue, Benthin

et al. [31] use a technique called *re-braiding* to exchange the portions of trees aimed at minimizing overlap.

In most BVH-based methods, axis-aligned bounding boxes (AABB) are used since they are very compact and fast to test. However, they can struggle with long-thin and diagonal geometry as AABB can not tightly bounds such primitives. Woop et al.[32] use oriented bounding boxes (OBB) to bound hair primitives to have better fitting bounds. They also split hair segments into multiple parts to reduce wasted space and overlap between nearby BVH nodes. Wald et al. [33] exploit the hardware-accelerated ray transforms to speed up ray tracing long-thin primitives. They use tighter OBBs and "trick" the GPU to use hardware-accelerated OBB tests on these primitives. Lauterbach et al. [34] proposed a parallelizable BVH construction called Linear BVH (LBVH) scheme where primitives are sorted in Morton order [35] derived from their positions. Later, Pantaleoni and Luebke [36] combine the SAH-based heuristic with the LBVH method, which is called *Hierarchical LVBH*. Vinkler et al. [37] extend Morton codes so that they can account for primitive size along with primitive position as well. Extended Morton codes result in better-performing trees where primitive sizes are not uniform in the scene.

In ray tracing, most of the time, the first hit is needed. However, in some scenarios (e.g., volume rendering or the rendering of transparent geometry), multiple hit data could be needed. For that purpose, Wald et al. [38] use an iterative method to find the multiple intersections by keeping track of the traversal state between queries.

Wide BVHs, where tree nodes contain more than two children nodes, are also commonly used. Wide BVHs generally make use of SIMD optimized instructions to accelerate Ray-AABB tests. Wald et al. [39] use wide BVHs to accelerate ray-tracing. Wide BVHs are generally more memory efficient than binary BVHs.

2.1.3 Octrees

The octree is another spatial indexing structure used to accelerate ray tracing [40]. It divides the space into eight subspaces in a recursive manner. During ray tracing, the octree is used to index the scene into subspaces, and it helps determine the sub-spaces that intersect with the rays. After an octree is constructed, triangles that reside in these sub-spaces can be queried, and the closest intersection with the rays can be found by performing a relatively small number of ray-surface intersection tests compared to the brute force approach. Whang et al. [41] propose a more efficient octree variant. This structure, called octree-R, uses more efficient split planes by considering the potential number of ray intersection tests after the resulting subdivision. In modern ray-tracers, octrees are seldom used.

2.1.4 k-d trees

Similar to the octree, the k-d tree is also a space partitioning structure that divides the space into two sub-spaces at each level by alternating the split axis. To reduce the average ray traversal cost on a k-d tree, these split planes are selected using the SAH proposed by [26]. SAH-based k-d tree construction approaches are later improved by [42]. Wald et al. [43] propose a SAH-based k-d tree construction scheme with $\mathcal{O}(N \log N)$ computational complexity. The k-d trees constructed using the SAH are adaptive to the scene geometry, which means that if a ray is not in the proximity of any scene geometry, only a few tree nodes are traversed. This approach reduces the computation cost of ray tracing on scenes where primitives in the scene are not uniformly distributed, which is a common scenario for 3D scenes. Choi et al. [44] propose a parallelized and SAH-based k-d tree construction scheme to speed up the build times, which is one of the weaknesses of the k-d trees for ray tracing. In modern ray-tracers, k-d trees are rarely used as BVH-based accelerators mostly outperform them.

2.1.5 Ray-specialized Acceleration Structures

In many ray tracing applications, rays share a common point, such as rays originating from the camera or rays cast to the light sources after ray-surface intersections. The structures discussed above do not exploit the characteristics of such rays in ray tracing. Better structures take advantage of rays that share a common point in space and create indices accordingly. Light Buffer [45] is an approach that partitions the scene according to one light source in the scene, which is then used for shadow testing. Hunt et al. [46] propose the perspective tree, similar to the Light Buffer that uses a 3D grid in a perspective space considering the position of the light source or the camera as the root. They later improve the perspective tree approach using an adaptive splitting scheme using SAH [47].

2.1.6 Hybrid Structures

Researchers also explored the idea of hybrid structures to a large extent to combine the strengths of the different spatial subdivision schemes. Klimaszewski [48] uses BVH and grids in a single acceleration structure to combine these two structures' orthogonal strengths. Lin et al. [49] propose *Dual Split Trees* where BVH-like nodes have k-d tree-like split planes to isolate and cull geometry more effectively. They later on provided the hardware-accelerated version [50] of this approach.

2.1.7 Other approaches

Arvo [51] proposes the ray classification technique, where rays are treated as 5-D vectors (three values using ray origin and two values using ray direction in the polar domain). These ray clusters define a convex region in the space that encloses a certain number of primitives. Therefore, during ray traversal, ray intersection tests are only performed against these primitives, which reduces the total number of ray-intersection tests. Reshetov [52] describes a collective

ray-testing method, where a beam is tested against a kd-tree and then used to accelerate ray testing for rays that belong to that beam. By doing so, they reported an order of magnitude improvement in the traversal. Ize et al. [53] propose an efficient Binary Space Partitioning (BSP) structure for ray-tracing. In BSP, a split plane is not necessarily axis-aligned. While their approach is efficient in terms of traversal speed, searching for an optimal split plane is very costly, making this approach less popular among the alternatives. During traversal, some primitives might be referenced from multiple nodes in a tree (or even by different trees in multi-level approaches). The mailboxing technique is commonly used to avoid redundant testing, proposed by Amanatides and Woo [54] and Arnaldi et al. [55]. Each ray is given a unique identifier in their method, and each primitive is associated with the ray identifier being checked during traversal. This scheme can be used to avoid redundant intersection tests between ray and primitives by doing a simple identifier check before the actual intersection test.

2.2 Tetrahedral Mesh Construction and Traversal

Given an input geometry, a tetrahedral mesh can be constructed using well-known algorithms in computational geometry. TetGen [56] is a commonly used tool to generate tetrahedral meshes. TetGen uses Bowyer-Watson algorithm [57, 58] and the incremental flip [59] algorithms. Both methods have the worst-case complexity of $\mathcal{O}(N^2)$. If points are uniformly distributed in space, the expected run-time complexity is $\mathcal{O}(N \log N)$. Robust geometric predicates are used to ensure numerical robustness [60].

There are tetrahedral mesh-based traversal methods used for accelerating ray-surface intersection calculations. Lagae and Dutré [1] use ScTP to traverse the tetrahedral mesh. Their method requires the computation of three to six ScTP to determine the exit face. ScTP computation involves a cross product followed by a dot product on 3D vectors. Maria et al. [61] propose a fast tetrahedral mesh

traversal method, which uses an efficient exit face determination algorithm based on Plücker coordinates.

Our method uses an efficient traversal method that works in 2D, resulting in very few floating-point operations per tetrahedron compared to these alternatives. Our data structures are also compact and memory aligned. We also use a space-filling curve to improve cache locality further. Maria et al. [62],[63]) also propose a new acceleration structure for ray tracing, constrained convex space partition (CCSP), as an alternative to tetrahedral mesh-based acceleration structures. CCSP is more suitable for architectural environments because such a scene partitioning contains a smaller number of convex volumes than a large number of tetrahedra.

2.3   Ray-casting for Direct Volume Rendering

Direct volume rendering methods for rendering irregular grids, mainly represented as unstructured tetrahedral meshes, rely on ray-casting and the composition of shades of samples along the rays throughout the volume to calculate pixel colors. For example, Silva et al. [64], [65] use a sweeping plane first applied in the x-z plane, and then a sweeping line applied on the z-axis. They process these sweep lines further to render volumetric data stored as an irregular grid. Berk et al. [66] focus on using hybrid methods to utilize the strengths of the image- and object-space methods. They rely on a next-cell operation for determining the next tetrahedron that the ray travels, as proposed by Koyamada et al. [67].

Garrity [68] uses a simple traversal method where the ray is intersected with tetrahedra faces, and the closest intersection gives the exit face for the tetrahedron. Koyamada [67] uses two (on average) point-in-triangle tests in 2D to determine the exit face. Riberio et al. [69] use a more compact data structure for reduced memory usage during traversal. They also utilize ray coherence to reduce run-time memory usage. Later on, they [70] improved this method by providing a hardware implementation with additional arrangements of the data

structure for reduced memory usage. Marmitt and Slusallek [71] use a method proposed by Platis and Theoharis [72], which employs Plücker Coordinates of the ray and the tetrahedron edges to determine the exit face. They use the entry face information to reduce the number of tests to determine the exit face. On average, they find the exit face using 2.67 ray-line orientation tests per tetrahedron.

Alternatively, Cell trees (based on Bounding Interval Geometry) [73] and Tetrahedral trees [74] provide efficient ways to answer point location queries in tetrahedral meshes and thus can be used to speed up sampling operations in volumetric rendering. However, these techniques cannot be used efficiently to answer ray-triangle intersection queries. Additionally, these structures are not designed for consecutive tetrahedron traversal.

We provide a fast and compact acceleration structure to quickly find ray-surface intersections for rendering 3D scenes composed of polygons (surface data). As opposed to direct volume visualization methods, our acceleration structure can handle queries for random rays scattered in different directions, given that their origin is already located (ray connectivity). Direct volume rendering techniques are geared towards rendering volumetric data from a specific camera position and orientation. Recently, our tetrahedral ray traversal scheme has been adapted to tetrahedron traversal (marching) consecutively for direct volume rendering methods for better cache utilization and reduced computational cost [11].

Chapter 3

Compact Acceleration Structures for Ray-tracing

3.1 Tetrahedral Mesh Representation

We use a compact tetrahedral mesh representation for good cache utilization. We store tetrahedral mesh in two arrays as in [1]. The first array stores the point data, and the second array stores the tetrahedron data. Figure 3.1 depicts the tetrahedron data representation for typical scenarios.

V_0^i	V_1^i	V_2^i	V_3^i
N_0^i	N_1^i	N_2^i	N_3^i

Figure 3.1: Typical tetrahedron representation in the memory. V_j^i represents the index of the j'th vertex of the i'th tetrahedron. N_j^i represents the neighboring tetrahedron index, which is across the vertex V_j^i. Each field is an integer and four bytes long. Thus, the full tetrahedron data occupies 32 bytes of memory.

Instead of using this representation, we propose three tetrahedron storage schemes that are more compact and better suited for efficient traversal: *Tet32*,

Tet20, and *Tet16*, which are 32, 20, and 16 bytes, respectively. We store a common field, exclusive-or sum (xor-sum), in all these structures, inspired by xor linked list structures for reducing the memory requirements of linked lists [75]. Mebarki ([76]) uses a similar structure for compact 2D triangulations. VX^i denotes the xor-sum of the vertex indices of the ith tetrahedron and V_j^i denotes the index of the jth vertex of the ith tetrahedron. We compute the xor-sum as follows.

$$VX^i = V_0^i \oplus V_1^i \oplus V_2^i \oplus V_3^i$$

Tet32 structure contains the first three vertex indices, xor-sum of all vertex indices, and four neighbor indices. Its memory layout is depicted in Figure 3.2.

V_0^i	V_1^i	V_2^i	VX^i
N_0^i	N_1^i	N_2^i	N_3^i

Figure 3.2: *Tet32* structure. Each field is an integer and four bytes long. The tetrahedron data occupies 32 bytes of memory.

With the *Tet32* representation, we can use the *xor* operation to quickly retrieve the vertex's index that is not on a given face. We can get the index of the fourth point V_3^i as follows.

$$V_3^i = V_0^i \oplus V_1^i \oplus V_2^i \oplus VX^i.$$

This follows from the fact that xor operation is associative, commutative, and has the property $X \oplus X = 0$.

If the corresponding face is a part of the scene geometry, neighbor index data points to a structure called *constrained face*. We use a single bit-mask to identify such faces on the neighbor tetrahedron index field, where the remaining 31 bits are used to reference either a neighboring tetrahedron or a constrained face depending on the value of the bit-mask. Constrained face structure holds a reference to the

actual triangle geometry and stores references to the neighboring two tetrahedra indices. These indices are used to recover and initialize the tetrahedron data when scattering rays are traced. Figure 3.3 illustrates the constrained face structure where N_{tet} and N_{other_tet} are the indices of the two neighboring tetrahedra on that face, and F is the face in the scene geometry.

N_{tet}	N_{other_tet}	F

Figure 3.3: *Constrained face* structure. Each field is an integer and four bytes long. Constrained face data occupies 12 bytes of memory.

It should be noted that multiple constrained faces can point to a single triangle when we allow triangles to be subdivided during tetrahedralization to enable high-quality tetrahedral meshes. This is analogous to having spatial-splits [27] in BVHs.

The XOR-based storage scheme has the following advantages:

- XOR-based representations are compact and thus require less memory. Hence, the cache memory can be utilized more efficiently to reduce the rendering times. Additionally, more data can be fitted to the memory, especially critical for GPU implementations.

- Fetching the subsequent tetrahedron data becomes straightforward. Thanks to XOR representation, only one additional vertex needs to be fetched, and it can be fetched easily by XORing all the vertex indices making the traversal code more straightforward and efficient.

3.2 Tetrahedron Traversal

As the first step of tetrahedron traversal, we construct a 2D basis $b = (\vec{u}, \vec{v})$ from the ray direction using the method described in [77]. Then, we define a new 2D coordinate system C with basis b and origin o where r_o is the ray origin. We

transform tetrahedron vertices to the coordinate system C to obtain four points in 2D. We determine the exit face in the initial tetrahedron using at most four points in triangle tests in 2D. The query point is at the origin because the ray origin is the center of the new coordinate system C. Once we determine the exit face, we keep the 2D coordinates and indices of the points of the exit face as p_0, p_1, p_2 and idx_0, idx_1, idx_2, respectively. We also fetch the next tetrahedron index using the neighbor data.

After the initialization step, we start traversing the tetrahedral mesh. We first fetch the index of the fourth corner of the next tetrahedron (three corners are already known because two neighboring tetrahedra share three vertices) using the following expression where XV^{next} denotes the xor sum of the next tetrahedron.

$$idx_3 = idx_0 \oplus idx_1 \oplus idx_2 \oplus XV^{next}.$$

Using the index idx_3, we fetch the vertex from the points array, transform it to the new coordinate system, C, and use the resulting 2D point p_3 to decide the exit face of the ray (cf. Algorithm 1). Because the query point is at the origin after transformation, only four floating point multiplications and two floating point comparisons are sufficient. The exit face index is denoted as $exit\_face\_idx$ and resides across the point $p_{exit\_face\_idx}$ whose index is $idx_{exit\_face\_idx}$. To get the next tetrahedron, we use the $idx_{exit\_face\_idx}$ in the current tetrahedron data to fetch the corresponding neighbor tetrahedron index. Figure 3.4 illustrates the coordinate system transformation for a ray and a tetrahedron.

In *Tet32*, we simply search for $idx_{exit\_face\_idx}$ in the current tetrahedron. Since vertex and neighbor indices correspond to each other, location of the $idx_{exit\_face\_idx}$ (value from 0 to 3) also reveals the location of the neighbor to be traversed next. We describe this process in Algorithms 2 and 3.

