Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Using Tight Bounding Volumes for Bounding Volume Hierarchies

Master's Thesis

Lucie Veverková

Field of study: Computer Graphics
Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Prague, May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Veverková Lucie**	Personal ID number:	**483797**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Graphics**		

II. Master’s thesis details

Master’s thesis title in English:

Using Tight Bounding Volumes for Bounding Volume Hierarchies

Master’s thesis title in Czech:

Využití těsných obálek pro hierarchie obalových těles

Guidelines:

Review the methods for building bounding volume hierarchies (BVH) for ray tracing. Focus on techniques applicable to interactive applications with dynamic scenes.
Implement efficient BVH construction algorithms that use tight bounding volumes such as OBB [1] and ODOP [2]. For the implementation, use the available codes in the CUDA language and unify them inside a common framework. Compare the resulting BVHs in terms of their construction speed and ray tracing speed on at least six different scenes. Identify bottlenecks of BVH construction algorithms and suggest possible improvements.

Bibliography / sources:

[1] Vitsas, N., Evangelou, I., Papaioannou, G., & Gkaravelis, A. (2023). Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees. In Computer Graphics Forum (Vol. 42, No. 2, pp. 245-254).
[2] Sabino, R., Vidal, C. A., Cavalcante-Neto, J. B., & Maia, J. G. R. (2023). Building Oriented Bounding Boxes by the intermediate use of ODOPs. Computers & Graphics, 116, 251-261.
[3] Benthin, C., Drabinski, R., Tessari, L., & Dittebrandt, A. (2022). PLOC++ Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited. Proceedings of the ACM on Computer Graphics and Interactive Techniques, 5(3), 1-13.
[4] Meister, D., & Bittner, J. (2017). Parallel locally-ordered clustering for bounding volume hierarchy construction. IEEE transactions on visualization and computer graphics, 24(3), 1345-1353.
[5] Apetrei, C. (2014). Fast and simple agglomerative LBVH construction.
[6] Vinkler, M., Bittner, J., & Havran, V. (2017). Extended Morton codes for high performance bounding volume hierarchy construction. In Proceedings of high performance graphics (pp. 1-8).
[7] Meister, D., Ogaki, S., Benthin, C., Doyle, M. J., Guthe, M., & Bittner, J. (2021, May). A survey on bounding volume hierarchies for ray tracing. In Computer Graphics Forum (Vol. 40, No. 2, pp. 683-712).

Name and workplace of master’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **15.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

doc. Ing. Jiří Bittner, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration

I hereby declare I have written this Master's thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis.

Abstract

This thesis reviews methods for building bounding volume hierarchies (BVH) for ray tracing. It focuses on techniques applicable to interactive applications with dynamic scenes. Based on the research of these techniques, two efficient BVH construction algorithms that use tight bounding volumes, such as OBB and ODOP, are implemented. For the implementation, available codes in the CUDA language were used and unified inside a common framework. The resulting BVHs were compared in terms of their construction speed and ray tracing speed on six different scenes. The bottlenecks of this construction were discussed, and possible improvements were suggested.

Keywords: Bounding volume hierarchy, OBB, ODOP, ray tracing, CUDA.

Abstrakt

Tato práce se zabývá metodami stavby hierarchie obálek (BVH), které jsou využitelné v algoritmech sledování paprsku. Soustředí se na techniky aplikovatelné pro interaktivní aplikace pracující s dynamickými scénami. Na základě důkladné analýzy těchto technik dva efektivní algoritmy na stavbu BVH, využívající těsná obálková tělesa, jako například OBB a ODOP, byly naimplementovány. Pro implementaci a integraci těchto metod do jednotného frameworku byly využity dostupné kódy v jazyce CUDA. Výsledné BVH jsou porovnány na základě časové náročnosti jejich stavby a rychlosti sledování paprsku otestované na šesti různých scénách. Práce dále diskutuje takzvaná úzká místa jejich konstrukce a navrhuje možná vylepšení.

Klíčová slova: Hierarchie obálek, OBB, ODOP, sledování paprsku, CUDA.

Acknowledgements

I want to express my deepest appreciation to my supervisor, doc. Ing. Jiří Bittner, Ph.D., for his great insights into the topic of this thesis and his valuable advice. I am also grateful to my family and friends who supported me throughout the whole period of writing this thesis.

I would like to extend a special thank you to Daniel Meister and Jiří Bittner who made the source codes of the PLOC algorithm publicly available.

List of Figures

Contents

Chapter 1

Introduction

In the fast changing and highly competitive field of computer graphics, achieving real-time rendering with realistic results is a necessity. This is a challenging process that requires special approaches to the rendering. This thesis aims at exploring these approaches focusing on the optimization of ray tracing techniques which produce highly photo-realistic images.

1.1 Motivation

As the realistic approach to rendering scenes is more and more aimed at the gaming and entertainment industry, new and improved methods have to be introduced to render the scene quickly to meet the demanding requirements.

The ray tracing method is one approach to realistic rendering as it enables achieving global illumination in the scene. Figure 1.1 shows a photo-realistic image rendered using the ray tracing method.

However, naive ray tracing is very slow, and acceleration data structures enabling fast traversal of the rays through the scene are essential during this process. There are many options for the different data structures, each with advantages and disadvantages fitting more into different algorithms and their expected results.

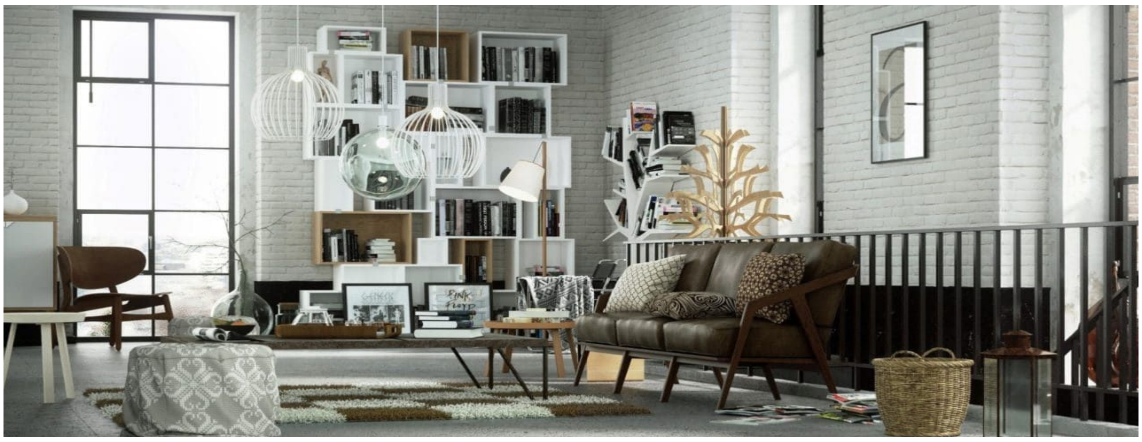


Figure 1.1: This computer-generated image, created by Enrico Cerica using OctaneRender, shows realistic ray tracing techniques, including glass distortion, diffuse lighting and frosted glass. Image taken from NVIDIA Blog, 2018 [26].

One of these data structures is the bounding volume hierarchy structure (BVH). This structure uses bounding volumes to encapsulate the scene primitives in geometry objects. It makes it possible for the ray to quickly and effectively traverse through the hierarchy to the intersected leaves.

Many algorithms have been presented implementing the construction of the bounding volume hierarchy data structure. One of them is the *Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction* (PLOC) algorithm that uses the axis-aligned bounding box (AABB) as the bounding volume encapsulating the scene objects [20].

However, even though the axis-aligned bounding volume offers simplicity and efficiency in the BVH construction and handling ray tracing queries, these volumes don't offer as tight fit as some other bounding volumes.

There is a wide possibility of different types of bounding volumes that can be used in the hierarchy structure. One of them is the oriented bounding box (OBB), which generally creates tighter-fitting encapsulation of the scene objects thanks to its adaptable orientation. The use of this type of bounding volume brings its benefits as well as drawbacks.

In scenarios where tighter bounding volumes are preferred while the time of the BVH construction isn't as critical, OBBs are a perfect option. OBBs bring a good balance between the tightness of the volume over the geometry. These volumes reduce the number of intersection tests while increasing the construction time. This fastens up the entire rendering process with a decrease in the construction performance.

1.2 Goals of the Thesis

Specialised BVH builders are essential for the construction of OBB BVH to keep the construction time bearable as their construction is complex and computationally intensive. This thesis focuses on such builders, exploring methods that significantly reduce the number of intersection tests during the ray tracing process.

One of discussed methods, the method proposed by Vitsas et al. 2023 [34], transforms an already existing AABB BVH into an OBB BVH. Another approach is brought by method developed by Sabino et al. 2023 [28], which builds the OBB BVH using ODOPs with the use of the PLOCTree algorithm.

This thesis discusses these methods for constructing OBB BVH along with the PLOC algorithm which produces AABB BVH. It details their specifics with their advantages for the BVH construction. This data structure is then used for rendering 3D scenes with the ray tracing rendering method.

The goal of the thesis is the implementation of these two methods and comparison of the resulting BVHs in terms of their construction speed and ray tracing speed on eight different scenes. It aims at identifying bottlenecks of this construction and suggesting possible improvements.

1.3 Structure of the Thesis

The first part of the thesis presents the theoretical background with a brief description of the ray tracing method and acceleration data structures, focusing on the bounding volume hierarchy. The description of Morton Codes follows with the introduction to the Agglomerative Clustering used in the PLOC algorithm.

Then, the thesis introduces related work in chapter chapter 3. It gives details of the algorithm propsed by Vinkler et al. 2017 [33] with the PLOC algorithm. Finally, it explains the DiTO algorithm proposed by Lengyel, 2011 which is exploited in one of the methods.

The chapter 4 follows, introducing both methods for the OBB BVH construction. It presents the details of both methods important for their implementation.

The final implementation is described in chapter 5. The details of the implementations are provided of how the methods were implemented, taking advantage of an already implemented PLOC algorithm project implemented by Daniel Meister et al. [5].

The results of the implementation tested and measured on eight scenes follow in chapter 6 chapter, along with the project's conclusion written in chapter 7.

Chapter 2

Theoretical Background

This chapter sets a theoretical background necessary for understanding the technical terms used throughout this thesis. It defines and describes the ray tracing process which produces a realistic images of a rendered 3D scene. Another method used for rendering images is the Path tracing used in many modern realistic rendering methods.

The chapter follows by presenting optimizations of the ray tracing algorithms where the acceleration data structures stand out. It discusses different acceleration data structures with their own variations and specifics, and focuses on one specific data structure, the bounding volume hierarchy. An important metric of a quality of a data structure which tells how fast the ray tracing will be over this data structure is a surface area heuristic. This chapter finishes with talking about this heuristic.

2.1 Ray Tracing

There are many methods of rendering a 3D scene. One of them is ray tracing, a widely used technique in modern graphics ensuring realistic visualisation of a 3D scene consisting of scene primitives. This method uses rays shot for each pixel of the scene. The rays are then followed until the intersection with a scene primitive is obtained. This method produces much more realistic results than rasterizing methods. The details of this method are described in this section. Additionally, there is also an improved version of ray tracing using a statistical approaches producing even more realistic results: path tracing. This section mentions this techniques as well.

2.1.1 Ray tracing variants

The ray tracing method is a method of computing global illumination in the scene. The ray tracing algorithm has two main variants, tracing of primary rays and tracing of secondary rays [37].

Tracing of primary rays (Ray casting): An idea of ray casting is shooting a ray from the camera's position through every pixel. The method aims to find the closest object hit by the ray. This means finding all intersections of the ray with the scene primitives and selecting the closest intersected object to the camera's position. The resulting colour of the pixel is computed using illumination models like the Phong model.

The naive approach means testing every scene primitive for an intersection with the ray. The time complexity for this naive implementation is $O(mn)$, where M is the number of scene primitives, and N is the number of shot rays.

Whitted Ray Tracing: As the primary rays don't consider any reflections or refractions, the rendered result does not look realistic if there are some indirect effects in question. The algorithm does not stop when the ray intersects an object in this method. It follows other "secondary" rays bounced off the surface or transmitted and scattered through the object. Additionally, to create realistic shadows, light sources add to the colour only if they illuminate the position of the intersection. To account for this, shadow rays are shot directly towards all the light sources in the scene, with the origin in the hit position. The computed colour is an addition of the colours obtained from these rays. This approach accounts for shadows and reflections or refractions, which makes the rendered result more realistic.

Both primary and secondary rays are visualized in Figure 2.1.

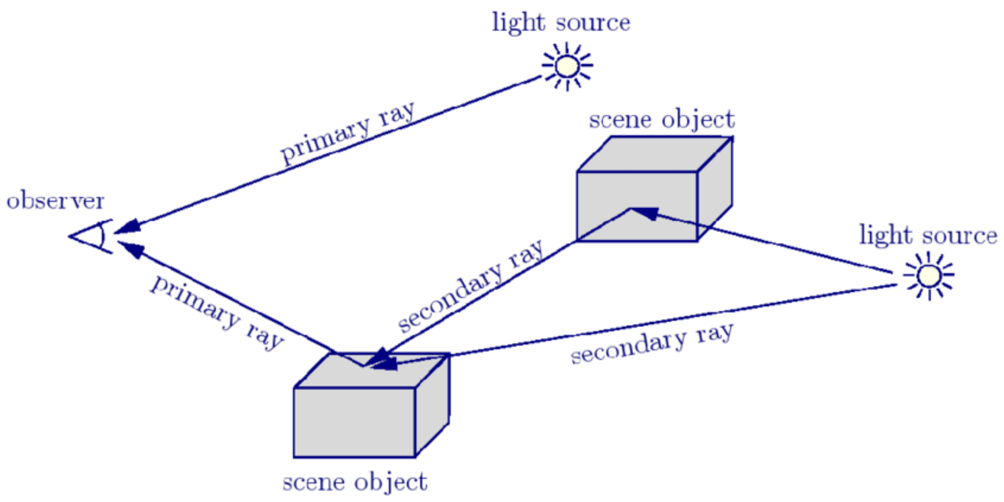


Figure 2.1: Visualization of ray tracing. Primary rays are traced back from the camera to the scene until an intersection with an object is found. The secondary rays originate at the intersection point. Shadow rays are shot in the direction of light sources to "shade" the ray. The other secondary rays computing reflections and refraction effects are shot following the laws of reflection and refraction. This makes the method recursive, as more rays can be shot from the intersection points of these secondary rays. Image taken from T. Müller, 2006 [24].

2.1.2 Algorithm details

In terms of computer graphics, ray tracing usually stands for the variant of ray tracing with casting secondary rays. The ray $R(t) = O + tD$ is described by its origin O and direction D, where $t \geq 0$ defines the distance along the ray. Then, point $P = R(t)$ lies on the ray [35].

The algorithm starts by generating rays that are shot from the camera through each pixel. This means computing the ray's direction based on the camera's position in the scene and the pixel it is evaluated for [11].

The ray cast to the scene is tested against scene primitives to find the closest intersection along the ray's direction from the ray's origin. If no intersection is found, the evaluated pixel is assigned the background color.

At the point where the ray intersects the scene primitive, the shadow rays are shot to each light source in the scene. If the first intersection of the shadow ray with another scene object is the light source, the light source lights the point and affects the displayed colour in the point. Otherwise, the point lies in the shadow of the light, and the light does not contribute to the resulting colour-computing equation. The shadow rays are responsible for rendering cast shadows.

After the contributions of the light sources are computed, the reflected and refracted rays are cast from the point, and the whole process is repeated until the maximal depth of the recursion is reached. The maximal depth of the recursion is set to stop the high repetition of the reflections of the secondary rays from objects. The secondary reflected and refracted rays are responsible for rendering reflections of other objects on the surface.

The resulting colour displayed in the pixel is evaluated as the sum of the colours obtained from the shadow rays and the refracted and reflected rays.

The ray tracing method has a great advantage in the possibility of computing the global illumination of the scene and the computation of reflections of other objects on the surface or realistically visualizing transparent objects. It can also visualize caustics appearing when the light is focused and then reflected by a specular surface [27].

This means that the ray tracing method is used when the realistic rendering is aimed to achieve even despite the downgrade in performance compared to the rasterization method.

Thanks to the high programmability of GPUs driven mainly by the demand of the computer games industry, it is possible to use GPU for ray tracing and photon mapping [35].

2.1.3 Path Tracing

An improved version of ray tracing which correctly evaluates the indirect illumination is called path tracing. It uses statistical techniques to solve a rendering equation that takes into account how heat spreads in an environment. The rendering equation mimics how light passes through the air and scatters from surfaces [25].