We terminate the traversal if the neighbor index points to a constrained face or tetrahedral mesh boundaries. Otherwise, knowing the next tetrahedron, we discard p_i and idx_i by replacing its contents with the newly fetched point data
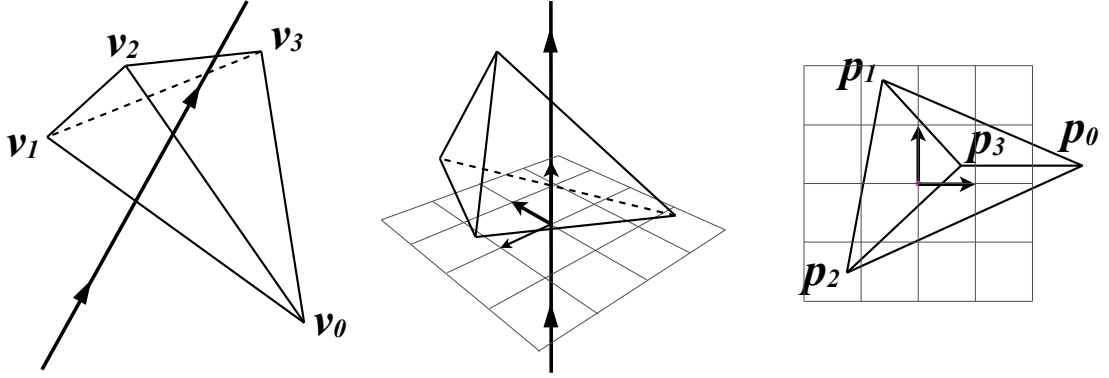
Figure 3.4: Ray-tetrahedron intersection. Left: a ray and a tetrahedron. Middle: the tetrahedron transformed into the coordinate system defined by the ray. The ray coincides with the z-axis. Right: the tetrahedron projected onto 2D. The ray is at the origin and points to the viewer.

Algorithm 1 Exit face selection

procedure GETEXITFACE($p_{0..3}$)
    $exit\_face\_idx \leftarrow 0$
    if $det(\vec{p_3}, \vec{p_0}) < 0$ **then**
        if $det(\vec{p_3}, \vec{p_2}) \geq 0$ **then**
            $exit\_face\_idx \leftarrow 1$
        end if
    else if $det(\vec{p_3}, \vec{p_1}) < 0$ **then**
        $exit\_face\_idx \leftarrow 2$
    end if
    return $exit\_face\_idx$
end procedure

Algorithm 2 Tetrahedron traversal loop for *Tet32*

while $tet\_idx \geq 0$ **do**
    $idx_{exit\_face\_id} \leftarrow idx_3$
    $idx_3 \leftarrow idx_0 \oplus idx_1 \oplus idx_2 \oplus VX^{tet\_idx}$
    $v_{new} \leftarrow points_{idx_3} - r_o$
    $p_{exit\_face\_idx} \leftarrow p_3$
    $p_3 \leftarrow (\vec{u} \cdot v_{new}, \vec{v} \cdot v_{new})$
    $exit_face_idx = $ GETEXITFACE($p_{0..3}$)
    $next_tet_idx = $ GETNEXTTET32(tet_idx, idx_i)
end while

Algorithm 3 Next tetrahedron determination for *Tet32*

---

procedure GETNEXTTET32($tet\_idx, idx_{0..3}$)
    $next\_tet\_idx \leftarrow N_3^{tet\_idx}$
    for $i \leftarrow 0, 2$ **do**
        if $idx_{exit\_face\_idx} = V_i^{tet\_idx}$ **then**
            $next\_tet\_idx \leftarrow N_i^{tet\_idx}$
        end if
    end for
    return $next\_tet\_idx$
end procedure

---

p_3 and idx_3. We repeat this process until a geometry is intersected or the tetra-hedral mesh boundaries are reached. In this method, no further modifications are necessary to ensure vertex ordering because the counterclockwise ordering is always preserved for points on the exit face.

Fetching a new vertex id requires three bitwise exclusive or operations. The coordinate system transformation of the newly fetched point is six floating-point multiplications and four floating-point additions. We decide whether a face is an exit face using four floating-point multiplications and two floating-point comparisons. Finally, we determine the next tetrahedron index using the appropriate method for the preferred structure.

Our alternative tetrahedral representations, namely *Tet20* and *Tet16*, are even more efficient since their compact nature results in better cache utilization. These representations can be used in the same way during traversal since we can reconstruct full tetrahedron data given that traversal operations exhibit *ray connectivity*. However, both representations require additional operations to reconstruct tetrahedron data. Appendix A describes these representations and operations required to reconstruct the tetrahedron data.

In *Tet20*, we eliminate vertex indices and store only the xor-sum and the neighboring indices. We use the xor-sum field to get the index of the unshared vertex of the next tetrahedron during traversal. To this end, shared vertices between two tetrahedra must be known, which is guaranteed by *ray connectivity*, meaning that the start and endpoints of rays are always connected in a typical

ray-tracing scenario. However, we use a *source tet*, a tetrahedron with complete index information, to initialize the indices at the beginning. We can choose this tetrahedron randomly. It is possible to reconstruct the indices of the neighboring tetrahedra starting from *source tet*. We need to sort the neighbor indices in a tetrahedron using their corresponding vertex indices to find the neighbor for a given vertex index. Figure 3.5 shows the memory representation of the *Tet20* structure.

VX^i	N_0^i	N_1^i	N_2^i	N_3^i

Figure 3.5: *Tet20* structure. Each field is an integer and four bytes long. The tetrahedron data occupies 20 bytes of memory.

In *Tet16*, instead of storing four neighbor indices explicitly, we store three values that can be used to reconstruct neighbor indices, given that the previous (neighbor) tetrahedron index is known. We compute these three indices as follows.

$$NX_0^i = N_0^i \oplus N_3^i$$

$$NX_1^i = N_1^i \oplus N_3^i$$

$$NX_2^i = N_2^i \oplus N_3^i$$

Knowing the index of a neighbor tetrahedron and its order, we can easily reconstruct the rest of the neighbors. For example, if we have N_2^i, we retrieve N_1^i as follows.

$$N_2^i = N_2^i \oplus NX_2^i \oplus NX_1^i.$$

The resulting *Tet16* structure is given in Figure 3.6.

25

VX^i	NX_0^i	NX_1^i	NX_2^i

Figure 3.6: *Tet16* structure. Each field is an integer and four bytes long. The tetrahedron data occupies 16 bytes of memory.

3.3 Tetrahedron Traversal for *Tet20* and *Tet16* Representations

In *Tet20*, we use the property that neighbor indices are sorted using their counterpart vertex indices as keys. Thus, to find the next neighbor index, we find the order of $idx_{exit\_face\_idx}$ among idx_0, idx_1, idx_2, idx_3 (which are actually the vertex indices of the tetrahedron). Because the neighbor indices are sorted using vertex indices, order of the vertex index also happens to be the next neighbor index. We describe this process in Algorithms 4 and 5.

Algorithm 4 Tetrahedron traversal loop for *Tet20*

---

    while $tet_{idx} \geq 0$ **do**
        $idx_{exit\_face\_idx} \leftarrow idx_3$
        $idx_3 \leftarrow idx_0 \oplus idx_1 \oplus idx_2 \oplus VX^{tet\_idx}$
        $v_{new} \leftarrow points_{idx_3} - r_o$
        $p_{exit\_face\_idx} \leftarrow p_3$
        $p_3 \leftarrow (\vec{u} \cdot v_{new}, \vec{v} \cdot v_{new})$
        $exit\_face\_idx = \text{GETEXITFACE}(p_{0..3})$
        $order_a \leftarrow$ sorted order of id_3 among id_i
        $next\_tet\_idx = \text{GETNEXTTET20}(tet\_idx, order\_a)$
    end while

Algorithm 5 Next tetrahedron determination for *Tet20*

---

    procedure $\text{GETNEXTTET20}(tet\_idx, order\_a)$
        $next\_tet\_idx \leftarrow N_{order\_a}$
        return $next\_tet\_idx$
    end procedure

In *Tet16*, we use the previous tetrahedron index to reconstruct next tetrahedron index using the values NX_j^i. As in $Tet20$, we need to construct the value NX_j^i using sorted vertex indices. To reconstruct the next tetrahedron, sorted order of

values are computed for idx_3, which corresponds to a previous tetrahedron and $idx_{exit\_face\_idx}$, which corresponds to an exit face, must be computed. We describe this process in Algorithms 6 and 7.

Algorithm 6 Tetrahedron traversal loop for *Tet16*

---

while $tet\_idx \geq 0$ **do**
    $idx_3 \leftarrow idx_0 \oplus idx_1 \oplus idx_2 \oplus VX^{tet\_idx}$
    $v_{new} \leftarrow points_{idx_3} - r_o$
    $p_3 \leftarrow (\vec{u} \cdot v_{new}, \vec{v} \cdot v_{new})$
    $order_a \leftarrow$ sorted order of id_3 among id_i
    $exit\_face\_idx = \text{GETEXITFACE}(p_{0..3})$
    $order_b \leftarrow$ sorted order of $id_{exit\_face\_idx}$ among id_i
    $next\_tet\_idx =$
      $\text{GETNEXTTET16}(tet\_idx, prev\_tet\_idx, order\_a, order\_b)$
    $\text{SWAP}(tet\_idx, prev\_tet\_idx)$
end while

Algorithm 7 Next tetrahedron determination for *Tet16*

---

procedure $\text{GETNEXTTET16}(tet\_idx, prev\_tet\_idx, order\_a, order\_b)$
    if $order_a \neq 3$ **then**
      $next\_tet\_idx = prev\_tet\_idx \oplus NX^{tet\_idx}_{order_a}$
    end if
    if $order_b \neq 3$ **then**
      $next\_tet\_idx = next\_tet\_idx \oplus NX^{tet\_idx}_{order_b}$
    end if
    return $next\_tet\_idx$
end procedure

3.4   Point Projection Using Specialized Basis

We project newly fetched points to the two-dimensional (2D) coordinate system using two dot product operations, which require six floating-point multiplications and four floating-point additions. We can optimize this step by scaling the basis vectors to make some components zero or one. Since the basis vectors are only scaled, the exit face determination still works correctly. To avoid numerical issues, we scale vectors so that only the absolute largest components become one (or minus one). Equation (3.1) describes the construction of the first basis vector

$\vec{u}$, which is orthogonal to $\vec{n}$ (and not necessarily of unit length). It should be noted that this technique is only useful for long traversals where a ray visits many tetrahedra. For short-lived travels, the setup cost renders the performance benefit negligible.

$$\vec{u}_{min} = 0,$$
$$\vec{u}_{(min+1) \ mod \ 3} = \frac{\vec{n}_{(min-1) \ mod \ 3}}{\vec{n}_{max}}, \qquad (3.1)$$
$$\vec{u}_{(min-1) \ mod \ 3} = -\frac{\vec{n}_{(min+1) \ mod \ 3}}{\vec{n}_{max}},$$

where $\vec{v}_0$, $\vec{v}_1$, and $\vec{v}_2$ correspond to $\vec{v}_x$, $\vec{v}_y$, and $\vec{v}_z$, respectively, min and max are the indices of the absolute smallest and largest components of the vector $\vec{n}$.

We construct the second basis vector $\vec{v}$, which is orthogonal to $\vec{n}$ and $\vec{u}$ (and not necessarily of unit length), as in Equation (3.2).

$$\vec{t} = \vec{n} \times \vec{u},$$
$$\vec{v} = \frac{\vec{t}}{\vec{t}_{(3-max-min)}}. \qquad (3.2)$$

Now, we can transform three-dimensional (3D) point v to the 2D coordinate system using the basis $b = (\vec{u}, \vec{v})$, as shown in Equation 3.3. It should be noted that the sign s of the last parameter $\vec{v}_{min}$ can be either positive or negative depending on the sign of $\vec{v}_{min}$.

$$other = 3 - max - min,$$
$$\vec{p}_x = \vec{u}_{max}\vec{v}_{max} + \vec{v}_{other}, \qquad (3.3)$$
$$\vec{p}_y = \vec{v}_{max}\vec{v}_{max} + \vec{v}_{other}\vec{v}_{other} \pm \vec{v}_{min}.$$

Three floating-point multiplications and three floating-point addition/subtractions are sufficient to perform the above computation. We implement this fast projection method using a templated function over the variables min, max, and

$sign(\vec{v}_{min})$ and call the corresponding function by inspecting the components of the new basis to avoid run-time overhead of keeping additional function arguments.

3.5   Handling Common Ray-tracing Operations

We handle common ray-tracing operations using tetrahedral meshes as follows.

Point lights: Point lights are handled by having a field denoting the tetrahedron index that contains that light. This way, point lights can be tested for visibility during rendering efficiently. Ray traversal must be initiated from the query point towards the point light. If the tetrahedron containing the point light is reached, point light is visible. If the ray hits a constrained geometry before doing so, that light source is blocked from the query point.

Mesh Lights: Mesh lights do not require special treatment. Once a ray hits a constrained geometry, its material is checked for a light-emitting material, and lighting computations can be carried out accordingly.

Reflection and refraction rays: Two tetrahedra that share the incident triangle must be reported at each intersection point to handle reflection and refraction rays. These two tetrahedra (one is further inside the incident triangle and the other is on the outside) are then used to initialize ray traversal for reflection and refraction, respectively. Since ray origin is on the incident triangle, an initial 4-face search method is unnecessary. In order to simplify this process, traversal data needed to initialize reflected and refracted faces could be stored in the *Constrained Faces* table. We call this data *Shared Face* which is illustrated in Figure 3.7 . It stores the indices of the incident triangle vertices along with the non-shared vertices of two neighboring tetrahedra on that incident triangle. Figure 3.8, illustrates different types of rays used in a typical ray-tracing scenario.

VX_{tet}	VX_{other_tet}	V_0	V_1	V_2

Figure 3.7: *Shared face* structure. Each field is an integer and four bytes long. Constrained face data occupies 12 bytes of memory.
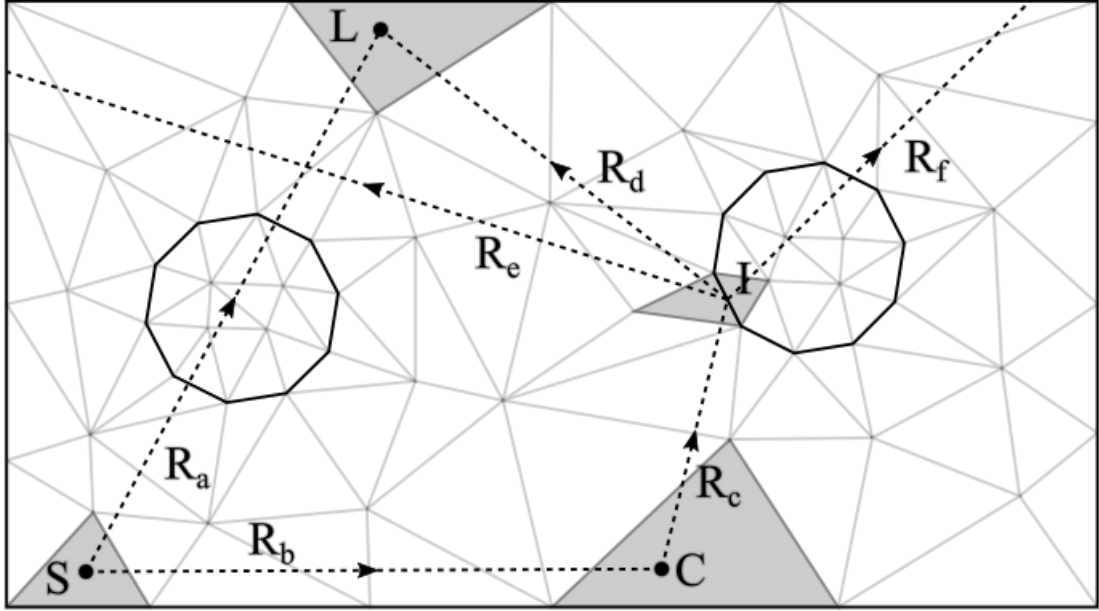

Figure 3.8: The types of rays in tetrahedral mesh-based ray tracing. S is the center of source tetrahedron. C is the camera position. I is the intersection point. L is the light source position. Rays R_a and R_b are used to locate the light source and camera position, respectively. R_c is the camera ray. R_d is the shadow ray. R_e and R_f are the reflection and refractions rays, which are cast from two different neighboring tetrahedra.