The equation is solved along the path of a satisfactory number of individual rays to approximate the lighting in the scene accurately with ray tracing technique.

This is done using the Monte Carlo method which is used to solve the equation. This brings an unbiased image estimate by generating random reflection/refraction directions.

It shoots multiple rays per pixel generating many primary samples and one secondary ray per hit point which resembles single light path. [37].

A comparison between this technique and simple ray tracing and rasterization is visible in Figure 2.2.

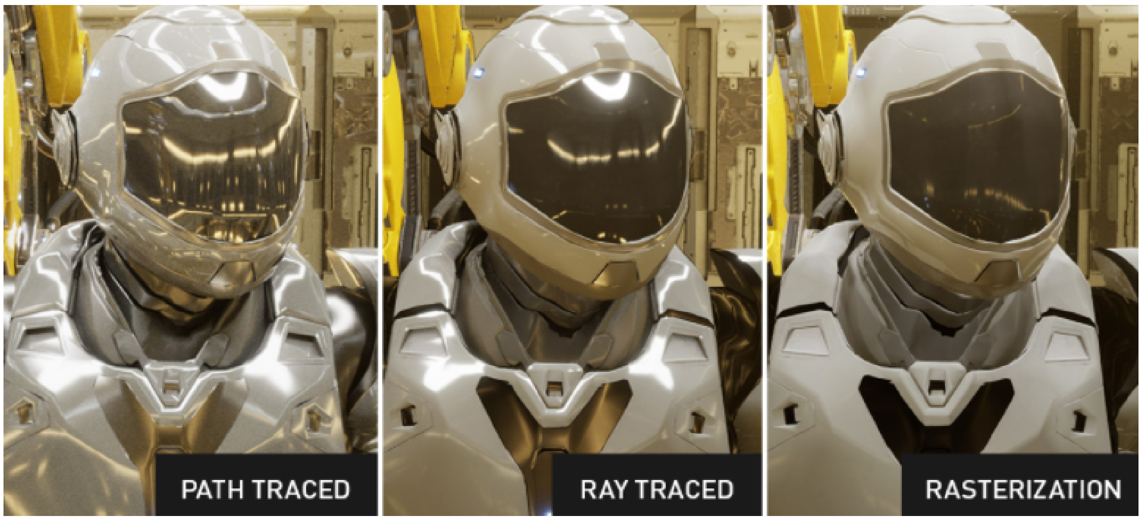


Figure 2.2: A trio of images rendered using different rendering techniques: path tracing (on the left) which produces the most realistic image, ray tracing (in the middle), and rasterization (on the right). Image taken from NVIDIA Blog, 2022 [25].

2.2 Acceleration Data Structures

Tracing a ray in a scene is a resource-intensive process. When done naively without any data organization technique, each ray needs to be tested against every scene primitive. The time complexity of the naive approach is O(mn), where m is the number of scene primitives, and n is the number of shot rays. This would make rendering each frame very time-consuming, even with a small number of triangles in the scene. This is especially true for scenes with millions of triangles.

Performance can be improved by organizing the data in a scene region or by arranging the data into an acceleration data structure. These methods significantly help speed up ray tracing. During ray tracing, groups of triangles are ruled out by pruning branches of the structure that are not intersected by the ray. When the ray does not intersect the region or bounding volume represented by one branch of the structure, it cannot intersect the primitives within this region or the bounding volume.

The ray-shooting algorithm's worst-case complexity is $O(logN)$. On the contrary, the algorithm's space complexity and preprocess time increase [31].

This means that a significant downgrade for these data structures is the time needed for their construction and the storage requirement. The more effective the acceleration data structure is, the longer it takes to construct it [37].

One example of a high-quality acceleration structure is a kd-tree. The best algorithm for constructing this high-quality acceleration structure has $O(NlogN)$ time complexity [4].

Shooting a ray through a data structure: The ray is followed from its origin along its direction vector. It is tested against the root of the structure for an intersection. If the ray does not intersect the bounding volume of the root, it does not intersect any scene primitive, and the algorithm ends. If the ray intersects the root's volume, it is nested into the hierarchy and recursively tested against the current node's children. This process is repeated until the ray reaches a leaf. At this stage, it undergoes testing against all the scene primitives stored in the leaf to determine an intersection.

Figure 2.3 depicts the process of shooting a ray through a data structure. Simple improvement, such as partitioning the scene into a regular grid, significantly enhances the performance of ray tracing by reducing the number of ray-primitive intersections.

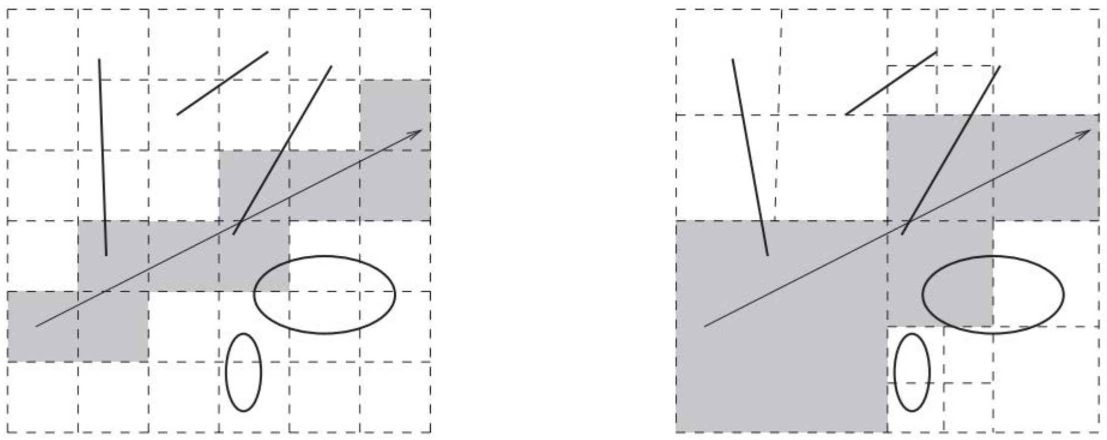


Figure 2.3: Shooting a ray through a grid structure on the left and a more complex structure, particularly octree, on the right. The grey regions visualize the intersected regions in the structure. Source: Elmar Langetepe and Gabriel Zachmann, 2006 [14].

2.2.1 Partitioning techniques

Three methods exist for organizing data within a scene based on the partitioned elements. The data structure can be created by organizing either space or objects or through a combination of both approaches.

Spatial partitioning technique: This technique, as the name suggests, primarily partitions the space. It subdivides the scene into grids or hierarchical regions that do not overlap, fully covering the original spatial region. However, some scene primitives may be split between multiple regions, resulting in duplicated primitives being stored in the leaves. Examples of spatial partitioning data structures include the Uniform grid, Binary Space Partitioning Tree (which splits the space into two equal regions by inserting a plane in the middle of the current region), and octree (which subdivides the space into eight equal regions). The kd-Tree is a special case of a BSP tree in which the splitting plane is moved in one direction along one of the axes. Figure 2.3 shows an example of an octree (quadtree in 2D) and a Uniform grid.

Object hierarchies: These structures are created by partitioning scene primitives into subsets. In this technique, overlapping regions can be created in the process, and some spatial areas do not have to be covered by the final structure. The leading example of this technique is the Bounding Volume Hierarchy.

2.2.2 Bounding Volume Hierarchy

Bounding volume hierarchy is a tree-like data structure encapsulating all scene objects into a hierarchy. It stores the scene primitives in leaves and the enclosing bounding volume in internal nodes. The bounding volumes are grouped into larger volumes with each higher tree level, resulting in storing only one large bounding volume in the tree's root.

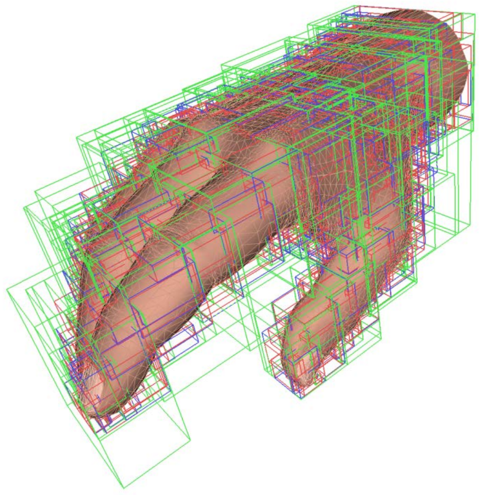


Figure 2.4: Visualization of bounding volume hierarchy built over a hand model, courtesy of Jeremy Fisher et al, 2011 [9].

As mentioned, using an organizing data structure like BVH can significantly improve ray-tracing performance. With the algorithm being able to prune out branches where the ray does not intersect, the time complexity improves from linear to logarithmic, nearly matching the complexity of a kd-tree [21].

Other benefits of BVH are easy parallelization and the memory efficiency of the structure, requiring storing mostly only minimal and maximal values of the bounding volume in the internal nodes.

There are two different approaches to constructing the hierarchy. It can be built up either from top to bottom or from bottom to top.

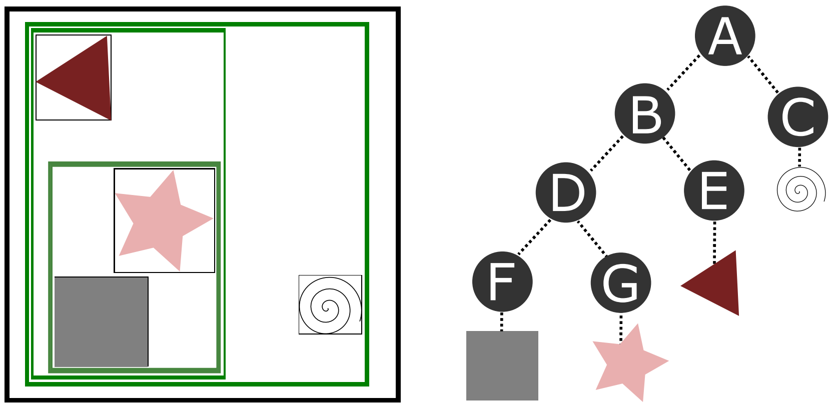


Figure 2.5: BVH built over scene primitives. Primitives encapsulated in their bounding box are stored in leaves in the final hierarchy. The internal nodes represent constructed bounding boxes visualized with the green colour in the left part of the image

There are various types of bounding volume hierarchies, with the main difference lying in the type of bounding volume used in the tree structure. The commonly used bounding volumes include sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), and discrete oriented polytope (DOP). These bounding volumes can be characterized by how tightly they encapsulate scene primitives, the efficiency of intersection tests, the memory efficiency of the BVHs, and how easily they can be updated. For example, sphere BVH is fast in terms of constructing the whole structure and computation of intersection tests. However, it often does not efficiently enclose complex objects within the bounding volume.

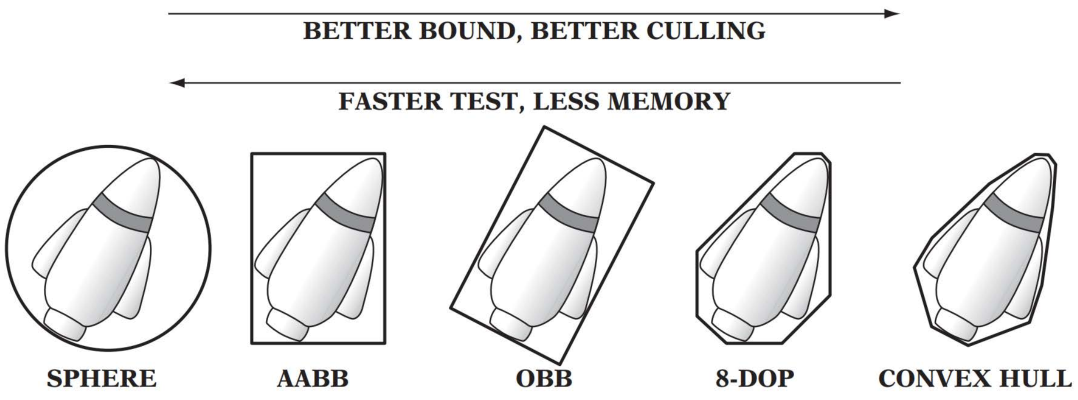


Figure 2.6: Types of bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), and convex hull. Image taken from Christer Ericson, 2004 [8].

K-dops, on the other hand, can, in many cases, tightly fit the object inside, leading to more accurate collision detections and faster ray tracing. However, constructing the whole structure takes longer, and the intersection tests of a ray and k-DOP are more complicated than the intersection test of a ray with a sphere

The choice of the specific type of bounding volume used in the hierarchy is a balance between achieving tighter enclosure of the scene objects and faster construction. This puts importance on using the correct BVH type based on what is expected from the algorithm it is used in.

2.3 Bounding volumes

Although an axis-aligned bounding box is, for its simplicity, the most used bounding volume in a bounding volume hierarchy, there are cases where using simple bounding volumes is inefficient. For example, when a lower ray tracing cost is needed and the construction time is not as critical. As already mentioned in subsection 2.2.2, there are various possibilities of which bounding volume could be used instead.

2.3.1 AABB

As the name suggests, an axis-aligned bounding box is a rectangular enclosure of the primitive that is aligned with the canonical axes of the coordinate system in which it is defined, typically the standard X, Y, and Z axes in the Cartesian system. Since the sides of the box are parallel to these axes, operations such as computing the bounding rectangle for geometry or performing intersection tests are simple and efficient [11].

AABB in bounding volume hierarchy

Using axis-aligned box as the bounding volume in a bounding volume hierarchy simplifies the construction of the hierarchy and the query tests. On the other hand, it results in less tight geometry fit compared to more fitting volumes.

Each node of the hierarchy stores only the minimum and maximum bounds of the bounding box along the parallel planes defining slabs of the bounding volume. This requires storing $3 * 2$ float numbers, which equals to 24 bytes of memory per node. This calculation does not include pointers to child nodes, parent nodes, or other hierarchy-related data since these components are consistent across all bounding volumes.

Some ray tracing kernels require the parent node to store its children's bounding boxes instead for efficient access during the ray tracing process [1]. In that case, each node consumes 48 bytes of memory.

2.3.2 OBB

An oriented bounding box is again a rectangular enclosure of the primitive as the AABB. However, its orientation is arbitrary with respect to the coordinate axes. The advantages of OBB are invariance to translation and rotations, yet the collision test is computationally more expensive than with AABBs [6].

OBB in bounding volume hierarchy

Using an oriented bounding box as a volume in a bounding volume hierarchy brings a balance between the increased complexity of the hierarchy operations and the improved tightness of the volume. The utilization of OBB optimizes both performance and accuracy in spatial queries.

Each node of the hierarchy stores the axes that determine the orientation of the rectangle along with the absolute values of the extents of its bounding box. These extents are measured along those axes from the centre point of the bounding box. Additionally, the midpoint in the standard base is stored in each node as well.

The centre point is represented by 3 floats, axes by 9 floats, and extents by 3 floats, which sums to 60 bytes of additional memory needed per node.

2.3.3 DOP

The object's most complex bounding volume is the convex hull. On the other hand, it is complicated to construct in a performance and memory utilization manner. The next option is a discrete-oriented polytope (DOP). A DOP is one of the most complex bounding volumes. It is a generalization of AABB volume.

k-DOP

k-DOP has an additional definition of the k number of hyperplanes specified by normal vectors that determine its shape. The planes are brought closer to the enclosed object until they collide [6].

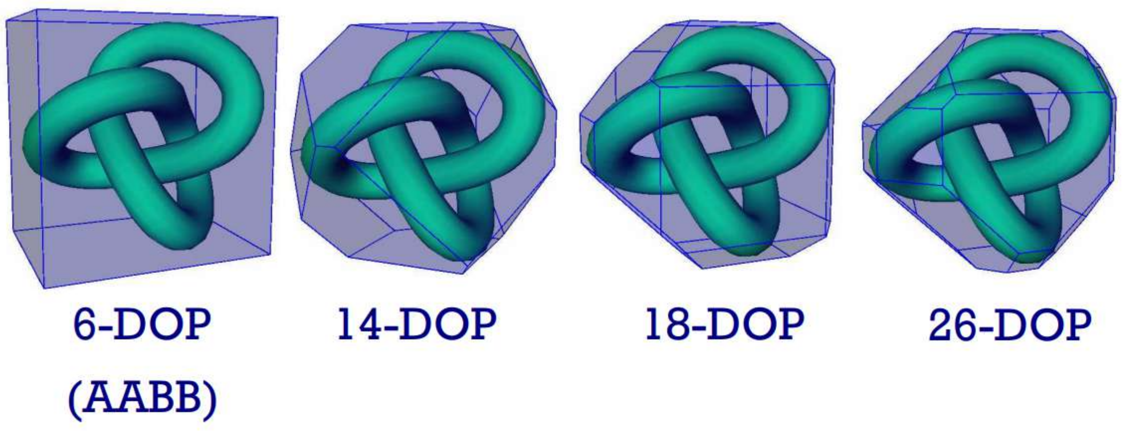


Figure 2.7: Examples of k-DOPs with different k values. Image created by Gabriel Zachmann et al., 2005 [36].