3.6   Accelerator Construction

Constructing the tetrahedral mesh-based accelerator out of a 3-D scene involves a couple of steps which are as follows.

1. Meshes that are part of the scene geometry (including the light-emitting meshes) are transformed into the world space. This step ensures that the tetrahedralization method runs on an agreed world-space transformation.

2. A 3-D spatial grid is constructed, and the vertices in the scene are inserted into this structure using keys derived from vertex positions. If two vertices share the same position, they are merged to ensure that tetrahedral meshing works in the existence of incident points. Such situations can occur if content generators model some geometry by breaking down the shared vertices for convenience. Given that the scene geometry contains no intersections, the resulting mesh is now PLC and ready to be tetrahedralized.

3. We insert Steiner points to improve the ray tracing performance of the tetrahedral mesh-based accelerator. This step is optional.

4. Constrained tetrahedralization is built using a tetrahedralization algorithm. Scene geometry is constrained, meaning it will be preserved in the tetrahedral output mesh.

5. We sort the tetrahedral mesh data (both points and tetrahedra) using a Hilbert Curve to improve the cache utilization.

6. We process the tetrahedral meshes for potential early termination optimizations such as *Half-space based early ray termination* and *Portal-based early ray termination.* This step is optional.

7. We built the *Constrained faces* table, enabling us to have a spatial splits-like approach while keeping the accelerator as compact as possible.

8. Canonical TetMesh structure is converted to one of the efficient TetMesh representations: *TetMesh32, TetMesh20* or *TetMesh16.*

9. We identify and store the data for source tetrahedron, which is used to locate the ray origin. In order to minimize numerical issues, we select a tetrahedron that is as equilateral as possible.
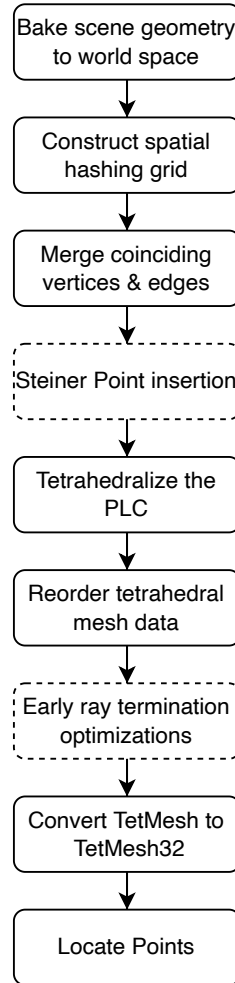
```
┌─────────────────────┐
│ Bake scene geometry │
│   to world space    │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  Construct spatial  │
│    hashing grid     │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  Merge coinciding   │
│  vertices & edges   │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│ Steiner Point insertion │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  Tetrahedralize the │
│         PLC         │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│ Reorder tetrahedral │
│      mesh data      │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│ Early ray termination │
│    optimizations    │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│  Convert TetMesh to │
│      TetMesh32      │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐
│    Locate Points    │
└─────────────────────┘
```

Figure 3.9: Steps during the construction of tetrahedral mesh based ray-tracing accelerator. Steps illustrated with dashed lines are optional.

3.6.1 Early Ray Termination

Tetrahedral mesh traversal can be terminated early if a ray passes through a specific triangle where the next half-space determined by it contains no geometry

to intersect. We implement early space termination in two ways: *half-space-based early ray termination* and *portal-based early ray termination.*

3.6.1.1   Half-space-based Early Ray Termination

Half-space-based early ray termination exploits the fact that each face of a tetrahedron defines a half-space. If a ray passes through a tetrahedron face where the defined half-space does not have a scene geometry, further traversal is no longer needed. Figure 3.10 shows a triangulated scene where half-space-based early ray termination is used to reduce the number of traversal steps to improve ray-tracing performance.
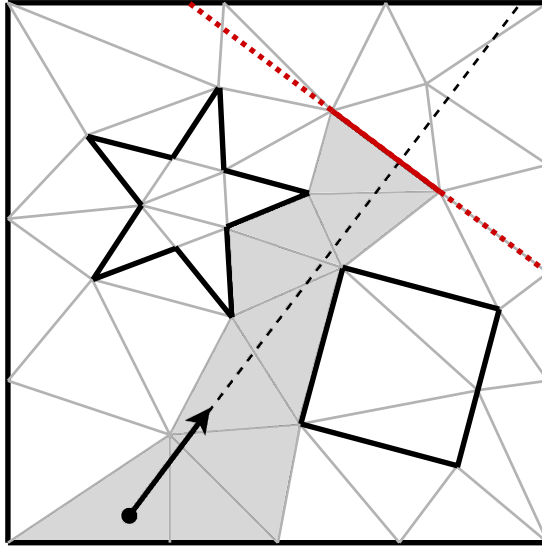
Figure 3.10: Half-space-based early ray termination. The red segment indicates the triangle where ray can be terminated early.

In order to use this technique, we preprocess the tetrahedral mesh to find faces that define empty half-spaces. For this purpose, scene geometry is checked against each face using a left-test. If all tests have the same sign, that face defines an empty half-space; we mark the corresponding neighbor link for that face to terminate the traversal immediately as soon as a ray passes through it. Preprocessing of half-spaces is highly parallelizable. Figure 3.11 shows the comparison of renderings with and without half-space-based early ray termination comparisons.
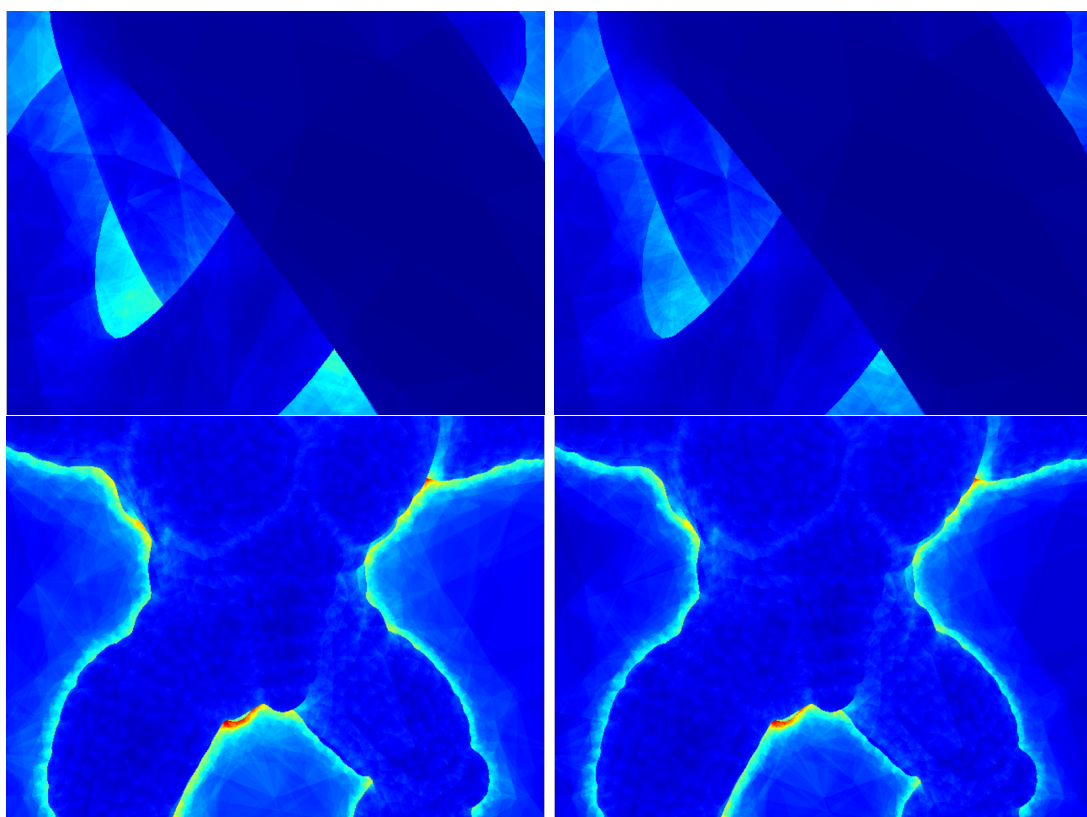
Figure 3.11: The effect of half-space-based early ray termination. Left: Rendered without half-space optimization. Right: Rendered with half-space optimization.

3.6.1.2 Portal-based Early Ray Termination

Portal-based early ray termination works by knowing that a ray that passes through two faces of a tetrahedron cannot intersect with some geometry in the scene. A ray that passes consecutively through two faces defines a region (referred to as *portal*). If this region contains no geometry, it is safe to terminate the traversal early. To use this optimization, we must keep track of entry and exit faces when traversing a tetrahedron. Since there are three possible entry faces for each exit face, three bits need to be stored to see whether to see the termination is possible or not. Therefore, additional 12 bits are needed for a tetrahedron to encode early ray termination information. Figure 3.12 illustrates the portal-based early ray termination.
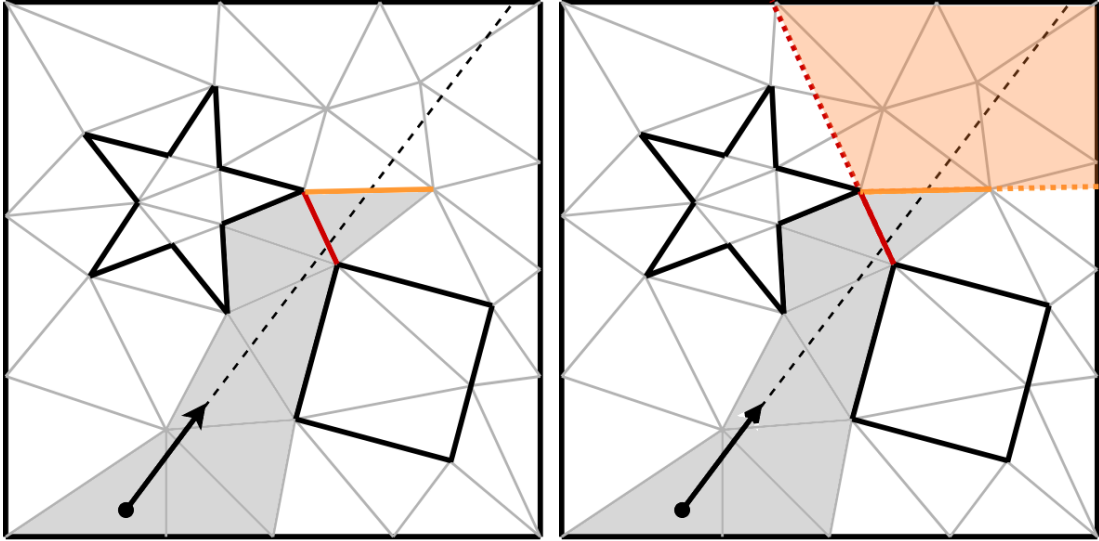
Figure 3.12: Portal-based early ray termination. Left: entry face is red, exit face is yellow. Right: Corresponding portal space for the given entry-exit face combination which contains no scene geometry, thus suitable for early termination.

3.6.2 Minimum Weight Triangulation

The structure of the tetrahedral mesh dictates the number of tetrahedrons traversed. Therefore, it is possible to increase ray-tracing performance by building *better* tetrahedral meshes. In the simulation domain, better tetrahedralization

often means tetrahedra with a better maximum radius-edge ratio, resulting in more equilateral tetrahedra, enabling more accurate simulations. However, for ray tracing performance, this statement does not hold. Lagae and Dutre [1] discuss that tetrahedral meshes with a smaller total surface area are better for ray tracing as rays pass through tetrahedron faces at each step. A smaller surface area means less number of nodes traversed. It is possible to reduce the total surface area (called *weight*) by adjusting the quality parameter of the tetrahedral mesh building algorithm. Figure 3.13 shows the resulting weight of a tetrahedral mesh for different scenes with varying maximum radius-edge ratio parameter values. Since the result of this experiment does not provide a clear relationship between the maximum radius-edge ratio and the resulting weight, we generally use values between 2-6 based on empirical observations, which results in smaller weight values in general.
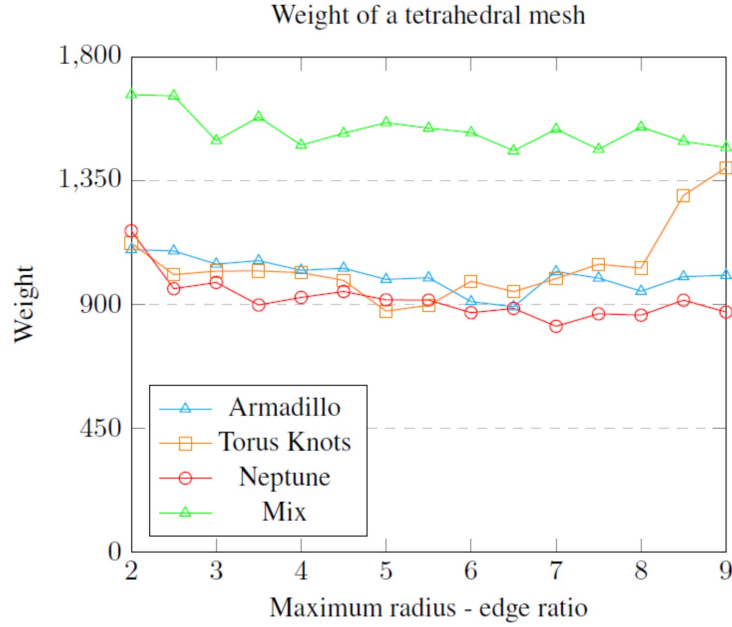
Figure 3.13: Tetrahedral mesh weight - tetrahedralization quality. Weight values are computed for each tetrahedral mesh for different scenes using different quality values.