With k being 6, the resulting polytope has six faces, each having a defined normal vector. The normal vectors are: $(\pm1, 0, 0), (0, \pm1, 0), (0, 0, \pm1)$ and the polytope corresponds to the axis-aligned bounding box. Examples of different k-DOPs based on the value of k are shown in Figure 2.7 [14].

k-DOP in Bounding Volume Hierarchy

Using k-DOP as the bounding volume in BVH brings an immense benefit. The complexity of the geometric shape and the tightness of the encapsulation of the object fitted inside the volume saves time during the ray tracing process. As already mentioned in the chapter 2, fewer scene primitives stored in the leaves of the hierarchy are tested against the ray intersection, which lowers the performance cost of ray tracing.

k-DOP volumes are stored in the inner nodes of the hierarchy and share a set of normals among the whole hierarchy. This means there cannot be, for example, a combination of 6-DOPs and 14-DOPs in the BVH.

Since the set of normals is fixed, the k-DOP volume requires storing only the minimal and maximal values along the normal vectors of the volume. Each k-DOP is then defined by k/2 number of minimal and maximal values making it k*4 bytes of memory.

2.4 Cost Model

A cost model of the hierarchy was introduced to measure the quality of the built hierarchy. One of them is the surface area heuristic (SAH).

2.4.1 SAH

The construction time of the hierarchy is minimal compared to the time it takes to traverse the hierarchy and calculate the ray/object intersections. This makes it useful to build a more efficient hierarchy that takes into account the additional time saved during hierarchy traversal [18].

The probability of a ray intersecting a node of the hierarchy equals the surface area of the node divided by the surface area of the root if the rays meet [30]:

- the ray's origins and directions are uniformly distributed across the object space
- ray origins are sufficiently far from the object.
- all rays intersect the bounding volume for the entire scene

This results in intersection estimates [18]:

$$\begin{aligned}
\text{Number of interior nodes hit per ray} &= \sum_{i=1}^{N_i} \frac{\mathrm{SA}(i)}{\mathrm{SA}(\mathrm{root})} \\
\text{Number of leaves hit per ray} &= \sum_{i=1}^{N_l} \frac{\mathrm{SA}(l)}{\mathrm{SA}(\mathrm{root})} \\
\text{Number of objects tested for intersection per ray} &= \sum_{i=1}^{N_l} \frac{\mathrm{SA}(l) \cdot N(l)}{\mathrm{SA}(\mathrm{root})}
\end{aligned}$$

where the various quantities are:

- N_i = number of interior nodes
- N_l = number of leaves
- SA(i) = surface area of interior node i
- SA(l) = surface area of leaf node l
- N(l) = number of objects stored in leaf l

Given these estimates of each type of node, the cost of the whole hierarchy can be computed as a sum of these three values. The cost of the tree is then given by:

$$\text{cost of tree} = \frac{C_i \sum_{i=1}^{N_i} \mathrm{SA}(i) + C_l \sum_{i=1}^{N_l} \mathrm{SA}(l) + C_o \sum_{i=1}^{N_l} \mathrm{SA}(l) \cdot N(l)}{\mathrm{SA}(\mathrm{root})}$$

where

- C_i = cost of traversing an interior node
- C_l = cost of traversing a leaf
- C_o = cost of testing an object for intersection

These values are considered to be constants for each hierarchy.

2.5 Morton Codes

Morton Codes is a way of mapping multidimensional data onto a one-dimensional curve. It is named after G. M. Morton, who introduced the usage of the Morton Codes ordering in file sequencing described in 1966 [23].

The main idea behind the Morton codes is that data close to each other in an area should also be close to each other in the sequencing. It transforms provided data into quantized vectors and orders them creating a curve connecting the data. Vectors with subsequent Morton codes are then locally close to each other. The final curve resembles the letter Z as visualized in Figure 2.8; that's why it is also referred to as the Z-curve.

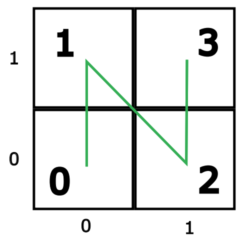


Figure 2.8: Z-curve visualized for four elements of data (frames).

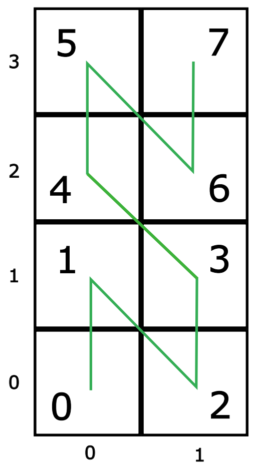


Figure 2.9: Z-curve visualized for eight elements of data (frames).

Morton dealt with area data and divided the space into square regions called frames. The Z-curve virtually connects these frames by ordering their Morton codes and creating a sequence of these frames. Visualization of the Z-curve for different numbers of frames can be seen in Figure 2.8 and Figure 2.9. It is noticeable that additional frames 4-7 added in the second image hold the same relationship as a whole as the frames 0-3.

Each frame can be represented by two binary numbers (x, y) according to its actual position, such as these rules proposed by Morton are followed:

1. Express the frame number in an even number of binary digits.
2. Starting with the leftmost bit, write down every alternate bit.
3. Starting with the second leftmost bit, write down every alternate bit again.
4. The number written in (2) is the binary representation of the x coordinate, and in (3), the binary representation of the y coordinate.

An example would be frame 14. 14 can be written as 1110 in binary form. This binary number would then be split into parts 11 and 10, while 11 represents the x coordinate and 10 represents the y coordinate. 11 is 3 in decimal form, and 10 is 2 in decimal form. This means that frame 14 would lay on position (3,2). This example can be verified in Figure 2.10 where number 14 lies, as computed, on position (3,2).

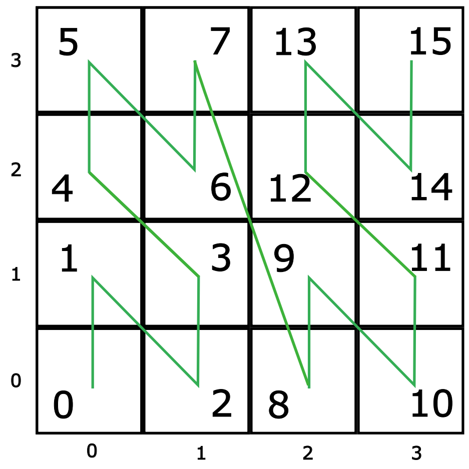


Figure 2.10: Z curve created for 16 frames

Morton codes are heavily exploited in BVH construction for efficient approximate spatial sorting of scene primitives [7, 15, 20].

2.5.1 Constructing BVH Using Morton Codes

As mentioned earlier in this section, the Morton Codes can be used for sorting scene primitives by representing them as points and encoding them. These sorted points can then be composed into clusters creating BVH recursively. There is also a critical property these clusters hold: Different clusters can be distinguished by a change of a bit in the Morton code on a particular bit position.

There are two main approaches to constructing the BVH using the Morton codes. Lauterbach et al. came up with a method called LBVH. This method sorts primitives by their centroid, finds changes in the bits, and recursively builds the hierarchy [16].

The other method, called HLBVH introduced by Garanzha et al. is similar to the LBVH method in the higher parts of the hierarchy but also uses a surface area heuristic to further subdivide small clusters by computing the surface area of the bounding volumes [10].

2.6 Agglomerative Clustering

Agglomerative clustering is a clustering of primitives based on similarity. It progresses from the bottom to the top, merging similar objects (clusters) until only one cluster remains. It starts in the leaves where only one object is stored and continues upwards, building a hierarchical structure during the process.

Locally ordered agglomerative clustering: In each step, the algorithm finds the nearest neighbour for each cluster. If both agree they are each other's nearest neighbours, the closest cluster pairs are merged, creating one larger cluster.

Chapter 3

Related Work

Many algorithms have been proposed for the construction of data structures, enhancing the performance of ray tracing. This chapter discusses several of these algorithms, focusing on the specific details of each, including data organization such as spatial partitioning, hierarchical structuring, and different strategies of the BVH construction. Understanding the different approaches to BVH construction, along with both their advantages and disadvantages can help in selecting the most suitable BVH construction algorithm to fit specific requirements. It presents the *Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction* [33] with the *Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction* [20] methods.

Additionally, this chapter mentions the *DiTO* algorithm, which effectively creates OBB bounding volume over a given mesh. This method is beneficial for forming the OBB bounding volume used in BVH, enhancing the overall summed area of the hierarchy across all nodes. This improvement brings more efficient ray traversal through the OBB structure compared to the AABB BVH.

Fast ray tracing relies on efficient ray-volume intersection tests. Specifically, ray tracing with the use of OBB BVH relies on a fast ray-OBB intersection test. Consequently, the chapter ends with the *Fast and Robust Ray/OBB Intersection Using the Lorentz Transformation* method.

3.1 Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction

Vinkler et al. proposed a new extended use of Morton Codes in a BVH construction that significantly improved the quality of the final BVH without increasing the computation time. Final BVH can almost match high-quality BVHs constructed with more time-complex algorithms [33]. In this section, the algorithm is described along with the benefits of this approach.

The algorithm believes that objects with significant differences in size should be during the process of BVH construction separated. Large bounding volumes would severely influence the bounding boxes of smaller objects. These boxes enforce large bounding volumes in the path from a leaf to the root. It causes the creation of not tightly fitting bounding volumes for small objects laying on this path after creating their common bounding volume. The solution for this problem is to separate large objects from the rest by encoding the object's size into the Morton code.

The Morton code has, after this addition, four coordinates (x, y, z, s), where x, y, z encode the spatial position of the scene primitive and s its size in a normalized form. The highest value of size representable with the given number of bits corresponds to the

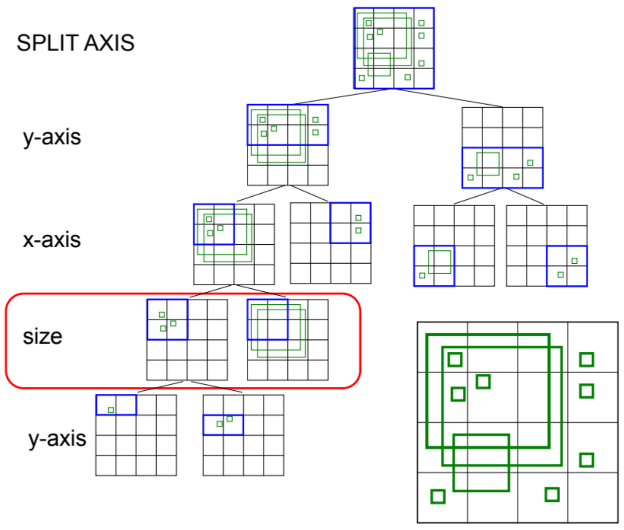


Figure 3.1: The visualization of the BVH construction using the extended Morton code with the injected object's size. On the third level of the hierarchy, the splitting axis is considered to be the size bit to subdivide the objects into two sections on smaller and larger objects. Larger objects are the ones that are larger than the diagonal of the current box (shown in blue). Illustration from Vinkler et al. 2017 [33].

diagonal of the scene's AABB. The size bit is injected into the quantized spatial coordinates to follow the sequence xyzsxyzs.

During the subdivision into subtrees, if the size is the current separation axis, objects with the size bit set to 1 are perceived as larger than the currently evaluated box and are separated into a different subtree. This subdivision is illustrated in Figure 3.1 on the third level of the hierarchy.

In scenes with a large number of triangles or for shorter bit codes, encoding the size could significantly impact the main subdivision by the spatial position of the object. Vinkler et al. introduced a method of using fewer bits for size encoding by injecting the size bits only every seventh bit. In other words, injecting it after two series of splits in each axis leads to the bit sequence being constructed as follows: xyzxyzs.

3.2 Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction

The Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction (PLOC) algorithm took a slightly different direction in the bounding volume hierarchy construction using the parallel approach to locally-ordered agglomerative clustering. This algorithm can efficiently speed up the building of the BVHs while improving the ray tracing over the structure [20]. This section summarizes the details of the algorithm.

The main idea of the algorithm is, as already mentioned in the introduction to this section, the usage of parallel locally-ordered agglomerative clustering performed on a large number of clusters. However, as mentioned in section Agglomerative Clustering, this method needs to find the nearest neighbour for each cluster in each step. Finding the nearest neighbour is a time-consuming step that requires the help of a supporting data structure, such as a Kd-tree. As Meister et al. pointed out, Kd-trees are unsuitable for parallel processing, resulting in this structure being impossible to use in the implementation.

Meister et al. came up with a solution of using Morton codes with local exploration for finding only the approximate nearest neighbour of each cluster in an ordered sequence, significantly reducing the time of finding the nearest neighbour.

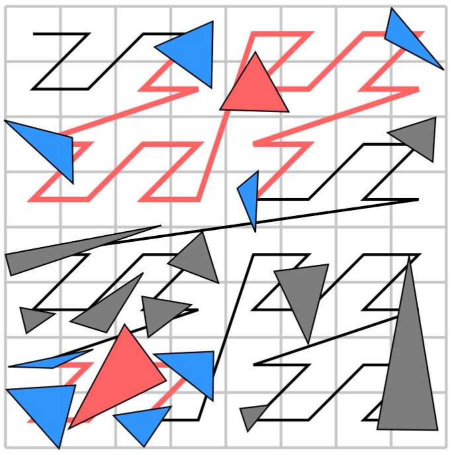


Figure 3.2: The illustration of the nearest neighbour search of clusters along the Z-curve. In the image, the red triangles are searching for their nearest neighbour. Image taken from Meister et al. 2018 [20].

Approximate nearest neighbour search: Firstly, the clusters are sorted based on the Morton codes computed for their centroids. Secondly, the algorithm finds their nearest neighbour along the Z-curve in a predefined radius. The search interval is then $< i - r, i + r >$ where r is the search radius, and i is the index of the currently evaluated cluster C_i. Each cluster except cluster C_i in the interval becomes a candidate of the nearest neighbour. The distance between the candidate cluster and the currently evaluated cluster is computed, and the candidate with the smallest distance value is marked as the nearest neighbour of cluster C_i. The process is demonstrated in Figure 3.2. Red triangles are searching for their nearest neighbours along the Z-curve, and the distance of their search is visualized with the red line.

3.2.1 Algorithm details

At first, n bounding boxes are created for each triangle, and their Morton codes are computed. The Morton codes are then sorted, which orders the triangles along the Z-curve. Initially, every bounding box is one cluster being the input into the recursive merging process of the clusters.

In each iteration, the algorithm finds the nearest neighbour in the one-dimensional input buffer for each cluster. It continues with merging the cluster pairs that have agreed on being each other's nearest neighbour. The merging process of a pair of clusters is done only by the thread processing a cluster with a lower index to avoid conflicts. The second cluster is marked as invalid. Then, a new cluster is created. The whole algorithm is demonstrated in Figure 3.3.

The final phase of the iteration is called compaction, where invalid clusters being the merging stage results, are removed after performing a parallel prefix scan. The prefix scan produces values that help determine the indexes of the active clusters that are then written into the output buffer. The global prefix scan must be done to determine the active clusters to remove gaps formed along the Z-curve after removing the invalid clusters. No additional sorting is required. The benefits of such sorting would not prevail over the high-performance cost of the operation.

The active clusters over a particular model in a specific iteration are illustrated in Figure 3.4.

At the end of each iteration, the input and output buffer are swapped, and the whole process is repeated until only one cluster is present.

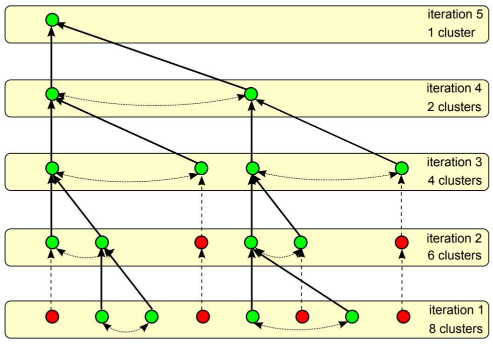


Figure 3.3: Illustration of the algorithm showing five iterations of the primary cycle that merged eight clusters into one. Green nodes connected by a dotted line are merged clusters that agreed on being each other's nearest neighbours. These merged clusters were the input to the next iteration along the clusters that did not agree with their nearest neighbour and were not merged (red nodes). The illustration comes from Meister et al. 2018 [20].