3.6.3 Intersecting Geometry

Intersecting geometry needs special treatment to build a tetrahedral mesh-based accelerator. For that purpose, intersection resolving operators are used. Such operators create additional geometry (vertices, edges, and faces) in the intersecting meshes so that the resulting mesh is still a PLC when combined. Popular mesh processing libraries (MeshLib [78]) and content creation tools (Blender [79]) provide the functionality to resolve intersections between meshes. Figure 3.14 illustrates this process where two meshes are shown before and after resolving the intersection.
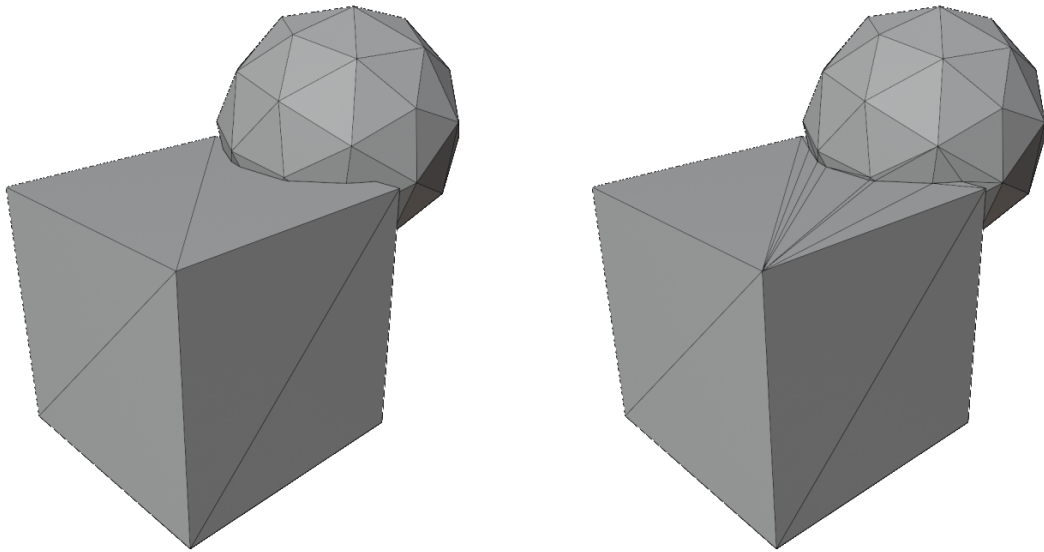
Figure 3.14: Resolving intersections for two meshes where tetrahedralization is not possible. Left: Two intersecting meshes. Right: New meshes after intersections have been resolved. These meshes can now be tetrahedralized.

3.6.4 Steiner Points

Special rays such as shadow and camera rays can be accelerated further by strategically optimizing the structure of the tetrahedral mesh. By inserting special vertices into the tetrahedralization, called Steiner points [80, 81], it is possible to reduce the number of traversed nodes for such rays. For example, placing a

Steiner point behind a light creates long tetrahedra along the direction of scene geometry vs. light. This means that rays will travel more distance in each tetrahedron, thus reducing the number of traversed tetrahedra.

3.6.5 Hidden-tetrahedra Removal

It is possible to reduce the accelerator size by removing the tetrahedra located inside the regions where rays cannot enter. For example, tetrahedra generated inside a fully opaque geometry might never be visited during ray-tracing. In such cases, vertices and tetrahedra inside such geometry could be removed safely without affecting the traversal.

Most tetrahedral mesh construction methods output a region id for each enclosed region in the resulting tetrahedral mesh. Thus, hidden regions can be conveniently detected by checking the region flags after tetrahedral mesh construction.

Hidden regions could still be visited when locating the source tetrahedron; therefore, a modification might be needed to locate the tetrahedron that contains the ray origin. A* algorithm [82], for example, could be used to locate the tetrahedron by initiating the search from the source tetrahedra. Tetrahedron centroids can be used to compute the cost function, and at each step, a point-in-tetrahedron test needs to be carried out to check for termination.

3.7   Reordering Tetrahedral Mesh Data

We reorder points and tetrahedra in memory to improve cache locality during ray-traversal. For this purpose, we use a two-step method. In the first step, we detect if there are distinct regions in the tetrahedralization. Each closed surface in the input geometry divides the space into two regions (outside and inside regions). These regions occur when a set of constrained faces completely

encloses a set of tetrahedra. Because the rays are traced until a constrained face is encountered, the tetrahedra from different regions are not visited in a single ray traversal, which is not the case for multi-hit traversal methods. Thus, we store the tetrahedra that belong to the same region nearby in memory. Furthermore, we reorder points based on their positions and tetrahedra based on their center points. We map points to memory using a Hilbert curve [83] (see Figure 3.15). A Hilbert curve is a space-filling curve that can map spatial data from three dimensions to one dimension by preserving the locality. In this way, primitives close to each other in 3D space are also close to each other in one dimension.
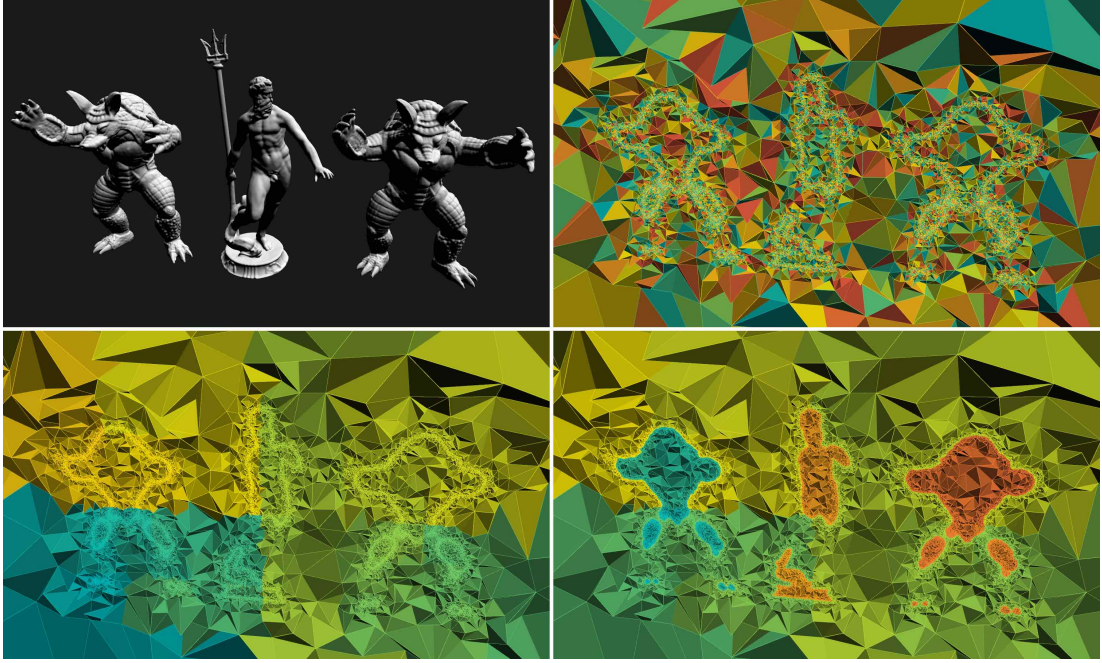
Figure 3.15: Sorting tetrahedron data. Top left: The three-dimensional scene. Top right: Unsorted tetrahedron data. Bottom left: Tetrahedron data sorted using a Hilbert curve. Bottom right: Tetrahedron data sorted using a Hilbert curve and mesh regions. Memory positions are coded with different colors. Tetrahedra that are close in memory are represented with similar colors. Tetrahedron data are stored in a contiguous manner. Courtesy of Aman et al. [10].

3.8 GPU Implementation

For the GPU implementation, we use the CUDA platform. Once we build the tetrahedral mesh-based acceleration structure, the tetrahedra and points data are copied to the GPU. We store the constrained face data on the host computer because it is not a part of the *hot* data, which is frequently accessed during traversal. Once the initialization is complete, the steps to render a single frame are as follows.

1. We identify the *source tetrahedron* on the CPU.

2. We pass the batch of rays and the source tetrahedron to the global memory of the GPU.

3. CUDA kernels run for each ray, traversing the scene, and terminate when they hit the scene geometry.

4. We store the results of the intersection calculations in the GPU's global memory and then pass them to the main memory. We use these results to perform shading and generate additional rays.

Our method can be trivially implemented for the CUDA platform. However, this trivial implementation does not provide the best performance on the GPU in terms of computation speed. Thus, we perform the following optimizations to make our method run faster on the GPU.

1. We project ray origin to the 2D coordinate system beforehand. When projecting the newly fetched point, translation is performed on the 2D coordinate system instead of a 3D one. Thus, instead of using the origin in 3D, we use the projected origin in 2D. This potentially results in fewer occupied registers on the GPU, resulting in better performance. We compute the projected origin, *po*, as follows:

$$po = (\vec{u} \cdot r_o, \vec{v} \cdot r_o), \qquad (3.4)$$

where $(\vec{u}, \vec{v})$ is the 2D basis constructed from the ray. During traversal, we can project the new point to the 2D plane as follows:

$$p_3 = (\vec{u} \cdot v_{new} - po_x, \vec{v} \cdot v_{new} - po_y), \tag{3.5}$$

where p_3 is the projected point and v_{new} is the newly fetched point from the next tetrahedron.

2. We make use of CUDA textures when accessing tetrahedral mesh data. To optimize traversal in *Tet20* and *Tet16* structures, we use a single channel integer texture (32 bytes). The required elements to use the texture are the xor field and one neighbor field for *Tet20*, the xor field, and one or two neighbor fields for *Tet16*. We fetch and store these in a local stack, potentially reducing the number of registers used.

3.9 Experimental Results

We compare our approach to k-d tree, BVH, and the state-of-the-art tetrahedral mesh-based methods, namely the ScTP-based traversal [1] and the Plücker coordinate-based traversal [61]. We use the k-d tree and SAH-based BVH implementations, as described in [24, 84]. We use the original implementation provided by Maria et al. ([61]) for the Plücker coordinate-based traversal.

We use TetGen [56] to generate the tetrahedral mesh of the 3D scene. We perform experiments on a computer with six cores @3.2 GHz (Intel), 16 GB of main memory, and NVIDIA GTX 1060 with 6 GB of memory. On the CPU, we render the scenes at a resolution of 1920×1440 using multi-threading by subdividing the image into 16×16 tiles and assigning them to available threads. To make a fair comparison between our method and the other state-of-the-art approaches, we render the same scene many times and pick the best result for each method to avoid noisy measurements due to background processes.

Tables 3.1, 3.2, and 3.3 show the computational costs of the construction of acceleration structures and rendering times of different traversal methods. The test scenes in Table 3.3 cannot be tetrahedralized using TetGen. We tetrahedralized them using TetWild [85] and used the remeshed geometry as input. To test the adaptiveness of the structures in a challenging scene geometry, we include the versions of the scenes with bounding boxes composed of large triangles. Experiments show that our method performs better than the ScTP- [1] and Plücker coordinate-based traversal methods [61] in all scenes. It performs better than the BVH-based traversal in seven of the fifteen scenes and better than the k-d tree-based traversal in six of the fifteen scenes. In the other scenes where BVH- and k-d tree-based traversal methods perform superior to our tetrahedral mesh-based traversal, the rendering times are mostly close. While testing the state-of-the-art tetrahedral-mesh-based traversal methods of [1, 61], we sorted the tetrahedral meshes using space-filling curves for a fair comparison. Although the construction times of BVH and k-d tree are lower than that of the tetrahedral meshes, the tetrahedral mesh is constructed during preprocessing, and it does not affect the ray-tracing performance for the scenes that do not require the update of acceleration structures. The tetrahedral mesh does not need to be updated for dynamic scenes where the topology does not change. If the topological changes to a tetrahedralization are local, the tetrahedral mesh can be updated with efficient insertion and removal operations [1].

Table 3.4 shows rendering times of tetrahedral mesh-based methods for test scenes on the GPU. *Tet20* representation gives the best performance, around 15% faster than Maria's method while occupying much less memory (half of the memory required by Maria's method in the largest scene). *Tet16* representation requires even less memory, but it is not as fast as *Tet20* (roughly the same performance as Maria's method) due to more memory and arithmetic operations needed to decode compressed neighbor data.

Table 3.1: The computational costs of different acceleration structures and rendering times for traversal methods (part 1). We compare our *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures and traversal methods with *Tet-mesh-ScTP* [1], *Tet-mesh-80* [61], BVH [13], and *k*-d tree [13].
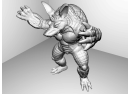
Scenes					
	Torus Knots	Torus Knots in a Box	Armadillo	Armadillo in a Box	Neptune
Scene statistics					
# of triangles	77,760	77,772	345,938	345,950	448,896
# of tetrahedra	270,036	270,351	1,027,749	1,021,965	1,240,582
Construction times (in seconds)					
Tet-mesh-ScTP	3.596	3.638	16.733	20.252	79.822
Tet-mesh-80	3.656	3.663	16.595	20.143	79.657
Tet-mesh-32	3.753	3.643	16.659	20.262	79.536
Tet-mesh-20	3.778	3.658	16.704	20.444	79.573
Tet-mesh-16	3.640	3.524	16.546	19.919	79.846
BVH	**0.078**	**0.079**	**0.391**	**0.396**	**0.474**
k-d tree	0.739	0.590	1.454	1.651	2.265
Rendering times (in milliseconds)					
Tet-mesh-ScTP	261.5	293.4	232.5	306.7	268.9
Tet-mesh-80	244.1	278.9	218.1	262.4	236.5
Tet-mesh-32	150.7	181.7	148.5	182.5	158.7
Tet-mesh-20	**125.8**	**142.3**	117.1	145.3	127.1
Tet-mesh-16	136.3	152.4	124.6	153.2	135.9
BVH	152.7	192.2	**78.1**	**126.1**	**78.7**
k-d tree	139.9	214.4	85.7	182.3	81.7

Table 3.2: The computational costs of different acceleration structures and rendering times for traversal methods (part 2). We compare our *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures and traversal methods with *Tet-mesh-ScTP* [1], *Tet-mesh-80* [61], BVH [13], and *k*-d tree [13].
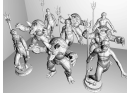
Scenes					
	Neptune in a Box	Mix	Mix in a Box	Mix close	Mix in a Box close
Scene statistics					
# of triangles	448,908	2,505,992	2,506,004	2,505,992	2,506,004
# of tetrahedra	1,242,883	7,259,175	7,252,946	7,259,175	7,259,193
Construction times (in seconds)					
Tet-mesh-ScTP	155.478	124.208	170.216	124.840	485.950
Tet-mesh-80	156.381	125.081	169.116	125.068	482.908
Tet-mesh-32	153.788	124.000	169.502	123.183	487.291
Tet-mesh-20	155.580	124.087	169.890	124.015	483.827
Tet-mesh-16	154.644	124.493	170.175	123.401	484.516
BVH	**0.487**	**2.968**	**3.017**	**2.966**	**2.997**
k-d tree	2.471	13.889	14.668	13.846	16.624
Rendering times (in milliseconds)					
Tet-mesh-ScTP	279.6	402.6	430.0	449.5	455.0
Tet-mesh-80	261.0	355.7	384.7	411.2	419.6
Tet-mesh-32	176.1	247.2	268.5	265.5	269.9
Tet-mesh-20	137.0	196.3	211.1	205.6	**210.2**
Tet-mesh-16	152.0	223.9	241.4	237.4	240.3
BVH	**120.4**	144.6	**187.9**	224.9	253.8
k-d tree	162.6	**143.5**	214.8	**193.1**	213.6

Table 3.3: The computational costs of acceleration structures and rendering times for traversal methods (remeshed scenes). We compare our *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures and traversal methods with *Tet-mesh-ScTP* [1], *Tet-mesh-80* [61], BVH [13], and k-d tree [13].
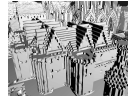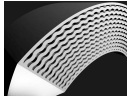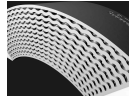
	Scenes				
	Rungholt Far	Rungholt Default	Rungholt Close	Exhaust Pipe Left	Exhaust Pipe Right
Scene statistics					
# of triangles	3,580,928	3,580,928	3,580,928	6,244,678	6,244,678
# of tetrahedra	8,381,071	8,381,071	8,381,071	18,480,542	18,480,542
Construction times (in seconds)					
BVH	4.230	4.247	4.212	7.771	7.767
k-d tree	37.140	37.078	37.039	66.462	66.403
Rendering times (in milliseconds)					
Tet-mesh-ScTP	554.035	525.523	436.716	333.291	344.483
Tet-mesh-80	488.299	466.933	400.589	312.152	320.361
Tet-mesh-32	353.444	333.274	265.337	202.472	207.239
Tet-mesh-20	282.211	265.337	215.118	163.814	166.619
Tet-mesh-16	313.335	293.351	238.027	183.198	186.125
BVH	198.140	227.333	243.165	177.263	187.540
k-d tree	**119.589**	**127.948**	**126.434**	**114.358**	**121.114**

Table 3.4: The rendering times of tetrahedral mesh-based acceleration structures on the GPU. We compare our *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures and traversal methods with *Tet-mesh-ScTP* [1] and *Tet-mesh-80* [61].
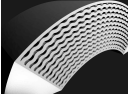
Scenes						
	Torus Knots	Armadillo	Neptune	Mix	Rungholt	Exhaust Pipe
Scene statistics						
# of triangles	77,760	345,938	448,896	2,505,992	3,580,928	6,244,678
# of tetrahedra	270,036	1,027,739	1,240,582	7,259,175	8,381,071	18,480,542
Kernel execution time (in milliseconds)						
Tet-mesh-ScTP	20.021	19.612	20.898	43.910	43.633	22.560
Tet-mesh-80	7.790	7.136	7.941	13.454	14.360	9.023
Tet-mesh-32	19.541	18.950	20.958	42.690	44.544	21.320
Tet-mesh-20	**6.156**	**5.803**	**6.529**	**11.322**	**12.172**	**6.931**
Tet-mesh-16	7.120	6.477	7.328	12.157	13.444	8.231

Table 3.5 shows the memory costs for different acceleration structures on different scenes. Our most compact structure, *Tet16*, can be stored using significantly less memory than the other tetrahedral-mesh-based alternatives, *Tet-mesh-ScTP* [1] and *Tet-mesh-80* [61], which provides two benefits. First, accelerators for much larger scenes can fit the main memory or GPU global memory. Second, 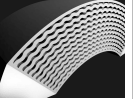this small footprint provides much better performance by facilitating cache locality. Our smallest accelerator data is memory aligned (16 bytes per tetrahedron). Memory usage of the k-d tree tends to be affected considerably by the distribution of the primitives in the scene, thus resulting in a smaller or larger accelerator size in different scenes.

On the other hand, memory used for a BVH structure is always smaller than its tetrahedra-mesh-based counterparts. It should be noted that BVHs and tetrahedra-mesh-based accelerators exhibit orthogonal strengths, which can be combined for better results, as demonstrated in [9]. This approach allows dynamic scenes to be rendered by combining two structures: a top-level acceleration structure, a BVH, and a bottom-level acceleration structure, a tetrahedral mesh.

Figure 3.16 demonstrates the effect of the tetrahedral mesh sorting on rendering performance. Even though sorting is not vital for performance in small scenes, it significantly improves the rendering performance in large scenes.

Table 3.6 demonstrates the efficiency of a tetrahedral mesh-based traversal approach when the camera gets closer to a surface. In this experiment, we render the images at varying distances to the Armadillo 3D model and compare the rendering times for different acceleration structures. Both BVH and k-d tree perform much better than the tetrahedral mesh structure when the camera views the object from a fair distance. However, as the camera gets closer to the surface, the traversal cost decreases because the tetrahedral mesh is not hierarchical, unlike the BVH and k-d tree. In the extreme case, when the camera is about to touch the surface, only one tetrahedron is traversed. This is not the case for hierarchical structures because many tree nodes may need to be traversed to find the closest ray-surface intersection. This comparison shows that, depending on the scene, tetrahedral mesh-based accelerator or BVH could be preferred over

Table 3.5: The memory requirements of different acceleration structures. We compare our proposed tetrahedra-mesh-based *Tet-mesh-32*, *Tet-mesh-20*, and *Tet-mesh-16* structures with the state-of-the-art tetrahedra-mesh-based acceleration structures, *Tet-mesh-ScTP* [1] and *Tet-mesh-80* [61], and other types of acceleration structures, Bounding Volume Hierarchy (BVH), and *k*-d tree. Because *Tet-mesh-32* and *Tet-mesh-ScTP* use the same tetrahedral mesh representations, their memory requirements are the same.

	Scenes					
	Torus Knots	Armadillo	Neptune	Mix	Rungholt	Exhaust Pipe
Scene statistics						
# of triangles	77,760	345,938	448,896	2,505,992	3,580,928	6,244,678
# of tetrahedra	270,036	1,027,739	1,240,582	7,259,175	8,381,071	18,480,542
Accelerator size (in megabytes)						
*k*d-tree	19.9	7.8	27.4	97.1	512.0	1417.8
BVH	4.7	20.4	10.6	150.0	175.6	380.0
Tet-mesh-ScTP	12.3	49.4	61.2	352.1	406.1	885.1
Tet-mesh-80	20.6	78.4	94.6	553.8	639.4	1410.0
Tet-mesh-32	12.3	49.4	61.2	352.1	406.1	885.1
Tet-mesh-20	9.2	37.7	47.0	269.0	310.2	673.6
Tet-mesh-16	8.2	33.7	42.2	241.3	278.2	603.1
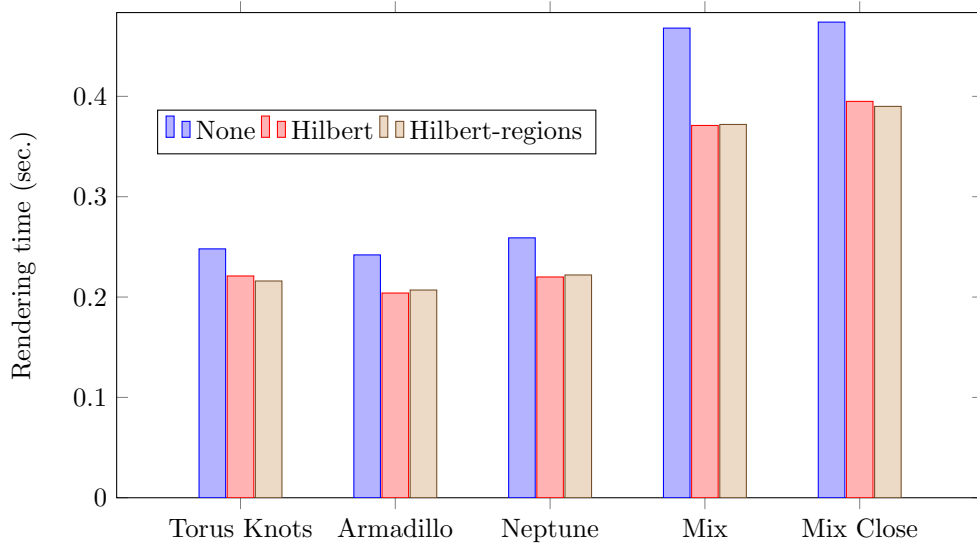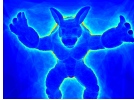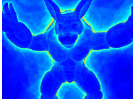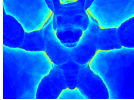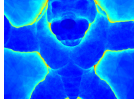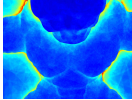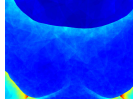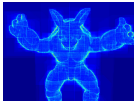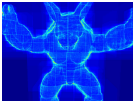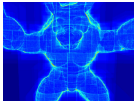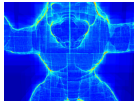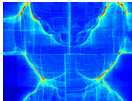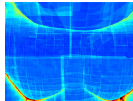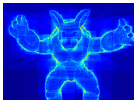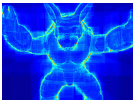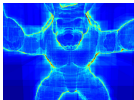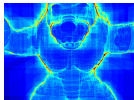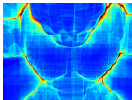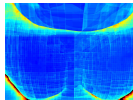
Figure 3.16: The rendering times for unsorted and sorted tetrahedral mesh data. Courtesy of Aman et al. [10].

each other for better performance. We can employ SAH and other heuristics to determine the best accelerator for a particular scenario. In Chapter 4.2, we present three traversal cost heuristics to determine the cost of ray-tracing using tetrahedral meshes which can be used to determine the best accelerator to use for a particular geometry.

Table 3.6: The rendering times and visited node counts for different types of accelerators as the camera gets closer to the mesh surface. We compare our *Tet-mesh-20* structure and traversal method with BVH [13] and k-d tree [13].

Scenes						
Tet-mesh-20						
BVH						
kd-tree						
Visited node count per pixel						
Tet-mesh-20	48.54	52.32	55.11	59.13	60.13	**43.62**
BVH	**27.23**	**32.90**	**38.53**	**46.67**	**57.88**	65.27
k-d tree	34.12	41.84	49.50	60.21	70.18	65.46
Rendering times (in milliseconds)						
Tet-mesh-20	140.3	151.0	168.3	179.4	182.3	**126.6**
BVH	87.4	109.4	123.9	146.8	175.9	193.9
k-d tree	**86.8**	**107.4**	**118.8**	**136.8**	**157.9**	144.0

Chapter 4

Applications

Tetrahedral meshes have their uses in many domains. We can use our compact and efficient methods in different fields. In the following sections, we describe the applications of our compact tetrahedral mesh-based representation to different domains such as volume rendering, hybrid accelerators, and point location queries in a triangulation.

4.1   Volume Rendering

NVIDIA recently introduced the Turing [86] architecture, which supports hardware-accelerated ray-tracing queries. We extended our method to support volumetric data utilizing the RTX cores on these GPUs through the OPTIX framework [87]. This approach has the following advantages:

- It uses a compact and efficient memory representation that can be used for fast ray-tetrahedra intersection tests and ray traversal inside the tetrahedralization. This GPU-friendly structure is based on the techniques described in Chapter 3.

- It can handle both primary and secondary rays; therefore, it could be used

to produce effects such as light shadows, scattering, and ambient occlusion.

- It can be used in both convex and non-convex volumetric domains and volumes with challenging geometry that contains holes and discontinuities.

4.1.1 Method Overview

A ray marcher can be used to perform volumetric rendering over a tetrahedral mesh. Since a tetrahedral domain can be in any form (convex or non-convex), entry and exit faces to these tetrahedral elements must be located efficiently. To this end, shell faces (boundary faces of the tetrahedral domain) are extracted and put into an efficient BVH structure in the GPU. By leveraging the computational power on the GPU, entry and exit faces can be located very quickly. Once these faces are located, rays are marched through the tetrahedral mesh using our fast tetrahedral mesh traversal method. This method provides a convenient way of computing secondary rays such as ambient occlusion, reflection, and refraction by utilizing the ray connectivity. Figure 4.1 shows some images rendered using this method. Sahistan et al. [11] and Sahistan [88] provide the details of these methods.

4.2 Two-level Hybrid Acceleration Structures

Rendering dynamic scenes is generally done by having a two-level accelerator structure. In such a structure, the bottom levels represent objects that can have a dynamic motion. The top-level structure is generally a BVH tree where bottom-level objects are treated as leaves. Similarly, we propose BVH-Tetrahedralization Hybrid (BTH) acceleration structure to speed up ray tracing and propose a construction method for the BTH structure. We also show that the BTH structure can be adapted to dynamic scenes.
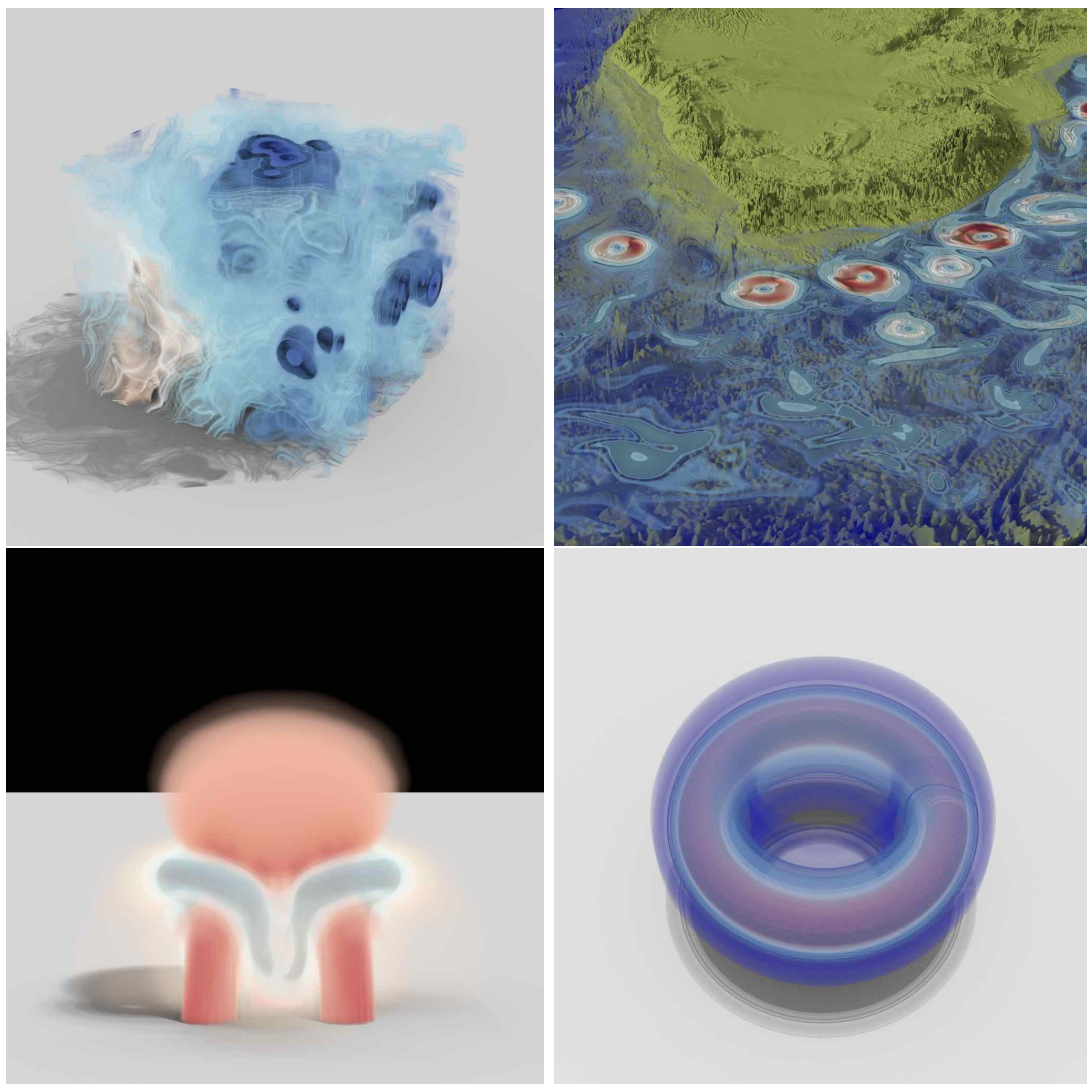
Figure 4.1: Volumetric images rendered on the GPU. Courtesy of Sahistan et al. [11].