3.3 Fast Computation of Tight-Fitting Oriented Bounding Boxes: DiTO Algorithm

As already mentioned, one way how to fasten up traversing algorithms like ray tracing, collision detection, frustum culling, occlusion culling, and other queries is by using tighter fitting bounding volumes. One of them might be the oriented bounding box (OBB). This section discusses an algorithm that presents a way to compute high-quality OBB from a set of vertices in a linear time [17].

3.3.1 Algorithm overview

The algorithm operates by generating a ditetrahedron from a small, constant number of extremal vertices selected along predefined axes. The resulting OBB is derived from this ditetrahedron; more specifically, the optimal orientation of the box is chosen. The algorithm's running time depends on the number of predefined axes along which the extremal vertices are selected. Different instances of this algorithm are called DiTO-k, where k represents the number of selected vertices.

One of the twelve triangles of a ditetrahedron should most closely represent the geometry orientation of most meshes. The algorithm iterates over all triangles of the ditetrahedron

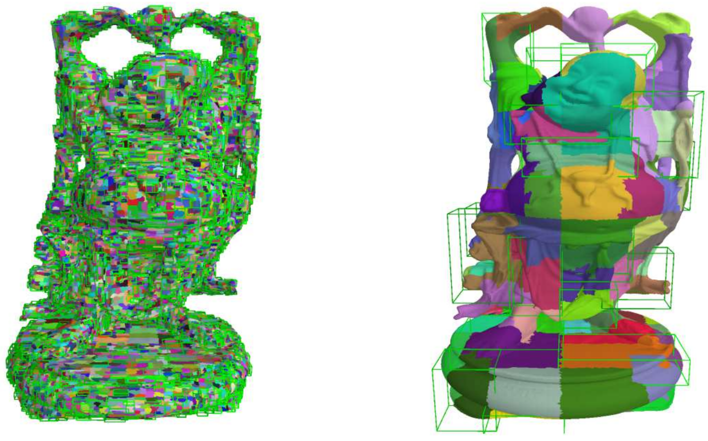


Figure 3.4: The visualization of the active clusters over the Happy Buddha model in iterations 20 in the left and 55 in the right images. The bounding boxes show clusters created in these iterations. Image taken from the PLOC article by Meister et al. 2018 [20].

and selects an orientation with the minimum-volume OBB. A comparison between an AABB and OBB is visualised in Figure 3.5.

3.3.2 Algorithm details

The algorithm finds a tight-fitting OBB defined by a set of three axes defining its orientation, a midpoint and the box's extents from a geometry mesh. It works in three main parts. At first, the algorithm selects extremal vertices from the mesh. Secondly, it generates a ditetrahedron from these extremal vertices. Finally, the algorithm iterates over all triangles of the ditetrahedron and picks one from which a local reference frame creates the smallest surface area OBB.

Extremal vertices selection is heavily dependent on a set of $s = \frac{k}{2}$ predefined axes. The algorithm finds k extremal points from a set P of geometry vertices by finding vertices with minimal and maximal projection values along the predefined normals. All these extremal vertices are guaranteed to lie on the convex hull of the geometry. Lengyel presents four various sets of normals with different k. These efficient sets are shown in Table 3.1.

This approach resembles building a k-DOP over the vertices with k predefined axes representing the k-DOP's planes.

N_6	N_{10}	N_7	N_{13}
$(0,1,a)$	$(0,a,1+a)$	$(1,0,0)$	$(1,0,0)$
$(0,1,-a)$	$(0,a,-1-a)$	$(0,1,0)$	$(0,1,0)$
$(1,a,0)$	$(a,1+a,0)$	$(0,0,1)$	$(0,0,1)$
$(-1,a,0)$	$(a,-1-a,0)$	$(1,1,1)$	$(1,1,1)$
$(a,0,1)$	$(1+a,0,a)$	$(1,1,-1)$	$(1,1,-1)$
$(a,0,-1)$	$(1+a,0,-a)$	$(1,-1,1)$	$(1,-1,1)$
	$(1,1,1)$	$(1,-1,-1)$	$(1,-1,-1)$
	$(1,1,-1)$		$(1,1,0)$
	$(1,-1,1)$		$(1,-1,0)$
	$(1,-1,-1)$		$(1,0,1)$
			$(1,0,-1)$
			$(0,1,1)$
			$(0,1,-1)$

Table 3.1: Sets of normals for k = 12, 20, 14 and 26.

Finding axes of the OBB is a second part of the DiTO algorithm. In this part, candidate axes are generated (candidate OBBs), and the best ones are kept (the OBB with the smallest area). The first step is finding the largest base triangle. This triangle is defined by two furthest extremal points p_1 and p_2 and a third point p_3 that is the furthest point from a line passing through points p_1 and p_2. Edges of this triangle are used to generate three different candidate orientations. The axes are computed as:

$$\begin{aligned} u_0 &= \frac{e_0}{\|e_0\|}, \\ u_1 &= \frac{n}{\|n\|}, \\ u_2 &= u_0 \times u_1. \end{aligned}$$

where n is a normal of the triangle and $e_0 = p_1 - p_0$. This step is repeated for the other two edges, and the axes of the resulting OBB with the smallest area are kept.

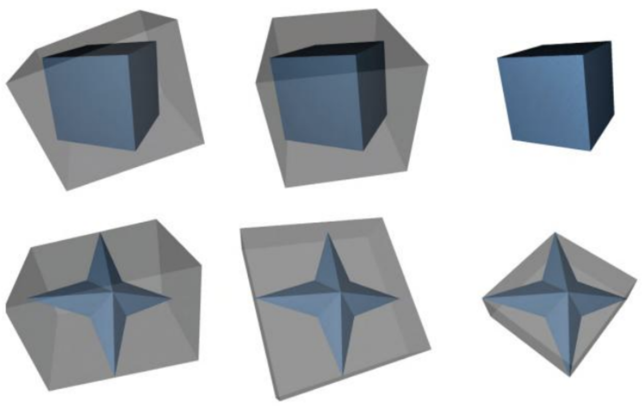


Figure 3.5: Comparison of computed boxes for a simple cube and star mesh by three different algorithms. The first column shows the AABB, the second column shows the OBB constructed by PCA methods, and the third column shows the OBB by the DiTO algorithm. The illustration taken from Lengyel, 2011 [17].

The ditetrahedron is constructed by finding the two furthest points from the large base triangle in both directions. See Figure 3.6 for an example of this process.

For every triangle of the ditetrahedron, three sets of axes are generated, as already described earlier. As before, these axes represent a candidate OBB, and the ones that represent OBB with the smallest area are retained.

The final phase of the algorithm consists of fitting every vertex into the final OBB and setting the final minimal s_u, s_v, s_w and maximal l_u, l_v, l_w projection values. It iterates over every vertex of the geometry, projects the vertex onto the OBB's axes, and updates the OBB's extents.

Finally, the algorithm determines whether the final OBB is smaller than an initially computed AABB. Otherwise, it is aligned with the AABB instead.

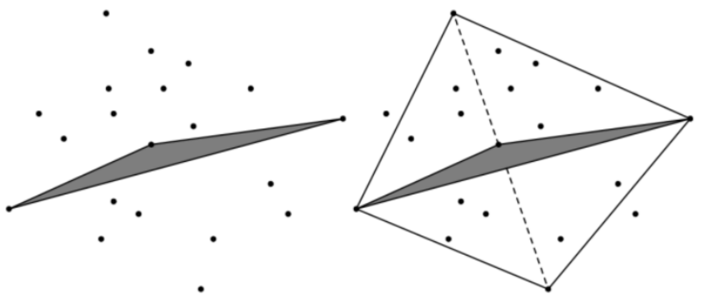


Figure 3.6: An example of a ditetrahedron created from a set of points with a large base triangle. Image taken from Lengyel, 2011 [17].

3.4 Fast and Robust Ray/OBB Intersection Using the Lorentz Transformation

Ray/AABB intersection is a relatively fast query that requires six scalar comparisons to test the ray against three slabs representing AABB sides. However, the ray/OBB intersection is a bit more difficult to query because of the box's orientation. This section presents an algorithm that speeds up the ray/OBB intersection query by transforming the ray into the OBB's local frame [29].

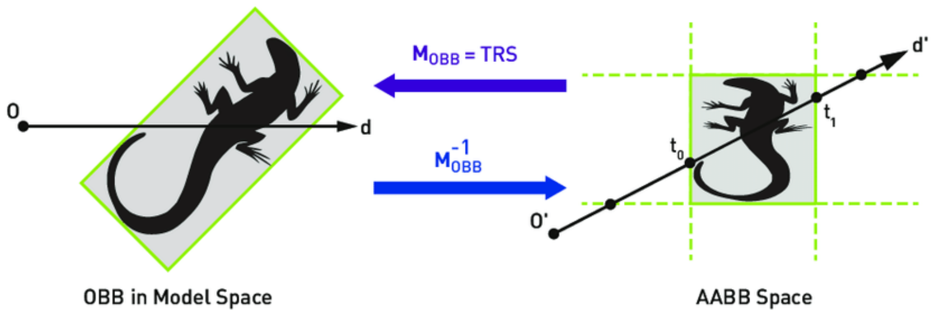


Figure 3.7: Ray being transformed from OBB model space into a unit AABB space using the $\mathbf{M}_{\text{OBB}}^{-1}$ matrix (lower blue arrow) and backwards from unit AABB space to the OBB in Model Space (upper purple arrow). Image taken from Sabino et al. 2021 [29].

3.4.1 Ray/AABB intersection

As AABB can be decomposed into six axes-aligned hyperplanes, the intersection can be written as:

$$t_{\text{min}} = \frac{P_{\text{min}} - O}{d}, \quad t_{\text{max}} = \frac{P_{\text{max}} - O}{d}.$$

where P_{min} and P_{max} are points defining the planes, more specifically, the corners of the AABB, O is the origin of the ray and d is the direction of the ray. t_{min} and t_{max} are the entry and exit points of the ray as it intersects the AABB. These values are calculated from the ray's origin along its direction.

This can be simplified as:

$$\begin{aligned} s_c &= \min(t_{\text{min}}, t_{\text{max}}), \\ s_f &= \max(t_{\text{min}}, t_{\text{max}}), \\ t_0 &= \max(\max(s_c.x, s_c.y), s_c.z), \\ t_1 &= \min(\min(s_f.x, s_f.y), s_f.z), \end{aligned}$$

The ray and the AABB intersection exists if no overlaps exist between the closest and farthest set of t's, meaning $t_0 \leq t_1$.

3.4.2 Ray/OBB intersection

As was already mentioned, the ray/OBB intersection test can become quite complex. This method provides a way of simplifying the query by transforming the ray into the local coordinate system of the OBB and then applying the standard ray/AABB intersection. This process is visualised in Figure 3.7.

To transform the ray into the local coordinate space of the OBB, it is multiplied by the $\mathbf{M}_{\text{OBB}}^{-1}$ transformation matrix. This transforms the ray into the local frame where the OBB is represented as a unit-axis aligned cube with spans ranging from [-0.5,0.5] in all axes.

The matrix can be precomputed for a mesh's vertex set P. It is derived as a matrix multiplication $\mathbf{M}_{\text{OBB}} = \mathbf{TRS}$, where $\mathbf{T}$ is the translation matrix, $\mathbf{S}$ represents the scale matrix, and the $\mathbf{R}$ is the rotation part. These matrices are extracted from the mesh by the PCA method. The rotational part is computed as the eigenvectors of the covariance matrix for P. The scaling part can be derived from the maximal and minimal coordinates of the mesh after the rotation. Finally, the translation part is put together based on the sample mean $\overline{\mathbf{P}}$ computed by the covariance matrix and the centroid $\mathbf{V}_{\text{centre}}$ of the transformed mesh, as follows:

$$\begin{aligned}
\boldsymbol{P}' &= \bigcup_{i=1}^{n} \boldsymbol{R}^T(\boldsymbol{P}_i - \overline{\boldsymbol{P}}), \\
\boldsymbol{V}'_{\text{min}} &= \min(\boldsymbol{P}'), \\
\boldsymbol{V}'_{\text{max}} &= \max(\boldsymbol{P}'), \\
\boldsymbol{V}'_{\text{center}} &= \left(\boldsymbol{V}'_{\text{min}} + \boldsymbol{V}'_{\text{max}}\right) \times 0.5, \\
S &= \text{scale}\left(\boldsymbol{V}'_{\text{max}} - \boldsymbol{V}'_{\text{min}}\right), \\
T &= \overline{\boldsymbol{P}} + \boldsymbol{R}\boldsymbol{V}'_{\text{center}}, \\
\boldsymbol{M}_{\text{OBB}} &= \boldsymbol{TRS}.
\end{aligned}$$

The matrix $\mathbf{M}_{\text{OBB}}$ transforms the unit AABB into the mesh's local OBB. This means that the inversion of this matrix $\mathbf{M}_{\text{OBB}}^{-1}$ transforms a ray in the model space into the mesh's unit AABB space. The ray is transformed using $\mathbf{M}_{\text{AABB}} = \mathbf{M}_{\text{OBB}}^{-1}\mathbf{M}_{\text{inst}}$ as followed:

$$\begin{aligned}
\mathbf{O}'' &= \mathbf{M}_{\text{AABB}}\mathbf{O}, \\
\mathbf{d}'' &= \mathbf{M}_{\text{AABB}}\mathbf{d}.
\end{aligned}$$

As the final step, the transformed ray replaces the original ray in the ray/AABB intersection, and the original OBB is replaced by unit AABB with extents [-0.5, 0.5] in all axes. To point out, the resulting t is valid in the world space.

Chapter 4

Building Bounding Volume Hierarchies using Oriented Bounding Boxes

As previously discussed, there are several advanced algorithms for the bounding volume hierarchy construction. These algorithms present state-of-the-art builders for the BVH construction. These algorithms tend to prefer the axis-aligned bounding boxes as bounding volumes in the construction of the BVH. This is done due to their simplicity and efficiency in the BVH construction and handling of ray tracing queries. Additionally, these volumes don't offer as tight fit as some other bounding volumes. However, in scenarios where tighter bounding volumes are preferred while the time of the BVH construction isn't as critical, oriented bounding boxes are used. OBBs bring a good balance between the tightness of the volume over the geometry, which reduces the number of intersection tests and increases the construction time.

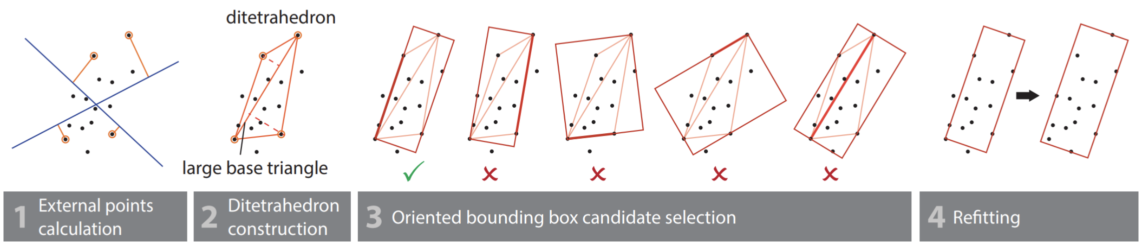
Specialised BVH builders are required for the building of OBB BVH to keep the construction time bearable as their construction is complex and computationally intensive. One of these methods uses the *DiTO* algorithm (discussed in section 3.3) to transform an existing AABB BVH into an OBB BVH. This algorithm can transform an already generated AABB BVH into an OBB BVH. This transformation involves recalculating the bounding volumes while keeping the topology of the hierarchy.

Another approach uses orthogonal sets of polytopes (ODOPs) to build the OBB BVH. This method uses ODOPs during the entire BVH construction process and, in the last step, converts the ODOP in each node into an OBB by selecting the highest quality OBB. The highest quality OBB minimizes the bounding volume's surface area and improves traversal efficiency.

These advanced methods provide flexibility in balancing the tightness of the bounding volumes and the complexity of the construction process. This chapter discusses both methods, presenting their specifics and advantages.

4.1 Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees

As was already mentioned in chapter 2, an axis-aligned bounding box is a heavily used bounding volume in bounding volume hierarchies. However, OBB provides a tighter-fitting bounding volume. DiTO algorithm, as presented in section 3.3, is an algorithm used for bounding a mesh into an oriented bounding box. This section discusses a method that, with the use of the DiTO algorithm, transforms an existing bounding volume hierarchy,



ditetrahedron
large base triangle
✓
✕
✕
✕
✕
1 External points calculation
2 Ditetrahedron construction
3 Oriented bounding box candidate selection
4 Refitting

Figure 4.1: An illustration of the process of the OBB construction with the use of the DiTO algorithm in two dimensions illustrated by Vitsas et al. 2023 [34].

switching from axis-aligned bounding boxes to oriented bounding boxes. [34].