4.2.1 BVH-Tetrahedralization Hybrid Structure

BTH refers to a Bounding Volume Hierarchy (BVH) where some leaves are tetrahedralized. This structure combines the strength of BVHs and tetrahedralizations, meaning that the structure can handle self-intersected models and support different primitives. This structure can also benefit from early termination.

4.2.1.1 Top-Down BTH Construction

BTH is constructed in three steps.

1. The complete BVH is constructed using a top-down construction method utilizing the Surface Area Heuristic (SAH).

2. Self-intersection-free nodes are marked. A node has a self-intersection if its descendants contain self-intersection or its children have intersecting primitives. The constructed BVH can be used to speed up self-intersection detection.

3. Nodes are marked as to be tetrahedralized if

$$Cost_{TET}(node) < Cost_{SAH}(node),$$

where $Cost_{TET}(node)$ is the total cost of traversal of a tetrahedralized node and $Cost_{SAH}(node)$ is the total traversal cost of a SAH-based BVH node. Marked nodes are then pruned, tetrahedralized, and stored in the BVH tree as leaf nodes.

4.2.1.2 BTH Nearest-Hit Traversal

BTH traversal is very similar to the BVH traversal. Traversal starts from the root, and when a tetrahedralized node is found, the first tetrahedron is located to initiate tetrahedral mesh traversal. Then, tetrahedralization is traversed by following neighboring links and our efficient tetrahedral traversal technique.

4.2.1.3 Tetrahedralization Nearest-Hit Cost Heuristic

The BTH Construction algorithm marks nodes to be tetrahedralized using the formula in the preceeding section, requiring that $Cost_{TET}$ needs to be computed or approximated. Similar to the SAH, $Cost_{TET}$ can be approximated in the following way, assuming the rays are distributed uniformly and originate outside the tetrahedralization. $Cost_{TET}$ becomes directly related to the number of tetrahedra traversed, denoted as N_{avg}, which can be approximated using different methods.

Sampling-based cost calculation: A Monte-Carlo approach is employed in this method. We randomly sample rays, traverse the tetrahedralization along the ray, and derive an average number of tetrahedrons traversed using those samples. Sampling-based cost calculation accuracy improves as the number of samples increases, and it can approximate N_{avg} well. However, since it requires a tetrahedralization, it is slow for BTH construction and used only for comparison and evaluation of the other methods.

Average depth-based cost calculation: In this method, the average number of tetrahedra traversed is derived using the average depth of rays and the average length of the edges of the tetrahedra. The average number of tetrahedra traversed is

$$N_{depth}(T_{node}) \approx d_{avg}/I_{avg},$$

where d_{avg} is the average depth of rays and I_{avg} is the average length of the edges. Compared to the sampling-based cost calculation method, this method tends to underestimate the cost.

Primitive count-based cost calculation: Alternatively, primitive count in a node can be used to approximate the N_{avg} as well such as

$$N_{count}(T_{node}) \approx \sqrt[3]{|F|},$$

where F is number of faces in that particular node. This method makes cost approximation more accurate than the average depth-based cost calculation.

4.2.1.4 Handling Animated Scenes

Acceleration structures need to be adapted for dynamic scenes. For this purpose, a complete rebuild or update can be done. Assuming objects have rigid-body motion only, we can employ a two-level strategy to handle updates to the BTH to render dynamic scenes. For this purpose, we construct a bottom-level acceleration structure (BTH) for each object and then construct a top-level acceleration structure (BVH) for these bottom-level structures. Figure 4.2 illustrates this two-level structure.
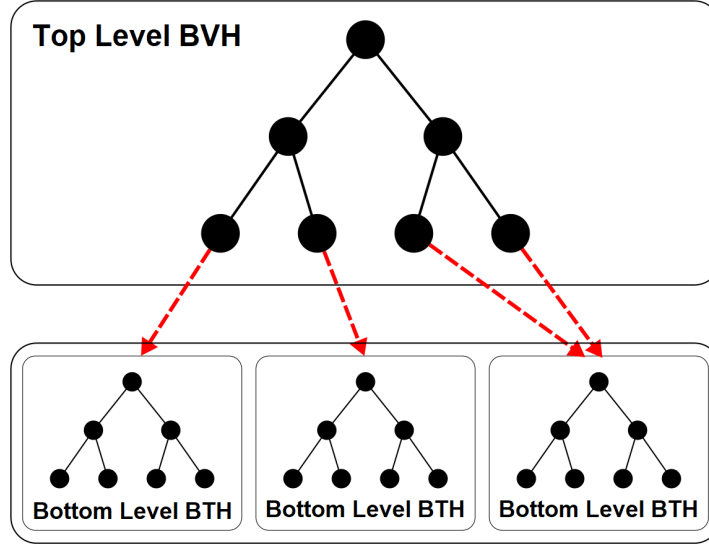
Figure 4.2: Two-level hierarchy to handle animated scenes. Courtesy of Aman et al. [9].

The nearest hit-traversal with this two-level hierarchy works as follows. First, the bounding volume hierarchy, i.e., the top-level acceleration structure (TLAS), is traversed. When the leaves of the TLAS, i.e., bottom-level acceleration structure (BLAS), which are tetrahedralizations, are reached, we transform rays into objects' coordinates and continue traversing the BLAS in the local coordinate system. If the scene is changed, only the top-level needs to be built as the bottom levels do not change. Aman et al. [9] and Demirci [89] provide a detailed explanation of these methods.

4.3 Point Location Queries in Two-dimensional (2-D) and Three-dimensional (3-D) Space

The traversal methods described in this work could be used to perform point location queries in triangulations in two different ways.

4.3.1 Point Location Queries in a Triangulation by Stabbing

Using the proposed ray traversal techniques to locate points in 2-D triangulations is straightforward. To this end, we can derive 2-D variants of our compact representation; the vertex data becomes an array of two-dimensional points. To represent the triangulation, we adapt the XOR-based scheme to triangles. Then, the triangle could be represented as shown in Figure 4.3.

$$\boxed{VX^i \mid N_0^i \mid N_1^i \mid N_2^i}$$
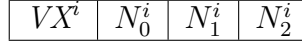
Figure 4.3: *Tri16* structure. Each field is an integer and four bytes long. The triangle representation occupies 16 bytes of memory.

It is possible to compress this data by XORing the neighbor links, similar to the compression process for *Tet16* (also proposed by Mebarki [76]).

$$NX_0^i = N_0^i \oplus N_2^i$$
$$NX_1^i = N_1^i \oplus N_2^i$$

Then, the resulting compact triangle representation, called *Tri12*, could be represented as shown in Figure 4.4.

The source triangle needs to be identified within the compact triangulation representation to trace a ray in this structure. Algorithm 8 and 9 describes the

$$\boxed{VX^i \mid N_0^i \mid N_1^i}$$

Figure 4.4: *Tri12* structure. Each field is an integer and four bytes long. The triangle representation occupies 12 bytes of memory.

ray traversal loop in a triangulation using *Tri16 structure.*

Algorithm 8 Triangle traversal loop for *Tri16*

---
    while $tri_{idx} \geq 0$ **do**
        $idx_{exit\_edge\_idx} \leftarrow idx_2$
        $idx_2 \leftarrow idx_0 \oplus idx_1 \oplus VX^{tri\_idx}$
        $v_{new} \leftarrow points_{idx_2} - r_o$
        $p_{exit\_edge\_idx} \leftarrow p_2$
        $p_2 \leftarrow (\vec{u} \cdot v_{new}, \vec{v} \cdot v_{new})$
        $exit\_edge\_idx = \text{GETEXITTRI}(p_{0..2})$
        $order_a \leftarrow$ sorted order of id_2 among id_i
        $next\_tri\_idx = \text{GETNEXTTRI16}(tri\_idx, order\_a)$
    end while

Algorithm 9 Next triangle determination for *Tri16*

---
    procedure $\text{GETNEXTTRI16}(tri\_idx, order\_a)$
        $next\_tri\_idx \leftarrow N_{order\_a}$
        return $next\_tri\_idx$
    end procedure

Algorithms 10 and 11 describe the ray traversal loop in a triangulation using *Tri12 structure.* Algorithm 12 describes the exit edge selection method for a ray traversal inside the triangulation.

4.3.2 Point Location Queries in a Triangulation Using Flattened Tetrahedral Mesh

Rather exciting usage of the proposed technique is the ability to use a *flattened tetrahedral mesh* to handle point location queries. For this, planar PCL containing the triangulation and another PCL, a very basic triangulation, must be fed into a tetrahedralization algorithm as constrained geometry. Once this structure is

---

Algorithm 10 Triangle traversal loop for *Tri12*

---

while $tri\_idx \geq 0$ **do**

$\quad idx_2 \leftarrow idx_0 \oplus idx_1 \oplus VX^{tri\_idx}$

$\quad v_{new} \leftarrow points_{idx_2} - r_o$

$\quad p_2 \leftarrow (\vec{u} \cdot v_{new}, \vec{v} \cdot v_{new})$

$\quad order_a \leftarrow$ sorted order of id_2 among id_i

$\quad exit\_edge\_idx = \text{GETEXITFACE}(p_{0..2})$

$\quad order_b \leftarrow$ sorted order of $id_{exit\_edge\_idx}$ among id_i

$\quad next\_tri\_idx =$

$\quad\quad \text{GETNEXTTET12}(tri\_idx, prev\_tri\_idx, order\_a, order\_b)$

$\quad \text{SWAP}(tri\_idx, prev\_tri\_idx)$

end while

Algorithm 11 Next triangle determination for *Tri12*

---

procedure $\text{GETNEXTTRI12}(tri\_idx, prev\_tri\_idx, order\_a, order\_b)$

$\quad$**if** $order_a \neq 1$ **then**

$\quad\quad next\_tri\_idx = prev\_tri\_idx \oplus NX^{tri\_idx}_{order_a}$

$\quad$**end if**

$\quad$**if** $order_b \neq 1$ **then**

$\quad\quad next\_tri\_idx = next\_tri\_idx \oplus NX^{tri\_idx}_{order_b}$

$\quad$**end if**

$\quad$**return** $next\_tri\_idx$

end procedure

Algorithm 12 Exit edge selection

---

procedure $\text{GETEXITEDGE}(p_{0..2})$

$\quad$**if** $det(\vec{p_3}, \vec{p_0}) < 0$ **then**

$\quad\quad exit\_face\_idx \leftarrow 1$

$\quad$**else**

$\quad\quad exit\_face\_idx \leftarrow 2$

$\quad$**end if**

$\quad$**return** $exit\_tri\_idx$

end procedure

---

tetrahedralized, point queries can be answered by shooting a ray from the top triangulation towards the bottom triangulation. Since ray is orthogonal to both triangulations, tetrahedral mesh information along the Z-axis could be safely ignored without affecting the traversal. Since the triangulation at the top level is simple (triangulated from regular quads), we can locate the initial tetrahedron in constant time. Point location query performance could be increased by increasing the resolution of the regular subdivision of the triangulation at the top level. It should be noted that an increase in the resolution also increases the memory required to store the flattened tetrahedral mesh. Figure 4.5 illustrates a point location query using a flattened tetrahedral mesh.
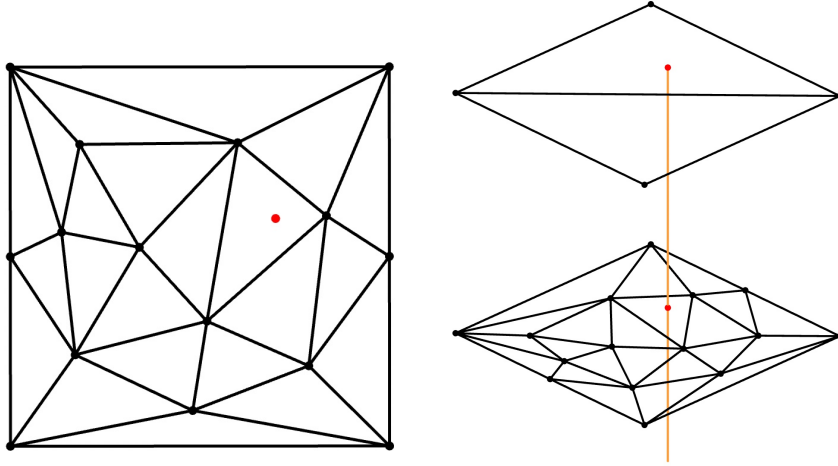
Figure 4.5: Point location query using a flattened tetrahedral mesh: Triangulation and the query point (red) (left). Simple triangulation on top of the actual triangulation (right). A ray orthogonal to the triangulations that passes through the query point will eventually end up in the queried triangle, given that the space between two triangulations is tetrahedralized.

4.4 Tetrahedralization of Very Large Meshes

Tetrahedralization is the core operation in the construction of a tetrahedral mesh-based accelerator. Therefore, to be able to use these accelerators for ray tracing purposes, large scenes need to be tetrahedralized as well. To this end, we employ

a practical method to tetrahedralize larges scenes with good quality where a tetrahedralization by well-known Tetrahedralization tools, such as TetGen [56], might fail due to memory limitations. We fix this problem using a simple yet practical divide-and-conquer algorithm, which is as follows [12].

1. We halve to input by having a split plane on the major axis of the scene geometry. We use the clip function provided by CGAL [90].

2. We repair the defects introduced by the clipping operation and ensure that triangulated surfaces at the split planes match for each side.

3. Each half is tetrahedralized separately and then merged. It should be noted that the algorithm is recursive, meaning that the input can be subdivided many times before halves can be merged again.

Erkoc et al. [12] and Erkoc [91] provide the details of this divide-and-conquer approach for tetrahedralization.

Chapter 5

Conclusions and Future Research Directions

We propose methods for fast tetrahedral mesh traversal for ray tracing. Specifically, we propose compact and memory-aligned tetrahedral mesh data structures. We use a space-filling curve to improve cache locality. We propose efficient traversal methods to improve ray-tracing performance and provide the GPU implementation. Experiments show that our approach can reduce rendering times substantially and perform better than other alternatives in different scenarios. There are two main limitations of using tetrahedral meshes as acceleration structures in ray tracing complex 3D scenes.

- Tetrahedral mesh generation is computationally costly and requires more memory than the alternative methods. Developing a high-quality tetrahedralization tool is also complex; most tools rely on a third-party library.

- Our current implementation cannot construct a tetrahedral mesh acceleration structure for scenes with intersecting geometry. We can overcome this limitation by a pre-processing step where mesh intersections are resolved so that the resulting geometry is a Piecewise Linear Complex [92], as proposed by Lagae and Dutre [1].

Other areas for further research regarding contemporary ray-tracing concepts are as follows.

- *Non-triangular models:* The proposed acceleration structure does not support non-triangular models. Recent research by Hu et al. ([93]) provides a way to build triangulations with curve constraints. The extension of this method to 3D with surface constraints can act as an accelerator to render paramteric 3D surfaces, which could be a potentially interesting and challenging research direction.

- *Real-time rebuilds:* Although our approach allows real-time manipulation of the geometry by certain deformers (smooth, C1 continuous) naturally, it is not very easy to have real-time rebuilds on changing geometry, which is well supported by the state-of-the-art BVHs.