This method works as a post-process step over an already existing binary AABB BVH. It uses a fast parallel approach to extract OBBs from an unordered set of points and proposes a fast parallel agglomerative algorithm that computes high-quality OBBs for all the hierarchy nodes.

4.1.1 Overview

This method parallelizes the DiTO algorithm for execution on a GPU. It processes all the steps of the DiTO algorithm using separate kernels, as outlined in section 3.3. An illustration of the DiTO algorithm is shown in Figure 4.1. The algorithm's first and last steps are implemented using shared memory or warp-level shuffle operations, which are suitable for reduction operations. The results are written into the global memory using atomic operations. The step of finding the ditetrahedron is a fast step that is also executed on the GPU with a small kernel launch.

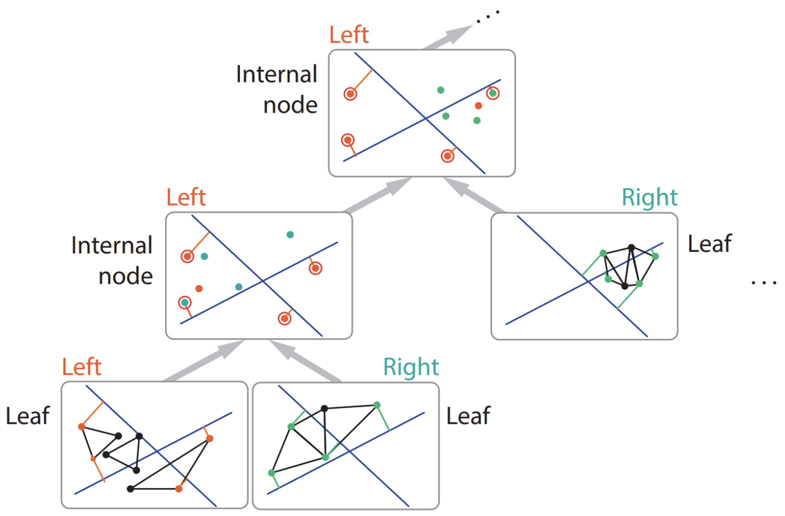


Figure 4.2: Propagation of the extremal values up the hierarchy, a first step of the method of transforming an AABB BVH into an OBB BVH. Extremal vertices out of a set of geometry vertices of each leaf are selected and propagated to the parent nodes up the hierarchy. Image taken from Vitsas et al. 2023 [34].

4.1.2 Method details

In the first step of the method, the kernel is launched for all leaf nodes. It traverses the nodes of the hierarchy bottom-up, updating the k extremal projection values for each node and propagating them up the hierarchy. Each node keeps the corresponding extremal points for these values. To minimize frequent writes to global memory, it only updates the

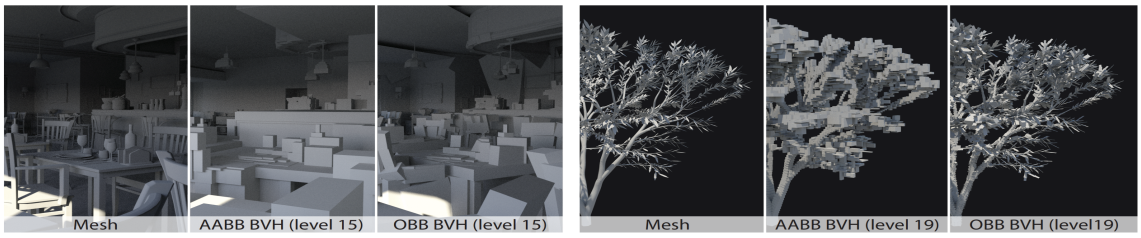


Figure 4.3: A comparison between bounding volume hierarchy using the axis-aligned bounding boxes and the one using oriented bounding boxes. It is visible that the one using OBB has a much more improved fitting of the actual geometry. The image comes from Vitsas et al. 2023 [34].

parent node's projection values when the second child reaches the parent node. The other child node's values are fetched, and the parent's projection values are updated accordingly. The visualisation of the propagation of the extremal values can be seen in Figure 4.2.

In the second step, the corresponding kernel is launched over all hierarchy nodes. It uses the pre-computed extremal values and corresponding vertices to determine the base triangle of the ditetrahedron. The method then utilizes these extremal points again to determine the other two corner points of the ditetrahedron.

The third kernel is launched over all leaves of the hierarchy. The threads traverse the hierarchy to the root and update the candidate OBB's extents. It propagates every vertex of the leaf to the parent node and updates the parent's bounding box with atomic operations. Not only does the second child that reaches the parent node propagate the values and continue, but every thread continues until it reaches the root.

The algorithm's final step is a kernel launch where the OBB is finalized for every hierarchy node. Additionally, the transformation matrix of the node is calculated, which is then passed to the ray tracer. It uses the approach presented in section 3.4.

The results of the method and the comparison between the AABB hierarchy and the newly constructed OBB hierarchy can be seen in Figure 4.3.

4.2 Building Oriented Bounding Boxes by the intermediate use of ODOPs

One approach to constructing a bounding volume hierarchy with volumes being oriented bounding boxes is to transform an existing BVH, as described in the method *Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees* in section 4.1. Another approach is to build the hierarchy using OBBs directly. This chapter presents one of these methods, specifically one that uses orthogonal sets of polytopes (ODOPs) to achieve this [28].

4.2.1 ODOP

An orthogonal discrete oriented polytope is a volume whose extents are given by the intersection of slabs. Groups of normals of the ODOP are perpendicular to each other, forming orthogonal bases. An example of a set of uniformly distributed normals specifying ODOP is visible in Figure 4.4.

Using ODOPs in this method brings the advantage of not having to process vertices during the conversion to OBBs. This fact is due to the structure serving as a storage of the unbiased topological features of the mesh.

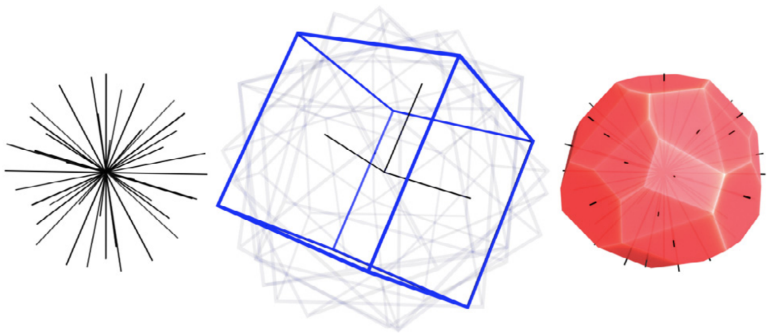


Figure 4.4: Uniformly distributed normals (left), trios of orthogonal normals representing an OBB (centre) and the intersection of all the OBBs creating the finale ODOP (right). The source of the image is Sabino et al. 2023 [28].

ODOP Construction

Bases and Normals: An ODOP is defined by N three-dimensional orthogonal bases, which means that an ODOP of N bases has $3N$ normals. The bases are generated in a way that they are uniformly distributed on the 3D sphere. As in k-DOP, each normal represents a pair of hyper-planes called slabs. The minimal ω_j and maximal Ω_j projection values along normal $q_j, j = 1, 2, 3$ define the distance these hyper-planes are placed from the origin. This is illustrated in Figure 4.5 for a slab defined by direction q_j and its minima ω_j and maxima Ω_j.

Degree of ODOPs:

The degree of an ODOP describes the number of its bases. A D-degree ODOP, D-ODOP, has $N = D^3$ bases. This means that 1-ODOP has $N = 1^3 = 1$ base representing the AABB bounding volume with the canonical base as the only base.

An ODOP with a higher number of bases generally offers a better fit of the geometry. However, on the other hand, it requires higher memory space and increased computational time to evaluate the maxima and minima along each normal. This means that the resulting OBB is a trade between tighter fit and higher building requirements.

Conversion to OBB:

An ODOP base forms a parallelepiped. This means that the volume defined by the intersection of three orthogonal slabs forms an OBB. Because the final ODOP bounds the whole geometry, it is given that each OBB formed by the trio of orthogonal normals

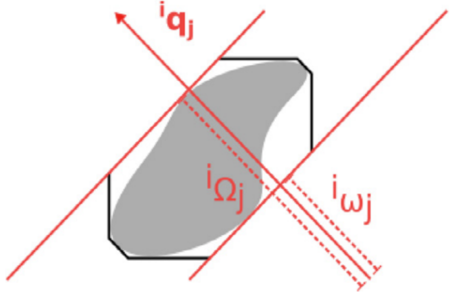


Figure 4.5: Visualisation of a slab with direction q_j and the minima ω_j and maxima Ω_j that binds the object illustrated by Sabino et al. 2023 [28].

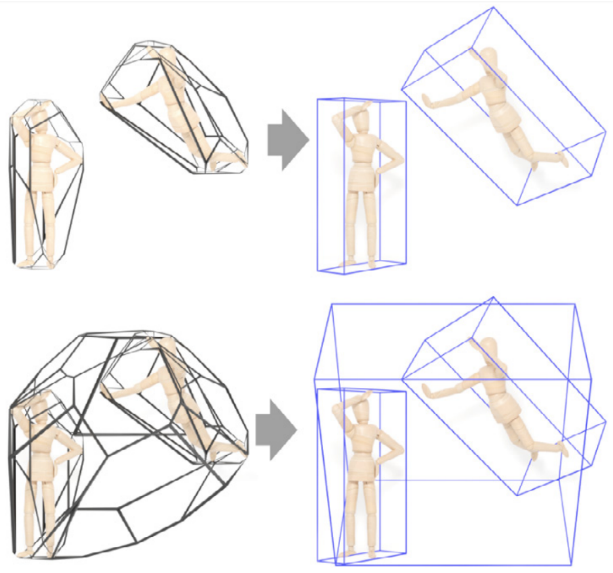


Figure 4.6: Conversion of ODOPs (black wireframe on the left) to OBBs (blue boxes on the right). Image taken from Sabino et al. 2023 [28].

bounds the whole geometry.

Converting an ODOP to an OBB consists of selecting the trio of normals whose representing OBB has the smallest area. This OBB is represented by its transformation matrix, which is then passed to the ray-tracing kernel. Figure 4.6 presents a visualisation of this process.

Constructing an ODOP from a mesh geometry involves projecting the mesh's vertices onto the ODOP's normals. These projected values then update the maximal and minimal values, defining the span of each corresponding normal. $\omega_j = \min(q_j * v_k), \quad \Omega_j = \max(q_j * v_k)$, where $v_k \in Mesh$.

Merging of ODOPs:

The simplicity of merging two ODOPs is an important feature of the ODOPs as it is heavily used during the hierarchy construction. It is done using only the minima and maxima of the two ODOPs. The finale merged ODOP C from ODOP A and B has slab intervals defined as:

$$\omega_j^C = \max(\omega_j^A, \omega_j^B), \quad \Omega_j^C = \min(\Omega_j^A, \Omega_j^B)$$

.

4.2.2 SAH Calculation

The surface area heuristic of the ODOP is calculated as the sum of the multiplicative combination of the span of the slabs of the corresponding OBB, specifically of the trio of the orthogonal normals:

$$x_j = \Omega_j - \omega_j, \quad j = 1, 2, 3$$

$$\text{SAH} = x_1 x_2 + x_1 x_3 + x_2 x_3$$

The minimal SAH value is then selected as the ODOP's SAH.

A midpoint of the ODOP is used during the algorithm for spatial ordering of the volumes. Spatial ordering orders geometric objects mapped to one point that can be used in the sorting. The point represents the ODOP's centre, specifically, the OBB inscribed to the ODOP with the smallest SAH. The midpoint p_i of the OBB i is computed as follows:

$$p_i = \frac{1}{2}\mathbf{Q}_i \begin{bmatrix} \omega_1^i + \Omega_1^i \\ \omega_2^i + \Omega_2^i \\ \omega_3^i + \Omega_3^i \end{bmatrix}$$

4.2.3 ODOP normals generator

The $3N$ normals are generated using a skew-symmetric matrix and a Cayley Transform to generate rotations without using the trigonometric functions.

The skew-symmetric matrix A is represented as:

$$A = \begin{bmatrix} 0 & z & y \\ -z & 0 & x \\ -y & -x & 0 \end{bmatrix} \quad x, y, z \in [0, 1)$$

, where x, y, z represent the rotation along each axis in a half-angle tangential notation. OBB's cubic symmetry makes only the rotation between 0° and 90° around specific axes relevant. The x, y, z elements are then generated by the 3D indexing of N and uniformly distributed over [0,1).

The matrix A is then mapped to the orthogonal matrix Q, which represents the bases of the ODOP, using the Cayley Transform. This is accomplished by solving $Q = (I - A)(I + A)^1$, where I represents an identity matrix. This mapping produces unique rotations for each parameter input.

4.2.4 Algorithm details

The algorithm modifies the PLOCTree building algorithm [32]. One version of this algorithm was talked about in section 3.2. This means that building the BVH tree uses parallel local ordering of the primitives in an array. In the original version of this algorithm, the axis-aligned bounding boxes are used. However, in this modified version, these AABBs are replaced by ODOPs and, as a final step, replaced by OBBs to get tighter fitting volumes.

In the first step, the algorithm bounds the geometry into ODOPs and sets up the clusters. These volumes are then sorted into an array by their centre point using Morton Codes.

Then, the algorithm finds the closest neighbours between these clusters along the Z curve in a certain search radius. It selects them as candidates but sets them to be the neighbour only the one where the merged ODOP of these two clusters has the smallest SAH. When both nodes agree to be each other's neighbours, they are merged and put into the tree hierarchy at the place of the neighbour with a smaller index. This process is repeated until only the root remains in the array.

The final step of the algorithm consists of going through every node in the hierarchy and converting the ODOP bounding volume into an OBB, more specifically, into an OBB with the smallest SAH out of all the candidate OBBs.

Chapter 5

Implementation

This chapter describes the implementation of the two methods for parallel build-up of OBB BVH presented in section 4.1 and section 4.2. The main goal was to unify these methods inside a common framework while using the available codes in the CUDA language. To implement these methods inside a common framework, the PLOC algorithm described in section 3.2, proved to be very beneficial as both methods can be derived from this algorithm. Consequently, the source code provided by Meister and Bittner [5] implementing the PLOC algorithm was used for this thesis and to unify the methods. In the project, Meister and Bittner use the GPU Ray Traversal framework [2] that contains the source code for the fast GPU-based ray traversal routines used in the paper *Understanding the Efficiency of Ray Traversal on GPUs* [1].

Using the PLOC algorithm and the project with the already implemented PLOC algorithm simplifies the comparison of different BVH structures. This includes the OBB BVH built using ODOPs, OBB BVH built with the DiTO algorithm executed over AABB BVH, and AABB BVH itself. These structures are built within this project which ensures the hierarchies are built in a highly similar way using the same methods and strategies. This benefits the evaluation of the presented method for OBB BVH construction.

However, using the same project for this comparison may also present a contradiction. The project was originally implemented to work optimally with the AABB BVH, thus, using the OBB BVH could result in less efficient queries.

As the main aim of the methods is to be run on GPU to speed up the construction of the BVH and the final ray tracing, the implementation is carried out using C++ and the CUDA language. The version of CUDA used for the implementation was 11.4 and 11.6.

5.1 Project Structure

The used project is originally already divided into three parts: the cub project with the cub library, the rt project, and the framework project. The framework project encapsulates essential data structures, streamlines input/output processes, and graphical user interface components and other critical elements necessary for the project functionality. The rt project capsulizes all the functional code, including classes accountable for the bounding volume hierarchy, rendering functionalities, CUDA kernels ensuring ray tracing, BVH construction methods, and application management.

As part of the `GPU Ray Tracing framework`, the application is started either in an interactive or benchmark mode. In both cases, the predefined scene is loaded with triangles, normals and texture coordinates and stored in a `m_scene` variable. The application via the `Renderer` class orders a construction of a BVH built over the loaded scene.

Based on a macro argument `ALG_TO_RUN` defined in the `Configure.hpp` file, the specific algorithm is used for the BVH construction.

```
#define DITO_ALG 0
#define ODOP_ALG 1
#define PLOC_ALG 2

#define ALG_TO_RUN 0
```

This argument can be set to numbers 0-2 which represents the specific methods for the BVH construction. That is either the method using the DiTO algorithm (DITO_ALG), the method using ODOPs (ODOP_ALG), or the PLOC algorithm (PLOC_ALG). If the DiTO method is to be run, the Renderer builds the BVH using the PLOC algorithm first and passes the already built AABB BVH to the `DiTOBuilder` object. The returned OBB BVH is passed to the ray tracing kernel to render the geometry using the constructed BVH.

In the project, the PLOC algorithm for the BVH construction was already implemented by Meister et al. using the axis-aligned bounding volumes. These classes are in the *ploc* directory.

As already mentioned, to ensure the same conditions for all methods, both AABB and OBB BVHs need to be built similarly. The method proposed by Sabino et al., which uses ODOPs to build the BVH, is constructing the BVH utilizing the parallel local ordering of the scene primitives. The other method transforms the already constructed AABB BVH into an OBB BVH. To ensure consistency between the methods, the PLOC algorithm is used to build the hierarchy over a scene, and the BVH is then transformed into the OBB BVH using the method.

Afterthe BVH is constructed, it is passed to the rendering part of the project, which is part of the `GPU Ray Tracing framework`, where the scene is rendered using the accelerating structure.

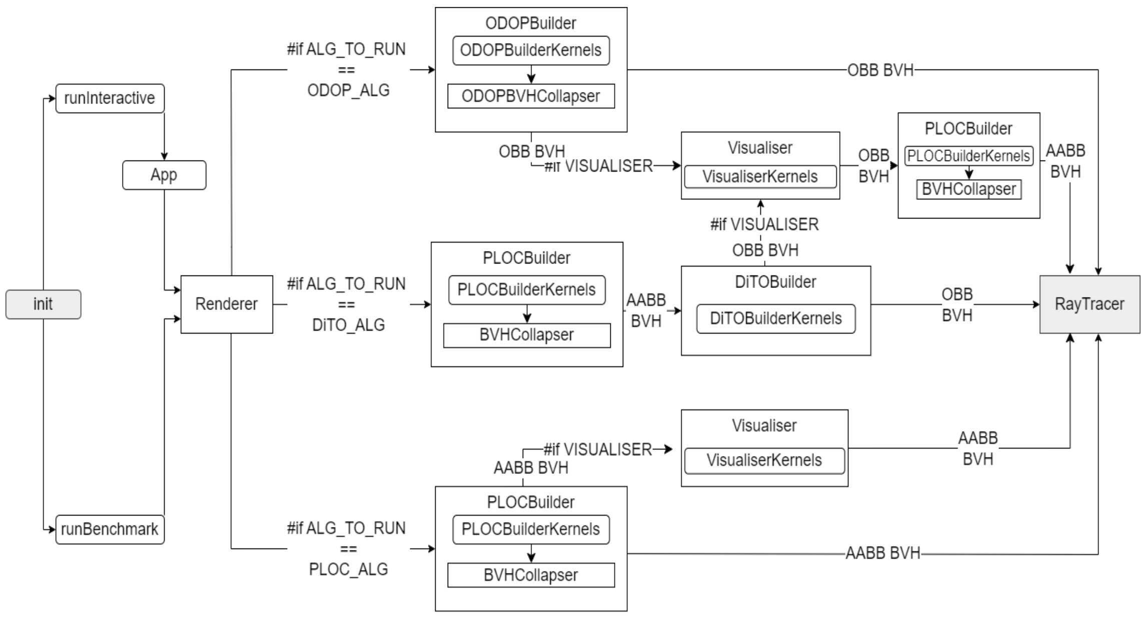


Figure 5.1: Visualisation of the possible runs of the application focusing on the BVH construction. The Renderer calls either the ODOP method or the PLOC method or if the DiTO algorithm is supposed to call, the Renderer first orders the building of the BVH to the PLOC algorithm. The constructed BVHs are either passed to the ray tracer or to the Visualiser that transforms the scene's vertices to represent the bounding volumes of the BVH in specific hierarchy level.

If the built BVH is aimed to be visualised instead of the actual geometry, the `VISUALISE` macro defined in the `Configure.hpp` file is set to 1. The `Renderer` then builds the BVH and passes the BVH to the `Visualiser` object. The `Visualiser` then transforms the BVH to contain triangles representing the bounding volumes instead of the triangles of the geometry. For AABB hierarchy, transforming the original BVH to comprise the new triangles representing the bounding volumes in nodes in the visualised hierarchy level is enough. However, with the OBB hierarchy where all bounding boxes are oriented differently, the bounding volumes of children nodes may exceed the edges of their parent's bounding volumes. This makes the visualisation of the original structure with new triangle vertices impossible without refitting the whole structure. Because of that, the PLOC algorithm is called over the created triangles, resulting in a new hierarchy structure, which is then passed to the ray tracer. The different branches of the application run are visualised in Figure 5.1.

5.2 OBB

The oriented bounding box class is declared and defined in the `OBB.hpp` file. The OBB class has three attributes as described in subsection 2.3.2:

```
class OBB
{
    float m_mn[3];
    float m_mx[3];
    Vec3f axes[3];
};
```

The `axes[3]` attribute represents the bases of the three axes of the local frame of the OBB. The `m_mn[3]` and the `m_mx[3]` attribute represent the minimum and maximum values along each axis, respectively. These extents with the axes sufficiently represent every oriented bounding box.

The `grow(const Vec3f& pt)` method extends the bounding volume by projecting the point `pt` in world coordinates onto each axis of the OBB. It compares these projections to the currently stored minimum and maximum values along each axis. The respective value is updated if the projected value exceeds the current maximum or is less than the minimum. This operation extends the bounding volume to include point `pt` if it lies outside the box.

The `area(void)` method returns the surface area of the bounding box. It is computed similarly to the area size of the axis-aligned bounding box. It computes the dimensions of the slabs of the OBB, multiplies these dimensions with each other and sums the result to calculate the area.

The `midPoint(void)` method returns the centre point of the bounding box in the local frame of the OBB. It sums the extents and divides them by two. It is likewise a similar operation as computing the midpoint of an axis-aligned bounding box.

The bounding volume node with an oriented axis-aligned bounding box is declared as a struct `CudaBVHNodeOBB` in the `CudaBVHNodeOBB.h` file:

```
struct CudaBVHNodeOBB {
    float transformMatrixLeft[3 * 4];
    float transformMatrixRight[3 * 4];
    int begin;
    int end;
    int size;
    int parent;

    FW_CUDA_FUNC CudaBVHNodeOBB(
        int begin, int end, int size, int parent) :
        begin(begin), end(end), size(size), parent(parent) {}
    FW_CUDA_FUNC CudaBVHNodeOBB(void) {}
    FW_CUDA_FUNC bool isLeaf(void);
    FW_CUDA_FUNC int getParentIndex(void)
};
```

The declaration of the CudaBVHNodeOBB struct took inspiration from the CudaBVHNode struct in the original project. The key difference is the memory size of the node. Instead of storing twelve floats representing bounding box values of the node's children, it stores twenty-four floats for transformation matrices of the children nodes. This change speeds up fetching values from memory during ray tracing. When using the speculative_while_while ray tracing kernel, intersections with the children of every node are tested one after the other during ray traversal. Testing showed that this approach results in better performance compared to storing only the transformation matrix of the specific node in the node.

The ray tracing kernel requires each node to keep track of the transformation matrices as the ray is transformed from the world coordinate system to the local coordinate system of the unit AABB cube, as described in section 3.4. This means it is sufficient to store only the transformation matrix without additional bounding box variables such as extents, the centre point, and axes.

The `begin` and `end` variables represent the indices of the first and last children in either an array of nodes or in an array of triangles. The `transformMatrixLeft` holds the transformation matrix from the world coordinate system to the local OBB's coordinate system of the left child. Accordingly, the `transformMatrixRight` represents the transformation matrix of the right child of the node.

The `CudaOBBUtil.cuh` file contains helper functions valuable for both OBB BVH builder methods. The `__device__ Mat4f getTransformationMatrix(const OBB& box)` function given an OBB computes the transformation matrix. As in the section 4.1 and section 3.4 articles, the function computes the scale matrix s, the rotation matrix r and the translation matrix t. The code for the computation of the matrices is written in Figure 5.2.

```
__device__ Mat4f getRotationMatrix(const Vec3f* axes) {
    Mat4f r;
    r.setCol(0, Vec4f(axes[0].x, axes[0].y, axes[0].z,  0.0f));
    r.setCol(1, Vec4f(axes[1].x, axes[1].y, axes[1].z,  0.0f));
    r.setCol(2, Vec4f(axes[2].x, axes[2].y, axes[2].z,  0.0f));
    r.setCol(3, Vec4f(0.0f,       0.0f,       0.0f,      1.0f));
    return r;
}

__device__ Mat4f getScaleMatrix(const Vec3f scale) {
    Mat4f s;
    s.setCol(0, Vec4f(scale.x,  0.0f,     0.0f,     0.0f));
    s.setCol(1, Vec4f(0.0f,     scale.y,  0.0f,     0.0f));
    s.setCol(2, Vec4f(0.0f,     0.0f,     scale.z,  0.0f));
    s.setCol(3, Vec4f(0.0f,     0.0f,     0.0f,     1.0f));
    return s;
}

__device__ Mat4f getTranslationMatrix(const Vec3f tsVals) {
    Mat4f t;
    t.setCol(0, Vec4f(1.0f,     0.0f,     0.0f,     0.0f));
    t.setCol(1, Vec4f(0.0f,     1.0f,     0.0f,     0.0f));
    t.setCol(2, Vec4f(0.0f,     0.0f,     1.0f,     0.0f));
    t.setCol(3, Vec4f(tsVals.x, tsVals.y, tsVals.z, 1.0f));
    return t;
}
```

Figure 5.2: CUDA device functions for matrix transformations.

The matrices are multiplied together as

```
Mat4f transform = t * (r * s);
```

The transform matrix is then inverted and returned by the function. In this way, the ray in the world coordinates is scaled first, then rotated and finally translated.

As was written in section 4.1, the components of the scale matrix are obtained as the spans of the OBB in each axis. If one of these values is smaller than a given epsilon, the epsilon is then used as the value. This inflates the side of the OBB to avoid a singular transformation matrix. The epsilon is defined as:

```
const float epsilon = 0.001f;
```

The `__device__ void atomicGrowOBB(OBB& obb, const Vec3f& pt)` function for an OBB and a scene point atomically extends the OBB's volume if the point lies outside the box. The atomicity of the operation is necessary in case multiple threads are updating the specific OBB at the same time.

5.3 Implementation of the Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees method

All files containing the implementation of this method are concentrated in the *dito* directory. There are five main files: `DiTOBuilder.cpp`, `DiTOBuilder.h`, `CudaDiTOUtil.cu`, `DiTOBuilderKernels.cu`, and the `DiTOBuilderKernels.h` file.

The `DiTOBuilder.cpp` and `DiTOBuilder.h` files include the `DiTOBuilder` class written in C++. This class manages all kernel launches and the entire transformation process. It takes the original BVH structure as an argument from the `Renderer` class in its constructor and returns the transformed BVH back to the `Renderer` class. The `DiTOBuilderKernels.cu` and `DiTOBuilderKernels.h` files contain the three kernels responsible for the parallel transformation of the BVH, as presented in section 4.1. The `CudaDiTOUtil.cuh` file contains helper CUDA functions that are called from within the kernels.

An additional helper class useful for the implementation of the method is the class `KDOP`. This class assists in the `getProjectionCoordinates` kernel. The k-DOP is constructed over each leaf's triangle's vertices and its extents are then passed to the parent's nodes up the hierarchy.

5.3.1 k-DOP

The definition of the k-DOP bounding volume with its parameters and methods used in the algorithm was added as `KDOP` class in **Util.hpp** file. The value of k (K_SIZE) and the set of normal vectors (axes) describing the bounding volume were similarly defined in the **Util.hpp** file. The set is pre-defined for k being a maximal of 24, with the first three normals representing the canonic base. As in the section 4.1 article, the set is taken from section 3.3. To match the article, the k value is set to 7, and the set is used according to the DiTO algorithm.

The parameters of the k-DOP are as defined in section 2.3.3 two static float arrays of the minimal and maximal values (m_mn[K_SIZE/2], m_mx[K_SIZE/2]).

The most important methods are: The `grow` methods expand the current k-DOP's bounding volume by adding points outside the existing volume or by merging it with another k-DOP volume. The `volume` and the *area* methods return the volume and the surface area of the axis-aligned bounding volume, respectively. The AABB encapsulates the k-DOP bounding volume given by the first three normals. The `min` and the *max* methods return the minimal and maximal values for the x, y, and z coordinates by returning the stored minimal and maximal values for the first three normal vectors.

The `midpoint` method also returns the midpoint of the axis-aligned bounding volume encapsulating the k-DOP.

definition of k and normal vectors

The `K_SIZE` constant representing the value of 2*k is defined as a macro in `Util.hpp` file. It is defined as a macro to speed up the compilation and enable the static allocation of memory when initializing arrays. The set of the axes is defined in the **Util.hpp** along the k value as follows:

```
#define K_SIZE 14
FW_CUDA_CONST int3 axes[]{
        {1,  0,  0},
        {0,  1,  0},
        {0,  0,  1},
        {1,  1,  1},
        {1,  1, -1},
        {1, -1,  1},
        {1, -1, -1},
        {1,  1,  0},
        {1,  0,  1},
        {0,  1,  1},
        {1, -1,  0},
        {1,  0, -1},
        {0,  1, -1},
        {-1, 1,  1},
};
```

5.3.2 The kernels

As was already mentioned, the cuda kernels are defined in the DiTOBuilderKernels.cu. There are multiple kernels valuable for the implementation of the method.

The `getProjectionCoordinates` kernel:

```
extern "C" __global__ void getProjectionCoordinates(
  const int       numberOfTriangles,
  const int       numberOfClusters,
  float*          nodeBoxesMin,
  float*          nodeBoxesMax,
  int*            termCounters,
  int*            nodeBoxesMinVertices,
  int*            nodeBoxesMaxVertices,
  int*            triIndices,
  CudaBVHNode *   nodes
)
```

is the first launched kernel from the `DitoBuilder::transformBVH(Scene* scene, CudaBVH& bvh, float sceneBoxArea)` method. This kernel, as the naming suggests, computes the projection coordinates for each node along a predefined set of axes.

As described in section 4.1, each thread starts at a specific leaf. For each triangle of the leaf, the thread traverses up the hierarchy until it reaches the root. During this traversal, it updates the k-DOP extents for each node. For each vertex of the triangle, it expands the node's k-DOP based on the pre-defined set of axes. Shared memory is used to store the minima, maxima, and extremal vertex indices to the array of vertices for each axis. Vertex positions are fetched from texture as needed. The process continues to the parent node, where the child increases an atomic counter in global memory for the parent.

If the thread is the first child to reach the parent, the cycle stops. Otherwise, if possible, the child fetches the other child's k-dop's extents from the shared memory and updates the parent minima and maxima accordingly.

The algorithm keeps the k-DOP box of the previously processed node and updates it with the other children's k-DOP values if needed. At the end of the process, both the global and shared memory at the parent's index are updated. This process repeats until the root is reached and processed as well.

In the article, the kernel is launched over all triangles and the leaf is selected based on the triangle's index. In our implementation, this approach is not possible as there is no reference to the leaf index from the triangle index, only the other way around. Additionally, no array of leaf nodes is available. Because of that, the kernel is launched over all nodes of the hierarchy, and continues with the algorithm only if it's a leaf node. Since each leaf node usually contains multiple triangles, the process is repeated for every triangle within the node.

The `getDitetrahedron` kernel:

```
extern "C" __global__ void getDitetrahedron(
      const int numberOfNodes,
      float* nodeBoxesMin,
      float* nodeBoxesMax,
      int* nodeBoxesMinVertices,
      int* nodeBoxesMaxVertices,
      Vec3f * nodeAxes,
      CudaBVHNode * originalNodes
)
```

, as described in section 4.1 selects an OBB candidate by generating the ditetrahedron using the minima and maxima of the k-DOP defined by the set of axes. For each minimum and maximum, it uses the corresponding extremal vertices to first create a base triangle from the most distant vertices. It continues by finding the farthest vertices in the positive and negative directions from the base triangle. This kernel utilizes helper functions defined in the `CudaDiTOUtil.cuh` file: `getBestAreaForTriangle`, `calculateDistanceToLine`, and `calculateDistanceToPlane`.

The kernel is launched independently for each node. It reads data from global memory but keeps them in the local memory. After creating the candidate OBB, it writes the axes and OBB extents back to global memory.