It is also worth mentioning that our compact tetrahedralization structure might be used in other domains where compressed and compact forms are essential. Tetrahedral meshes are widely used in volume rendering, medical visualizations, finite element methods, and other forms of simulation. Efficient and compact tetrahedral mesh representation could be helpful in such fields, especially in resource or/and bandwidth-limited environments such as streaming applications or low-end mobile devices.

Bibliography

[1] A. Lagae and P. Dutré, "Accelerating ray tracing using constrained tetrahedralizations," *Comper Graphics Forum*, vol. 27, no. 4, pp. 1303–1312, 2008.

[2] E. Angel and D. Shreiner, *Interactive Computer Graphics with WebGL*. Addison-Wesley Professional, 7th ed., 2014.

[3] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[4] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the Spring Joint Computer Conference*, AFIPS '68 (Spring), (New York, NY, USA), pp. 37–45, ACM, 1968.

[5] G. Tran, "Glasses, pitcher, ashtray and dice (POV-Ray)," 2016. Online, Available at `http://www.oyonale.com/modeles.php?lang=en&page=40`, Access date: 29 June 2022.

[6] K. Dudka, "RRV - Radiosity Renderer and Visualizer," 2016. Online, Available at `http://dudka.cz/rrv`, Access date: 29 June 2022.

[7] A. S. Glassner, ed., *An Introduction to Ray Tracing*. London, UK: Academic Press Ltd., 1989.

[8] A. Aman and U. Güdükbay, "Fast Tetrahedral Mesh Traversal for Ray Tracing," in *High-Performance Graphics, Posters*, HPG '17, 2017.

[9] A. Aman, S. Demirci, U. Güdükbay, and I. Wald, "Multi-level Tetrahedralization-based Accelerator for Ray-tracing Animated Scenes,"

Computer Animation and Virtual Worlds, vol. 32, no. 3-4, Article No. e2024, 10 pages, 2021.

[10] A. Aman, S. Demirci, and U. Güdükbay, "Compact tetrahedralization-based acceleration structures for ray tracing," *Journal of Visualization*, In Press.

[11] A. Sahistan, S. Demirci, N. Morrical, S. Zellmann, A. Aman, I. Wald, and U. Güdükbay, "Ray-traced shell traversal of tetrahedral meshes for direct volume visualization," in *Proceedings of the IEEE Visualization Conference-Short Papers*, VIS '21, pp. 91–95, IEEE, 2021.

[12] Z. Erkoç, A. Aman, U. Güdükbay, and H. Si, "Out-of-core constrained delaunay tetrahedralizations for large scenes," in *Numerical Geometry, Grid Generation and Scientific Computing* (V. A. Garanzha, L. Kamenski, and H. Si, eds.), (Cham), pp. 113–124, Springer International Publishing, 2021.

[13] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 3rd ed., 2016.

[14] A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-tracing System," in *Tutorial: Computer Graphics; Image Synthesis* (K. I. Joy, C. W. Grant, N. L. Max, and L. Hatfield, eds.), pp. 148–159, New York, NY, USA: Computer Science Press, Inc., 1988.

[15] D. Cohen and Z. Sheffer, "Proximity clouds—an acceleration technique for 3d grid traversal," *The Visual Computer*, vol. 11, no. 1, p. 27–38, 1994.

[16] A. Es and V. İşler, "Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor," *Journal of Parallel and Distributed Computing*, vol. 67, no. 11, pp. 1201–1217, 2007.

[17] A. Lagae and P. Dutré, "Compact, fast and robust grids for ray tracing," *Computer Graphics Forum*, vol. 27, no. 4, pp. 1235–1244, 2008.

[18] A. V. Aho and J. D. Ullman, *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*. USA: Addison-Wesley Longman Publishing Co., Inc., 1977.

[19] J. Kalojanov, M. Billeter, and P. Slusallek, "Two-level grids for ray tracing on GPUs," *Computer Graphics Forum*, vol. 30, pp. 307–314, 2011.

[20] A. Pérard-Gayot, J. Kalojanov, and P. Slusallek, "Gpu ray tracing using irregular grids," *Computer Graphics Forum*, vol. 36, no. 2, p. 477–486, 2017.

[21] J. Goldsmith and J. Salmon, "Automatic creation of object hierarchies for ray tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, 1987.

[22] J. Bittner and V. Havran, "RDH: Ray distribution heuristics for construction of spatial data structures," in *Proceedings of the 25th Spring Conference on Computer Graphics*, SCCG '09, p. 61–67, Association for Computing Machinery, 2009.

[23] T. Aila, T. Karras, and S. Laine, "On quality metrics of bounding volume hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, (New York, NY, USA), p. 101–107, Association for Computing Machinery, 2013.

[24] I. Wald, "On fast construction of SAH-based bounding volume hierarchies," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, RT '07, (Washington, DC, USA), pp. 33–40, IEEE Computer Society, 2007.

[25] P. Ganestam, R. Barringer, M. Doggett, and T. Akenine-Möller, "Bonsai: Rapid bounding volume hierarchy generation using mini trees," *Journal of Computer Graphics Techniques*, vol. 4, no. 3, pp. 23–42, 2015.

[26] D. J. MacDonald and K. S. Booth, "Heuristics for Ray Tracing Using Space Subdivision," *The Visual Computer*, vol. 6, no. 3, pp. 153–166, 1990.

[27] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the High-Performance Graphics*, HPG '09, (New York, NY, USA), pp. 7–13, ACM, 2009.

[28] S. Popov, I. Georgiev, R. Dimov, and P. Slusallek, "Object partitioning considered harmful: Space subdivision for bvhs," in *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*, pp. 15–22, 01 2009.

[29] D. Wodniok and M. Goesele, "Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees," *Computers & Graphics*, vol. 62, pp. 41–52, 2017.

[30] I. Wald, C. Benthin, and P. Slusallek, "Distributed interactive ray tracing of dynamic scenes," in *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pp. 77–85, 2003.

[31] C. Benthin, S. Woop, I. Wald, and A. T. Áfra, "Improved two-level bvhs using partial re-braiding," in *Proceedings of High Performance Graphics*, HPG '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[32] S. Woop, C. Benthin, I. Wald, G. Johnson, and E. Tabellion, "Exploiting local orientation similarity for efficient ray traversal of hair and fur," in *Proceedings of the High-Performance Graphics*, HPG '14, 2014.

[33] I. Wald, N. Morrical, S. Zellmann, L. Ma, W. Usher, T. Huang, and V. Pascucci, "Using Hardware Ray Transforms to Accelerate Ray/Primitive Intersections for Long, Thin Primitive Types," *Proceedings of the ACM on Computer Graphics and Interactive Techniques (Proceedings of High Performance Graphics)*, 2020.

[34] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," *Computer Graphics Forum*, vol. 28, pp. 375–384, 2009.

[35] G. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966. Online, Available at `https://books.google.co.uk/books?id=9FFdHAAACAAJ`, Access date: 29 June 2022.

[36] J. Pantaleoni and D. Luebke, "HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry," in *Proceedings of High Performance Graphics*, HPG '10, pp. 87–95, 2010.

[37] M. Vinkler, J. Bittner, and V. Havran, "Extended morton codes for high performance bounding volume hierarchy construction," in *Proceedings of High Performance Graphics*, HPG '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[38] I. Wald, J. Amstutz, and C. Benthin, "Robust iterative find-next-hit ray traversal," in *Proceedings of the Symposium on Parallel Graphics and Visualization*, EGPGV '18, (Goslar, DEU), p. 25–32, Eurographics Association, 2018.

[39] I. Wald, C. Benthin, and S. Boulos, "Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, RT '08, pp. 49–57, 2008.

[40] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Computer Graphics and Applications*, vol. 4, no. 10, pp. 15–24, 1984.

[41] K.-Y. Whang, J.-W. Song, J.-W. Chang, J.-Y. Kim, W.-S. Cho, C.-M. Park, and I.-Y. Song, "Octree-R: An adaptive octree for efficient ray tracing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, pp. 343–349, 1996.

[42] V. Havran and J. Bittner, "On improving kd tree for ray shooting," *Journal of WSCG*, vol. 10, pp. 209–216, 2002.

[43] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pp. 61–69, 2006.

[44] B. Choi, R. Komuravelli, V. lu, H. Sung, R. Jr, S. Adve, and J. Hart, "Parallel SAH k-D tree construction," in *Proceedings of the Conference on High Performance Graphics*, HPG '10, pp. 77–86, 2010.

[45] E. Haines and D. Greenberg, "The light buffer: A shadow-testing accelerator," *IEEE Computer Graphics and Applications*, vol. 6, no. 9, pp. 6–16, 1986.

[46] W. Hunt and W. Mark, "Adaptive Acceleration Structures in Perspective Space," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, RT '08, pp. 11–17, 2008.

[47] W. Hunt and W. Mark, "Ray-specialized Acceleration Structures for Ray Tracing," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, RT '08, pp. 3–10, 2008.

[48] K. Klimaszewski and T. Sederberg, "Faster ray tracing using adaptive grids," *IEEE Computer Graphics and Applications*, vol. 17, no. 1, pp. 42–51, 1997.

[49] D. Lin, K. Shkurko, I. Mallett, and C. Yuksel, "Dual-split trees," in *Proceedings of the Symposium on Interactive 3D Graphics and Games*, I3D '19, (New York, NY, USA), ACM Press, 2019.

[50] D. Lin, E. Vasiou, C. Yuksel, D. Kopta, and E. Brunvand, "Hardware-accelerated dual-split trees," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 2, Article No. 20, 21 pages, pp. 20:1–20:21, 2020.

[51] J. Arvo and D. Kirk, "Fast ray tracing by ray classification," *ACM Computer Graphics (Proceedings of SIGGRAPH '87*, vol. 21, no. 4, pp. 55–64, 1987.

[52] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Transactions on Graphics*, vol. 24, no. 3, p. 1176–1185, 2005.

[53] T. Ize, I. Wald, and S. G. Parker, "Ray tracing with the bsp tree," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pp. 159–166, 2008.

[54] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," in *Proceedings of Eurographics '87*, pp. 3–10, 1987.

[55] B. Arnaldi, T. Priol, and K. Bouatouch, "A new space subdivision method for ray tracing CSG modelled scenes," *The Visual Computer*, vol. 3, no. 2, pp. 98–108, 1987.

[56] H. Si, "TetGen, a Delaunay-based quality tetrahedral mesh generator," *ACM Transactions on Mathematical Software*, vol. 41, no. 2, Article no. 11, 36 pages, 2015.

[57] A. Bowyer, "Computing Dirichlet tessellations," *The Computer Journal*, vol. 24, no. 2, pp. 162–166, 1981.

[58] D. F. Watson, "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes," *The Computer Journal*, vol. 24, no. 2, pp. 167–172, 1981.

[59] H. Edelsbrunner and N. R. Shah, "Incremental topological flipping works for regular triangulations," *Algorithmica*, vol. 15, p. 223–241, 1996.

[60] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete & Computational Geometry*, vol. 18, pp. 305–363, 1996.

[61] M. Maria, S. Horna, and L. Aveneau, "Efficient ray traversal of constrained Delaunay tetrahedralization," in *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, vol. 1 of *VISIGRAPP '17*, pp. 236–243, 2017.

[62] M. Maria, S. Horna, and L. Aveneau, "Constrained convex space partition for ray tracing in architectural environments," *Computer Graphics Forum*, vol. 36, no. 1, pp. 288–300, 2017.

[63] M. Maria, S. Horna, and L. Aveneau, "Topological space partition for fast ray tracing in architectural models," in *Proceedings of the International Conference on Computer Graphics Theory and Applications*, GRAPP '14, pp. 1–11, 2014.

[64] C. T. Silva, J. S. B. Mitchell, and A. E. Kaufman, "Fast rendering of irregular grids," in *Proceedings of the Symposium on Volume Visualization*, pp. 15–22, 1996.

[65] C. T. Silva and J. S. B. Mitchell, "The lazy sweep ray casting algorithm for rendering irregular grids," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 2, pp. 142–157, 1997.

[66] H. Berk, C. Aykanat, and U. Gudukbay, "Direct volume rendering of unstructured grids," *Comp. & Graph.*, vol. 27, pp. 387–406, 2003.

[67] K. Koyamada, "Fast traverse of irregular volumes," in *Visual Computing* (T. L. Kunii, ed.), (Tokyo), pp. 295–311, Springer Japan, 1992.

[68] M. P. Garrity, "Raytracing irregular volume data," in *Proceedings of the Workshop on Volume Visualization*, VVS '90, (New York, NY, USA), pp. 35–40, ACM, 1990.

[69] S. Ribeiro, A. Maximo, C. Bentes, A. Oliveira, and R. Farias, "Memory-aware and efficient ray-casting algorithm," in *Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing*, SIBGRAPI '07, pp. 147–154, 2007.

[70] A. Maximo, S. Ribeiro, C. Bentes, A. Oliveira, and R. Farias, "Memory efficient GPU-based ray casting for unstructured volume rendering," in *Proceedings of the IEEE/ EG Symposium on Volume and Point-Based Graphics*, SPBG '08, (Goslar, DEU), pp. 155–162, Eurographics Assoc., 2008.

[71] G. Marmitt and P. Slusallek, "Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering," in *Proceedings of the Eighth Joint Eurographics / IEEE VGTC Symposium on Visualization*, EUROVIS '06, (Aire-la-Ville, Switzerland), pp. 235–242, Eurographics Assoc., 2006.

[72] N. Platis and T. Theoharis, "Fast ray-tetrahedron intersection using Plücker coordinates," *Journal of Graphics Tools*, vol. 8, no. 4, pp. 37–48, 2003.

[73] C. Garth and K. I. Joy, "Fast, memory-efficient cell location in unstructured grids for visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1541–1550, 2010.

[74] R. Fellegara, L. D. Floriani, P. Magillo, and K. Weiss, "Tetrahedral trees: A family of hierarchical spatial indexes for tetrahedral meshes," *ACM Transactions on Spatial Algorithms and Systems*, vol. 6, no. 4, Article no. 23, 34 pages, 2020.

[75] P. Sinha, "A memory-efficient doubly linked list," *Linux Journal*, vol. 2005, no. 129, p. 10, 2005.

[76] A. Mebarki, "XOR-based compact triangulations," *Computing & Informatics*, vol. 37, pp. 367–384, 2018.

[77] T. Duff, J. Burgess, P. Christensen, C. Hery, A. Kensler, M. Liani, and R. Villemin, "Building an orthonormal basis, revisited," *Journal of Computer Graphics Techniques*, vol. 6, no. 1, pp. 1–8, 2017.

[78] A. Jacobson, D. Panozzo, *et al.*, "libigl: A simple C++ geometry processing library," 2018. Online, Available at `https://libigl.github.io/`, Access date: 29 June 2022.

[79] B. O. Community, "Blender - a 3D modelling and rendering package," 2018. Online, Available at `http://www.blender.org`, Access date: 29 June 2022.

[80] F. Hwang, D. Richards, and P. Winter, *The Steiner tree problem*. Annals of Discrete Mathematics, Burlington, MA: Elsevier, 1992.

[81] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second ed., 2000.

[82] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[83] D. Hilbert, "Ueber die stetige abbildung einer line auf ein flächenstück," *Mathematische Annalen*, vol. 38, no. 3, pp. 459–460, 1891.

[84] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, RT '07, (Washington, DC, USA), pp. 113–118, IEEE Computer Society, 2007.