There are multiple cases during the ditetrahedron generation when the ditetrahedron is degenerate. The process is stopped, and the OBB is replaced by AABB. These cases are:

- when the largest distance between a vertex and a line given by two vertices would be close to zero. In that case, no base triangle could be created.
- when the largest distance between a vertex and a plane given by the base triangle is close to zero (in both directions). In that way, no ditetrahedron can be constructed.

The last possibility of replacing the OBB with AABB is at the end of the process of selecting the OBB candidate. If the quality of the OBB is lower than the original AABB, it is replaced.

The third kernel:

```
extern "C" __global__ void refitOBB(
      const int numberOfNodes,
      float* nodeBoxesMin,
      float* nodeBoxesMax,
      Vec3f* nodeAxes,
      int* triIndices
)
```

is a kernel that performs the refitting step of the DiTO algorithm. The kernel is launched over all leaves, with each thread traversing up the hierarchy. For each vertex of each triangle of the leaf, the thread updates the extents of the parent's bounding box if needed and continues upward. Unlike the `getProjectionCoordinates` kernel where certain threads stop at the parent, all threads in this kernel traverse to the root. Due to this, atomic operations are crucial to ensure correctness, as previously discussed.

The final kernel performing one of the transforming steps is the `finalizeOBB` kernel:

```
extern "C" __global__ void finalizeOBB(
      const int numberOfNodes,
      Vec3f * nodeAxes,
      float* nodeBoxesMin,
      float* nodeBoxesMax,
      CudaBVHNode * originalNodes,
      CudaBVHNodeOBB * nodes
)
```

This kernel computes the transformation matrix for each node. Therefore, the kernel is launched for every node separately. Before generating the matrix, it computes its bounding volume's surface area to see if it isn't larger than the original axis-aligned bounding box. If it proves to be larger, the bounding box is substituted by the original axis-aligned bounding box. This means its extents are updated, and the axes are set to the canonical axes, ensuring a tighter fit.

This situation may occur during the algorithm's refit part when the vertices of children's triangles extend beyond the parent's boxes. This might result in a bounding box that no longer represents the tightest possible option.

The transformation matrix is computed using the OBB's helper functions, as previously described. As a final step of the transformation, the CUDA OBB BVH node is created with the values of the original node. These contain the children and the parent indices and the size of the node. Additionally, the transformation matrix finalises the OBB BVH node.

There are other kernels that don't serve a purpose in the transforming of the BVH, however, that are still useful. For example, the `computeCost` kernel, which was inspired by the original `computeCost` kernel implemented with the PLOC algorithm by Meister et al.. This kernel computes the cost of the bounding volume hierarchy. Resuing the same SAH to compute the quality metric of the BVH, the costs will be comparable between the hierarchies and used methods.

5.4 Building Oriented Bounding Boxes by the intermediate use of ODOPs method implementation

This section talks about the implementation of the method described in section 4.2. The implementation is concentrated in the ODOP directory, where the main files are stored. As the method builds the hierarchy in the same way as the already implemented PLOC algorithm, the code was reused from the original project of the PLOC algorithm written by Meister et al.. Instead of the axis-aligned boxes, the ODOPs are used during the BVH construction and BVH collapsing. This fact enhances the comparison of both methods, as the hierarchies are constructed in the same manner.

For the method to be initiated, it requires the normals to be generated, as described in section 4.2. This is ensured by the `Generator.hpp` file with the declaration of the Generator struct:

```
template <int T> struct Generator {
    const union {
  int N_val = T * T * T;
  Mat3f bases[T * T * T];
  float normals[3 * 3 * T * T * T];
```

```
    };
}
```

The Generator, based on a specific T (given by N degree of ODOP), generates $3 * T^3$ normals used throughout the method. Because the normals are stored as individual float numbers, there are $3 * 3 * T^3$ float numbers stored in the memory. These normals are created as specified in section 4.2 by using the skewsymmetric matrix with the Cayley transform. The exact process of generating x, y, z elements isn't properly described in the paper, only as "generated by 3D indexing of N and uniformly distributed over [0,1)". Because of that, multiple approaches of generating the normal vectors were tested and visualised to determine the approach that would bring the best results. These methods were differentiating in the generation of the x, y, z elements that are used in the creation of the skew-symmetric matrix A, as presented in the paper. The visualisation of these methods is shown in Figure 5.3. It was performed by projecting the generated normal vectors as points onto a unit sphere.

Three methods brought the best distribution of the normals in the 10-degree ODOP. The first method generated the x, y, and z elements evenly on a unit cube. These elements were normalized by T to fit them in the $[0, 1)$ range. This method doesn't generate the normals uniformly on a sphere, however, in lower degrees, the normal vectors are the most evenly distributed out of these three methods.

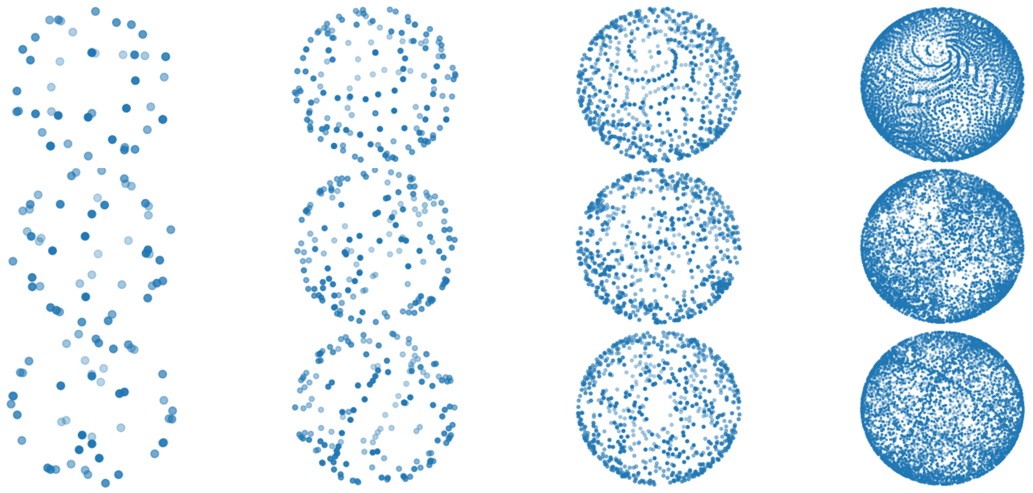


Figure 5.3: Visualisation of normal vectors as points on a sphere generated by three different methods, presented across rows. The columns correspond to 2-ODOP (24 normals), 3-ODOP (81 normals), 5-ODOP (375 normals), and 10-ODOP (3000 normals) respectively.

The second method randomly distributes the x, y, z elements within a unit cube using the uniform random distribution in range $[0, 1)$. For 10-ODOP, it generates 3000 normals. For lower degrees of ODOP, the normals are less evenly spread than in the first method.

The third method generates the x, y, z elements in a way the points are uniformly distributed on a sphere using trigonometric functions and uniform random distribution. The theta and phi angles are computed from uniformly distributed random numbers. This method generates less evenly spread normals in lower degree ODOP. As the method is used on mostly lower-degree ODOPs, the methods working better with fewer generated normals are preferred.

The first method brought overall the best results when tested on 2-ODOP and 3-ODOP on nine scenes presented in chapter 6. Generated bounding volume hierarchies had smaller cost and area in 14 measurings out of 16 where four scenes couldn't be run using 3-ODOP. This results in keeping the first method in the implementation and the final testing of the method due to its best outcomes.

After the generation of the x, y, and z elements using the presented method, these elements are passed to `Mat3f createTheSkewSymMatrix(float x, float y, float z)`, which returns the skew-symmetric matrix A. This matrix is then used for generating the normals using the Cayley Transform.

The degree of the ODOP is defined in the **Configure.hpp** file and used for the `Generator` initialization:

```
static constexpr int ODOP_DEGREE    = 2;
static constexpr int NUM_OF_NORMALS =
        3 * ODOP_DEGREE * ODOP_DEGREE * ODOP_DEGREE;
```

5.4.1 ODOP

The ODOP class is declared and defined in the `ODOP.hpp` file. Certain methods used in the kernels are, however, declared in the `ODOPBuilderKernels.cu` file for the methods to easily access the normals passed to the GPU static memory as the `odop_normals` array.

The `ODOP` class has two member values to keep it consistent with the `AABB` class:

```
float m_mn[NUM_OF_NORMALS];
float m_mx[NUM_OF_NORMALS];
```

The class's methods are very similar to those of the `OBB` class described earlier in this chapter, with a few changes.

The `__device__ void ODOP::grow(const Vec3f& pt)` method iterates through all the generated normals and projects the point pt onto each normal, similar to a k-DOP. Afterwards, it compares the projected value with the stored minimum and maximum to see if the point exceeds the bounds of the ODOP for that specific normal.

Another method is the `FW_CUDA_FUNC int getSmallestOBBAxisIndex(void) const` method. It iterates through trios of orthogonal normals representing each OBB inscribed within the ODOP. It computes the area of each OBB and returns the index of the first normal in the `odop_normals` array of the OBB with the smallest area size.

The `FW_CUDA_FUNC float area(int nodeIndex) const` computes the surface area of each inscribed OBB and returns the smallest computed area as the ODOP's area.

The `__device__ Vec3f ODOP::midPoint(void)` computes the centre point similarly as in the OBB class for the OBB with the smallest area. The centre point is then transformed into the world coordinates by multiplication with a transformation matrix. The columns of the matrix are normals corresponding to the specific OBB.

5.4.2 The kernels

As was already mentioned, the PLOC algorithm heavily inspired the implementation of the method. The module managing and kernel launching happens in the `ODOPBuilder` and the `ODOPBVHCollapser` classes, similarly as in the PLOC algorithm. The important kernels are implemented in the `ODOPBuilderKenerls.cu` and the `ODOPBVHCollapserKernels.cu` files. The ODOPs replaced the AABB volume in every step of the algorithm except for the computation of the whole scene box. In that step, the AABB is preserved to keep the uniformity of the BVH construction.

In kernels `computeMortonCodes30` and `computeMortonCodes60`, the `nodeBoxesMin` and `nodeBoxesMax` arrays of floats are filled with float minimal and maximal values in all the normals of the ODOP.

Kernels `generateNeighboursCahed` and `generateNeighbours` are modified to be able to process ODOP volumes. These kernels restore all the minimal and maximal values from the incoming arrays. These values are used to create new ODOPs merged with the currently processed ODOP. This allows the method to calculate the surface area of the entire merged box. As pointed out earlier in this section, the surface area of ODOP is obtained as the surface area of the smallest OBB inscribed within the ODOP.

The extents are directly sent to the constructor of the ODOP from the incoming `nodeBoxesMin` and `nodeBoxesMax` buffers to save the memory allocation of new arrays.

However, as the ODOP is demanding on memory, for an N-degree ODOP, each node needs to store $2 * 3 * N^3$ float values representing the minima and maxima along the normals. This implies that for each node of the hierarchy, the algorithm needs to store $24 * N^3$ bytes. Because of that, the cached version of the `generateNeighbours` kernel is impossible to use for higher ODOP degrees or bigger scenes with more triangles.

In the `merge` kernel, the clusters are merged by processing all N^3 pairs of minimal and maximal values. For each pair, the minimal value from the two and the maximal value of the two values representing two neighbour ODOPs are chosen.

5.5 BVH collapsing and converting

During the collapsing and BVH converting part of the PLOC algorithm, called methods had to be modified to work with the ODOP volume. This section presents the changes mentioned to the original PLOC BVH Collapser algorithm by Meister et al.

In the **BVHCollapserKernels.cu** file, the `computeNodeStatesAdaptive`, `compact`, and the `convert` kernels were modified to work with ODOP bounding volumes instead of AABBs.

Both the `compact` and the `convert` kernels construct the final nodes of the hierarchy. In the OBB BVH case, it means creating objects of the `CudaBVHNodeOBB` class. These kernels are in charge of converting the ODOPs into OBBs, including the generation of the transformation matrices for both child nodes.

For each thread, the process starts by selecting the $2 * N^3$ extents of the ODOP and extracting the specific ODOP from it. Subsequently, the thread determines the minimum area OBB inscribed within the ODOP by calling the object's specific methods. After obtaining the tightest fitting OBB, the thread generates the transformation matrices and assigns them to the newly created BVH nodes. This is achieved by calling the `void setTranfromMatrix(const OBB& box, CudaBVHNodeOBB* node)` device function in the `CudaOBBUtil.cuh` file. As both implemented methods use the same Cuda OBB BVH node structure, this method also utilizes 112 bytes of memory per node.

5.6 Ray Tracing

For ray tracing, the kernel `tesla_persistent_speculative_while_while` that is part of the `GPU Ray Traversal` project was utilised. This kernel needed to be adjusted to handle oriented bounding boxes instead of exclusively the axis-aligned ones. This involves fetching the transformation matrix for each node and using it to transform the ray. After that, the classic ray-AABB intersection test can be performed.

By calling the `tex1Dfetch(t_ ## NAME, IDX)` function, which returns four floats from a texture memory (the buffer of values), the node-specific values for each node in the ray tracing kernel are obtained. To simplify this fetching of quadruples from the buffer, the transformation matrix is stored in a 1D array in a row-major array to ease up the ray transformation multiplications in the ray tracing kernel.

The `TRACE_FUNC` function in the `tesla_persistent_speculative_while_while.cu` file was modified to perform this operation. The values for each node are stored and obtained as described earlier as quadruplets based on the node's address index. `tex1Dfetch(t_ ## NAME, IDX)` function is used to gather the data. The node's address index only gives information about the node's position in the sequence of nodes. The actual position in the buffer is calculated by multiplying the address index by a jumpConst constant, which has a value of 7. This is because each node occupies $7 * 4$ floats in the memory.

In each kernel cycle that performs the ray-AABB/OBB intersection tests at least seven data fetches from texture memory are required. The first fetch retrieves the indices of the children of the processing node. The next six fetches retrieve the transformation matrices for each child, with three fetches per child. An example of the first column of the matrix being fetched is shown below.

```
float4 firstRow =
    FETCH_TEXTURE(nodesA, nodeAddr * jumpConst, float4);
float c0_origx = (firstRow.x * origx + firstRow.y * origy +
        firstRow.z * origz + firstRow.w);
float c0_dirx = (firstRow.x * dirx + firstRow.y * diry +
        firstRow.z * dirz);
```

The x value of both the ray's origin and direction after transformation is calculated immediately. NodesA is a buffer in texture memory that stores the hierarchy nodes. It contains both transformation matrices for both children, their indices, the size of the node, and the parent index in this order. The variable nodeAddr represents the index of the currently processed node in the nodes array. The float4 parameters suggest a float4 structure is being fetched from the memory. In this case, it represents the row of the first (3x4) matrix.

This is repeated for the other two rows of the matrix and for the matrix of the second child. For the second child, the offset added to the jumpConst variable which points to the correct position in the memory for the second child, is 3 (3x4 float values).

The code of the ray-unit AABB intersection computation is written in Figure 5.4.

```
float3 tmin = { (-0.5f - c0_origx) / c0_dirx,
    (0.5f - c0_origy) / c0_diry, (-0.5f - c0_origz) / c0_dirz };
float3 tmax = { (0.5f - c0_origx) / c0_dirx,
    (0.5f - c0_origy) / c0_diry, (0.5f - c0_origz) / c0_dirz };

float c0min = max4(fminf(tmin.x, tmax.x), fminf(tmin.y, tmax.y),
    fminf(tmin.z, tmax.z), aux->tmin);
float c0max = min4(fmaxf(tmin.x, tmax.x), fmaxf(tmin.y, tmax.y),
    fmaxf(tmin.z, tmax.z), hitT);
```

Figure 5.4: The ray/unit AABB is computed as shown in this C++ code.

5.7 BVH Visualiser

As was already mentioned, the Visualiser class, with some other classes and methods, serves to visualise the bounding volume hierarchy. The triangles of the geometry are swapped for triangles of the bounding volumes for AABB BVH. For the OBB BVH, the scene's triangles are substituted by triangles representing the bounding volumes of the nodes. The Visualiser also computes normal vectors of these triangles and their shading.

In the Configure.hpp file, the VISUALISE macro sets whether the BVH or the geometry will be rendered. Another macro, MAX_LEVEL, decides at which level the BVH will be visualised.

The Visualiser.cpp with the header file contains the Visualiser class definition. This class is responsible for the whole BVH converting for the Renderer to be able to render the hierarchy instead of the geometry. It is responsible for managing the memory and launching the CUDA kernels.

There are two main kernels in the `VisualiserKernels.cu` file converting the BVH in parallel. Every thread in the **convert** kernel starts at the root and traverses down the hierarchy. The threads are divided to the left and the right children based on their current index. When the thread reaches the node in the `MAX_LEVEL`, it stops the search.