[85] Y. Hu, Q. Zhou, X. Gao, A. Jacobson, D. Zorin, and D. Panozzo, "Tetrahedral meshing in the wild," *ACM Transactions on Graphics*, vol. 37, no. 4, pp. Article no. 60, 14 pages, 2018.

[86] NVIDIA, "NVIDIA TURING GPU ARCHITECTURE.." Online, Available at `https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, Access date: 29 June 2022.

[87] NVIDIA, "NVIDIA OptiX 6.0–Programming Guide.." Online, Available at `https://raytracing-docs.nvidia.com/optix7/guide/index.html`, Access date: 29 June 2022.

[88] A. Sahistan, "Hardware-accelerated direct visualization of unstructured volumetric meshes," Master's thesis, Department of Computer Engineering, Bilkent University, 2022.

[89] S. Demirci, "Bounding volume hierarchy-tetrahedralization hybrid acceleration structure for ray tracing," Master's thesis, Department of Computer Engineering, Bilkent University, 2020.

[90] The CGAL Project, *CGAL User and Reference Manual*. CGAL Editorial Board, 5.4.1 ed., 2022. Online, Available at `https://doc.cgal.org/5.4.1/Manual/packages.html`, Access date: 29 June 2022.

[91] Z. Erkoç, "Memory-efficient constrained delaunay tetrahedralization of large three-dimensional triangular meshes," Master's thesis, Department of Computer Engineering, Bilkent University, 2022.

[92] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, and H. Wang, "Control volume meshes using sphere packing: Generation, refinement and coarsening," in *Proceedings of the 5th International Meshing Roundtable*, pp. 47–61, 1996.

[93] Y. Hu, T. Schneider, X. Gao, Q. Zhou, A. Jacobson, D. Zorin, and D. Panozzo, "TriWild: Robust triangulation with curve constraints," *ACM Trans. Graph.*, vol. 38, no. 4, pp. Article no. 52, 15 pages, 2019.

[94] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. Online, Available at `https://developer.nvidia.com/cuda-toolkit`, Access date: 29 June 2022.

[95] C. Guillemet, "ImGuizmo: Immediate mode 3D gizmo for scene editing and other controls based on Dear Imgui." Online, Available at `https://github.com/CedricGuillemet/ImGuizmo`, Access date: 29 June 2022.

[96] O. Cornut, "Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies." Online, Available at `https://github.com/ocornut/imgui`, Access date: 29 June 2022.

[97] S. Kaslev, "gl3w: Extension wrangler for OpenGL," 2017. Online, Available at `https://github.com/skaslev/gl3w`, Access date: 29 June 2022.

[98] Camilla Löwy, Marcus Geelnard, "GLFW: an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop." Online, Available at `https://www.glfw.org`, Access date: 29 June 2022.

[99] G-Truc, "OpenGL Mathematics (GLM)," 2020. Online, Available at `https://github.com/g-truc/glm`, Access date: 29 June 2022.

[100] G. Chereau, Z. Shahaf, and A. Belt, "Noc File Dialog: A portable library to create open and save dialogs on Linux, OS X and Windows," 2019. Available at `https://github.com/guillaumechereau/noc/blob/master/noc_file_dialog.h`, Access date: 29 June 2022.

[101] S. T. Barrett, "STB: single-file public domain libraries for C/C++." Online, Available at `https://github.com/nothings/stb`, Access date: 29 June 2022.

[102] S. Fujita, "tinyOBJLoader: Tiny but powerful single file Wavefront obj loader." Online, Available at `https://github.com/tinyobjloader/tinyobjloader`, Access date: 29 June 2022.

Appendix A

Ray-tracing Toolkit

We built a raytracer, called *Neptun*, with editing capabilities to build and interact with 3-D scenes and render them.

A.1   Editor

Neptun editor has the following capabilities:

1. User-friendly interface to create and edit 3-D scenes. It can export .obj files and supports the generation of procedural shapes such as grids and icosahedrons with different subdivision levels. As found in popular 3-D content generation tools, those shapes can be selected visually and manipulated using transformation gizmos. Inspector windows allow the user to edit properties of scene objects such as position, material colors, and light intensity.

2. Build and analyze different accelerators. Neptun can construct different types of accelerators (BVH, k-d trees and tetrahedral mesh-based accelerators). The leaves and nodes of these accelerators can be visualized

separately from the scene objects for better understanding and analysis. Parameters of these accelerators can be modified using the user interface.

3. Render images and generate diagnostic images. Neptun can render 3-D scenes using different types of accelerators and possibly different rendering techniques such as basic raycasting, recursive ray tracing, or simple path tracing. Figure A.1 top part shows scene objects and the tetrahedral mesh-based accelerator for a scene. It can also generate diagnostic images to analyze the efficiency of the acceleration structures (cf. Figure A.1 bottom part). For tree-based accelerators, it creates a diagnostic image out of the tree nodes traversed. It creates the diagnostic images for tetrahedral meshes using the number of traversed tetrahedra per pixel.

4. Neptun is cross-platform, meaning it can be built and used in Windows, Linux, and macOS. Neptun also can utilize the GPU for accelerating ray-intersection tests. GPU support is only available through CUDA [94] platform.

A.2   Accelerator Interface

Accelerator interfaces provide a convenient way to use accelerator functions independent of the accelerator structure underneath. This interface supports single, and packet ray traversal queries and provides an output format to retrieve and process traversal data. Figure A.2 shows the Unified Modeling Language (UML) diagram for the accelerator interface that shows the relationship between acceleration structure classes.

A.3   Command-line Interface

Neptun can be used in *headless* mode, meaning that it can be run without needing a user interface. Such a mode is useful for experiments and benchmarks. An
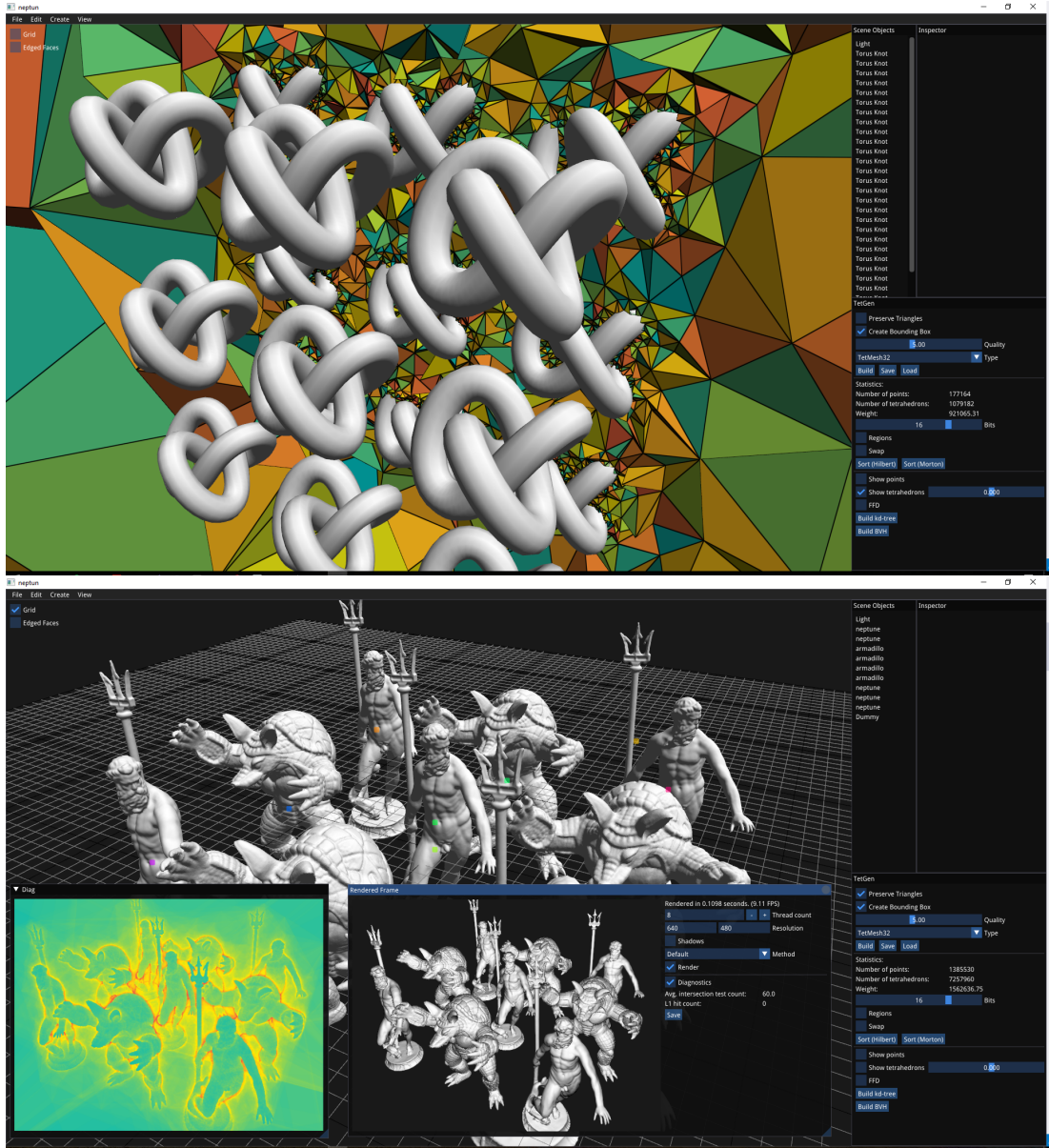
Figure A.1: Top: Neptun editor showing scene objects and the tetrahedral mesh-based accelerator built for the scene. Bottom: Neptun editor showing a scene along with the rendered image and accompanying diagnostic image showing the rendering heatmap (based on the number of traversal steps).
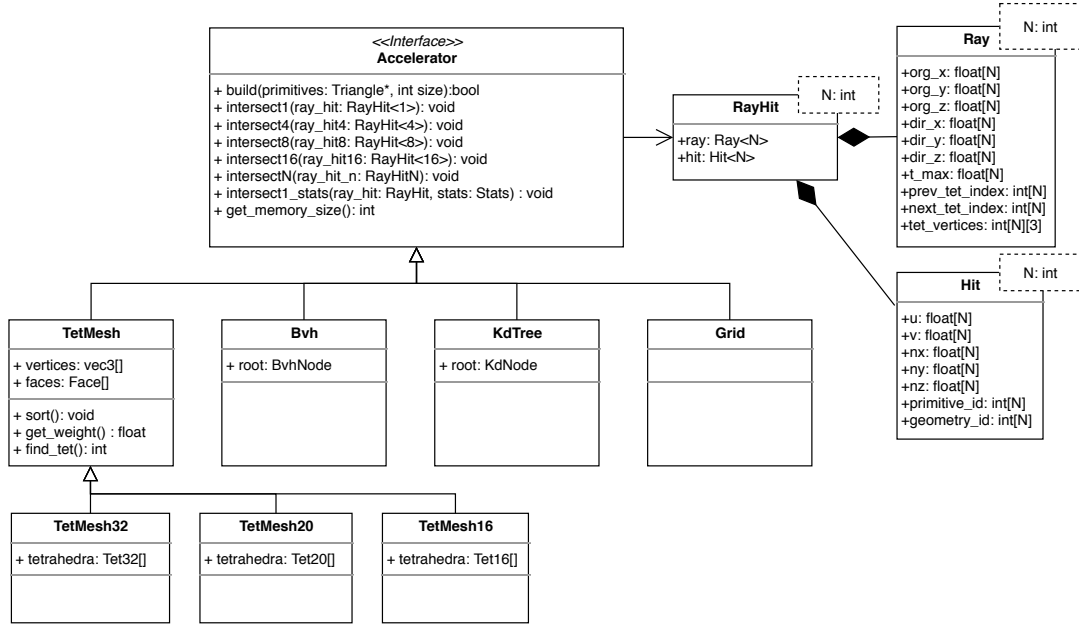
Figure A.2: Unified Modeling Language (UML) diagram of the accelerator interface.

external scripting language can be used to invoke Neptun commands through the command line and can perform many experiments in a single run. For convenience and reliability, we use extensive Python scripts to manage many tests for detailed comparison. The command-line interface allows the following parameters to be specified by the user.

Scene file: Input file path of the scene to be rendered. We use a custom scene format to describe the 3-D scene. If available, it is possible to use a cached version of the accelerator.

Accelerator: Specifies the ray tracing accelerator type: Tetrahedral mesh, *k*-d tree or Bounding Volume Hierarchy (BVH). Tetrahedral mesh-based accelerators have sub-types such as TetMesh32 and TetMesh16.

Output file: Specifies the output file path for the rendered images. The resulting output is often used to check for differences that might point to the problems in different techniques.

Test: Instructs Neptun to output diagnostic images used to analyze the accelerator performance. The resulting diagnostic images show the number of intersected nodes using a heatmap for different types of accelerators used.

Resolution: Resolution of the rendered image. By default, we use 1920×1440 pixels.

Thread-count: Number of threads to be used by the rendered. By default, all available threads are used. Each thread is assigned to render the next available tile in the image.

Repetition count: For more reliable results, Neptun can be instructed to render a scene many times. For rendering, it is often advised to use the shortest render time as a rule of thumb since it is more robust to the effect of other processes running on the system that uses the computing resources.

Sorting: This is used to reorder the tetrahedral mesh data (points and tetrahedra) using a space-filling curve. Neptun supports the Hilbert Curve and the Morton curve.

Tetmesh cache: Tells Neptun to use a prebuilt tetrahedral mesh-based accelerator when rendering a scene. This option reduces the time needed to evaluate tetrahedral mesh-based accelerators.

Quality: Specifies the tetrahedral mesh quality parameter. This number dictates the maximum-radius-edge ratio of the tetrahedrons in the resulting tetrahedral mesh.

Split-triangles: If enabled, the tetrahedralization method can modify and subdivide constrained faces to improve tetrahedral mesh quality. This is analogous to having spatial splits in BVHs.

Use-gpu: If enabled, Neptun will use GPU accelerated ray intersection tests. Right now, only CUDA-capable GPUs are supported in this mode.

A.4 Third-party Libraries

Neptun rendering tool relies on the following libraries for various purposes.

ImGuizmo — Immediate mode 3D gizmo for scene editing and other controls based on Dear Imgui [95]: This library provides a convenient and straightforward way of creating transform manipulation gizmos for 3-D applications that use Dear ImGui [96].

gl3w — Simple OpenGL core profile loading [97]: gl3w provides an easy way to use the functionality offered by the OpenGL core profile specification.

GLFW — An OpenGL library [98]: GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES, and Vulkan development on the desktop. It provides a simple Application Programming Interface (API) for creating windows, contexts, and surfaces, receiving input and events.

OpenGL Mathematics (GLM) [99]: GLM is a header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

Dear ImGUI — Bloat-free graphical user interface library for C++ (GLM) [95]: GLM is a header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

noc file dialog (noc) [100]: noc is a portable library to create open and save dialogs on Linux, OS X, and Windows.

STB Image (GLM) [101]: GLM is a header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

TetGen [56]: TetGen is a tetrahedralization library that can generate Delaunay tetrahedralization, Voronoi diagram, and convex hull for three-dimensional point sets. It can generate the constrained Delaunay tetrahedralizations and quality tetrahedral meshes for three-dimensional domains with piecewise linear boundaries.

tinyOBJ Loader (GLM) [102]: Tiny but powerful single file wavefront obj loader, written in C++03. No dependency except for C++ STL. It can parse over 10M polygons with moderate memory and time.