For the last processed node, i.e. the node in the MAX_LEVEL level, the corners of the OBB are derived and transformed into the world coordinates such as:

$$\mathbf{c_{i,world}} = Q\mathbf{c}_i$$

Columns of the matrix Q are axes vectors of the bounding box, $Q = \begin{bmatrix}\mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2\end{bmatrix}$, and $\mathbf{c}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$ represents the coordinates of corner i.

From these corners, twelve triangles are created. These triangles represent the bounding box, replacing the triangles of the geometry in the scene object and storing the triangles. Normal vectors are calculated for these triangles, as well as their shading. The processed node is set to be the leaf node, and their left and right child addresses are set to be the first index and the last index of the sequence of these newly created triangles.

Chapter 6

Results and Discussion

This chapter presents the results of the thesis and its implementation. The primary goal was to compare the methods discussed in chapter 4. The specific implementation of the method's functions and launched kernels is presented in chapter 5. The results are organized into several sections. Firstly, as the methods have to be tested on specific scenes, these scenes are presented at the beginning of this chapter. The evaluation includes the specific setup of the measuring environment and the parameters of the BVH builders. It explains the measuring process using the ray tracing technique and how rays are generated. Finally, it presents the performance metrics that indicate the specific qualities of the compared BVHs.

The OBB BVH construction methods were integrated into a project with the PLOC algorithm implemented by Meister and Bittner [5], which serves as a builder for an axis-aligned bounding box (AABB) hierarchy. This constructed AABB BVH is used as a baseline for comparison in the measuring.

The method proposed by Vitsas et al. (2023) utilises the DiTO algorithm to transform an already constructed AABB BVH into an oriented bounding box (OBB) BVH. The PLOC algorithm was employed to generate the initial AABB BVH, which was subsequently transformed into an OBB BVH. These methods are unified within a common framework, using consistent building strategies to ensure that the BVH structures are constructed in a similar manner. This allows for accurate and meaningful comparisons of the measured values. An example of an AABB BVH and an OBB BVH constructed over the Conference scene is visualised in Figure 6.1.

All of the measurings were done on a workstation with the following hardware parameters: Intel(R) Core(TM) i9-10900x CPU @ 3.70GHz, 10 Core(s); RAM 32 GB; GPU NVIDIA GeForce GTX 1070 Ti.

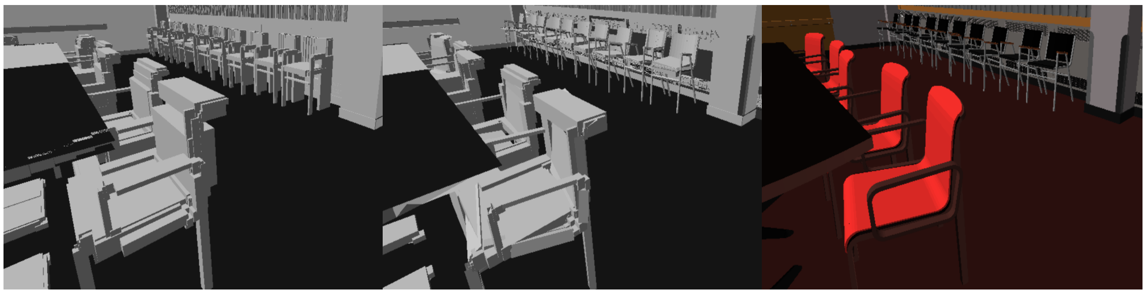


Figure 6.1: An example of an AABB BVH (on the left) and an OBB BVH (in the middle) built over the geometry of the Conference scene (on the right). The difference between the tightness of the volumes is most noticeable on chairs in the background. The OBB BVH nearly represents the actual geometry.

6.1 Rendered scenes

To test the presented methods, the hierarchies need to be built over 3D scenes. For this purpose, a collection of eight different scenes with various complexities were selected. These scenes were flattened into a single mesh scene if needed. Four of these scenes representing a single object were rendered from outside the object's borders. The rest, representing a bigger object covering a larger area, was traced from within the scene. The sizes of the scenes range from 300k triangles to 12.7M.

Tree [19]

The tree scene shown in Figure 6.2 has over 313k triangles, which makes it one of the smallest scenes in the collection alongside the Conference scene. The many small oriented objects representing the leaves and roots should make this scene beneficial for OBB volumes, creating tighter fitting volumes.

Figure 6.2: The Tree.

Dragon [3]

Figure 6.3: The Dragon.

With over 800k triangles, the Dragon scene is one of the smaller scenes. However, according to Vitsas et al. [34], BVH using OBBs should outperform BVH using AABBs by 1.35x using the LBVH builder. The Dragon scene is visible in Figure 6.3.

Hairball [19]

This scene represents a mass of thin strands and was originally created for *Two Methods for Fast Ray-Cast Ambient Occlusion [13]*. It is a single-object scene with over 2.8M triangles. With many differently oriented triangles, this scene performs well with OBBs. According to Vitsas et al. [34], the OBB BVH outperforms the AABB hierarchy by 1.5x using the LBVH builder. The Hairball scene is shown in Figure 6.4.

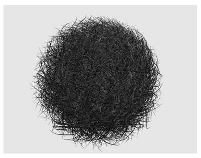


Figure 6.4: The Hairball.

Crown [3]

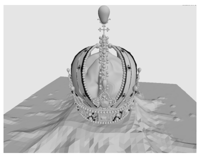


Figure 6.5: The Crown.

The Crown scene with over 3.5M triangles indicates the preferential use of OBB volumes with different orientations at specific levels of the hierarchy. According to Sabino et al. [28], tracing primary and secondary rays using an OBB BVH shows around 1.3x improvement over an AABB BVH. The scene is visible in Figure 6.5.

Conference [12]

This scene, shown in Figure 6.6, has over 331k triangles, making it a smaller scene in the collection. It is heavily used within the computer graphics community for benchmarking. Its regular axis-aligned nature presents challenges for oriented bounding boxes.

Figure 6.6: The Conference.

Rungholt [19]

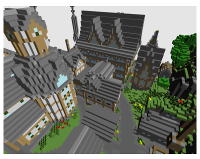


Figure 6.7: The Rungholt.

This scene is a scene transformed from a *Minecraft* map created by kescha [22]. Consisting solely of axis-aligned objects and primitives, the scene has more than 6.7M triangles. As it is composed of boxes, it favours the AABB BVH over the OBB BVH. This was demonstrated by Sabino et al. [28], where the trace time was slower for BVHs built using any degree of ODOP compared to the original AABB BVH. The scene is visualised in Figure 6.7.

Rotated Rungholt scene [19]

The Rotated Rungholt scene, as the name suggests, represents a Rungholt scene that was rotated along the x, y, and z axes. The scene was rotated around each axis by 45 degrees. As the original Rungholt scene has all the scene primitives strictly axis aligned, favouring the AABB BVH, the rotated scene should, on the other hand, favour the OBB BVH and result in the largest possible AABB bounding volumes.

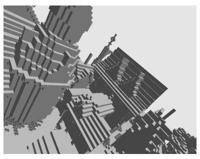


Figure 6.8: The Rotated Rungholt.

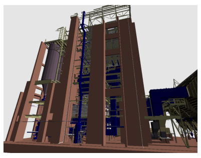


Figure 6.9: The Powerplant.

Powerplant [19]

The powerplant scene is the largest in the collection, with 12.7M triangles. Its thin, long geometry groups, which can be seen in Figure 6.9, perform well with the OBB hierarchy. Vitsas et al. [34] reports up to 1.47x faster rendering of primary rays for the OBB BVH compared to the AABB BVH.

6.2 Evaluation

The evaluation of the method proposed by Vitsas et al. (2023) [34] and the method developed by Sabino et al. (2021) *Building Oriented Bounding Boxes by the intermediate use of ODOPs* was done using the proposed implementation using C++ and CUDA 11.4 language. The methods were tested on eight scenes listed above. The resulting bounding volume hierarchies were furthermore compared with the AABB BVH constructed using the PLOC algorithm [20] to serve as a baseline. The results are presented in this section.

Although the methods can be run on dynamic scenes, specifically the method proposed by Vitsas et al. using the DiTO algorithm, the tested scenes are static and consist solely of triangles. And While the methods can create a BVH for other types of scene primitives, triangles were chosen due to their simplicity. It's also important to note that the implemented PLOC algorithm was specifically designed to work with triangles in static scenes. This would make it challenging to compare methods if different scene primitives or dynamic scenes were used.

A comparison of the presented methods is illustrated in Figure 6.10. It shows different visualised BVH hierarchies in hierarchy level 30 built over the Hairball scene. The tightness of the OBB BVH is significant for non-arbitrary oriented scene primitives.

6.2.1 Settings

For BVH construction, the c_T and c_I constants had to be set, according to as was discussed in subsection 2.4.1. These constants influence the BVH collapsing as well as the final computation of the BVH cost. For the PLOC algorithm, these constants are set to $c_T = 3$ and $c_I = 2$. For OBB BVH, these constants are set to $c_T = 4$ and $c_I = 2$ to mimic the increase in performance difficulty of the ray-OBB intersection test.

The integrated `GPU Ray Traversal` framework benchmark test was used for measuring. The PLOC algorithm was configured to run 30-bit Morton codes, the radius of the neighbour's search was set to 100, and the adaptive leaves algorithm was enabled with the

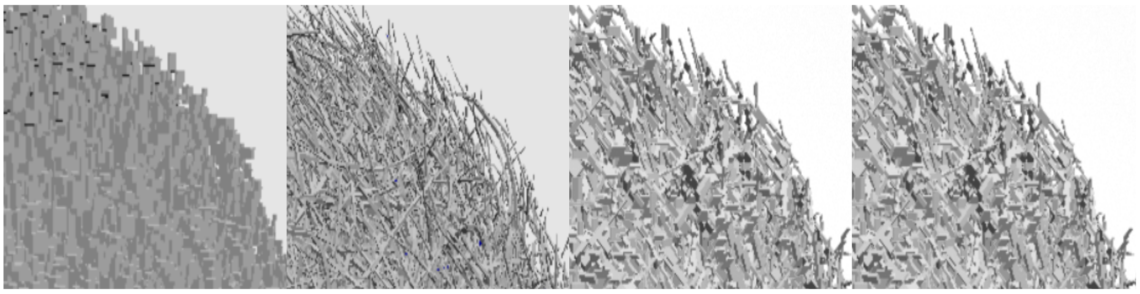


Figure 6.10: Comparison of constructed BVHs (AABB, DiTO OBB, 2-ODOP OBB, and 3-ODOP OBB in this order) over the *Hairball* scene visualised in hierarchy level 30. The improved fitting of the OBBs should lower the number of intersection tests per ray during ray traversal process. The DiTO method shows the best fitting of the scene primitives in hierarchy levels closer to leaves than the ODOP.

maximal number of triangles in the leaves set to 8. These settings were also used for the ODOP algorithm.

The whole measuring process was performed at least five times and the best results were kept. These results are summarised in Table 6.1. The results are separated for the different BVH constructed by the earlier presented methods and for each scene.

The first row in each metric section of the 6.1 Table shows the baseline AABB BVH generated by the PLOC algorithm. The second row represents the method developed by Vitsas et al. (2023) [34] for transforming the AABB BVH into an OBB BVH using the DiTO algorithm (DiTO). The next rows represent the method proposed by Sabino et al. [28], which constructs OBB BVH using a different degree of ODOP: 2, 3, and 4 (2-ODOP, 3-ODOP, 4-ODOP respectively). For larger scenes, the 3-ODOP and the 4-ODOP require allocating buffers of 244*(2*N - 1) float numbers for 3-ODOP or 576*(2*N - 1) float numbers for 4-ODOP to represent the minimal and maximal values along the generated normals, where N is the number of triangles in the scene. This memory requirement is larger than what the GPU allows. This results in an error and impossibility to test the larger scenes for the 3-ODOP or 4-ODOP, leaving the specific cells in the 6.1 table empty.

The k in the DiTO algorithm is set to represent 7 normals. These normals are specified in chapter 5 and in section 3.3.

6.2.2 Ray generation

The ray tracer renders images of the frame size of $1024x768$ pixels. It builds the hierarchy while measuring the kernel times. Subsequently, it shoots primary, AO and diffuse rays into the scene, measuring the throughput of the rays. As part of the `GPU Ray Traversal` framework, which includes the used benchmark, the primary rays generate 32 secondary rays, both AO or diffuse. These rays are distributed according to a cosine-weighted Halton sequence on a hemisphere. If primary rays miss the geometry, they generate dummy secondary rays that the ray traversal kernel ignores and excludes from the rays-per-second calculations. AO rays have a limited length and stop immediately after an intersection. In contrast, diffuse inter-reflection rays are long and traverse the hierarchy until they find the closest intersection. An example of an image rendered using specific ray types is shown in Figure 6.11.

6.2.3 Ray batches

The benchmark shoots 4 sets of ray batches into a scene as a warm-up to the graphics card to present it to a stable state. Afterwards, it runs 10 measures to measure the average time of the ray tracing process over five different positions of the camera in each scene.

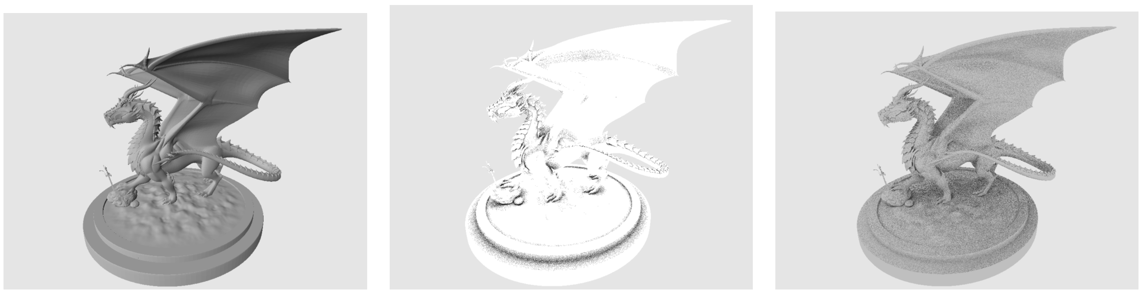


Figure 6.11: A rendered image of the Dragon scene using primary rays (on the left), AO rays (in the middle), and diffuse rays (on the right).

As already mentioned, the majority of the camera placement is for single-object scenes outside the bounds of the object. For scenes representing a larger set of objects, the camera is placed inside the scene.

All primary rays are traced in one batch. The secondary rays are divided into batches of 2^{20} rays, each traced in a separate launch. This is visible in Figure 6.12, where the separate batches are rendered once a time. Because of that, during the ray tracing process, pixels are rendered in each batch, showing part of the screen with the already rendered image generated by tracing primary rays along with part of the screen rendered with the secondary rays. The screen is updated with each batch until the process finishes, and the whole image is rendered with the secondary ray values.

Primary rays are generated in 2D Morton order in screen space. Each batch of secondary rays is sorted using a 6D Morton order based on ray origin and direction vectors. This sorting, performed by the CPU, is computationally intensive, taking about one second per batch. Due to this, the sorting is disabled in interactive mode.

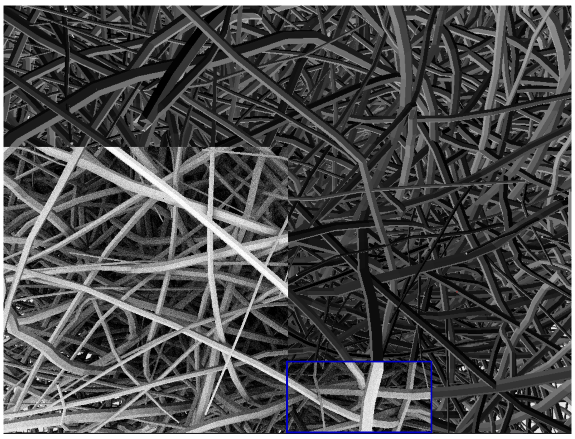


Figure 6.12: Visualisation of the middle of a process of ray tracing of AO rays traced in a hairball scene in batches. Part of the image rendered using one batch of AO rays is highlighted with a blue rectangle. The process of tracing AO rays is done in stages. The rays are traced in batches and their associated pixels are filled with values gathered from these rays, likewise in batches. Most of the AO rays were already traced on the left part of the image. The rest of the rays would follow.

For the ray tracing, the modified `tesla_persistent_speculative_while_while` kernel was used. The original kernel is part of the GPU Ray Traversal framework. It was modified to run on OBBs, as well as AABBs. This kernel was specifically chosen as it's the same kernel type used in the method proposed by Vitsas et al. 2023 [34]. This kernel produces the best performance results out of the included kernels.

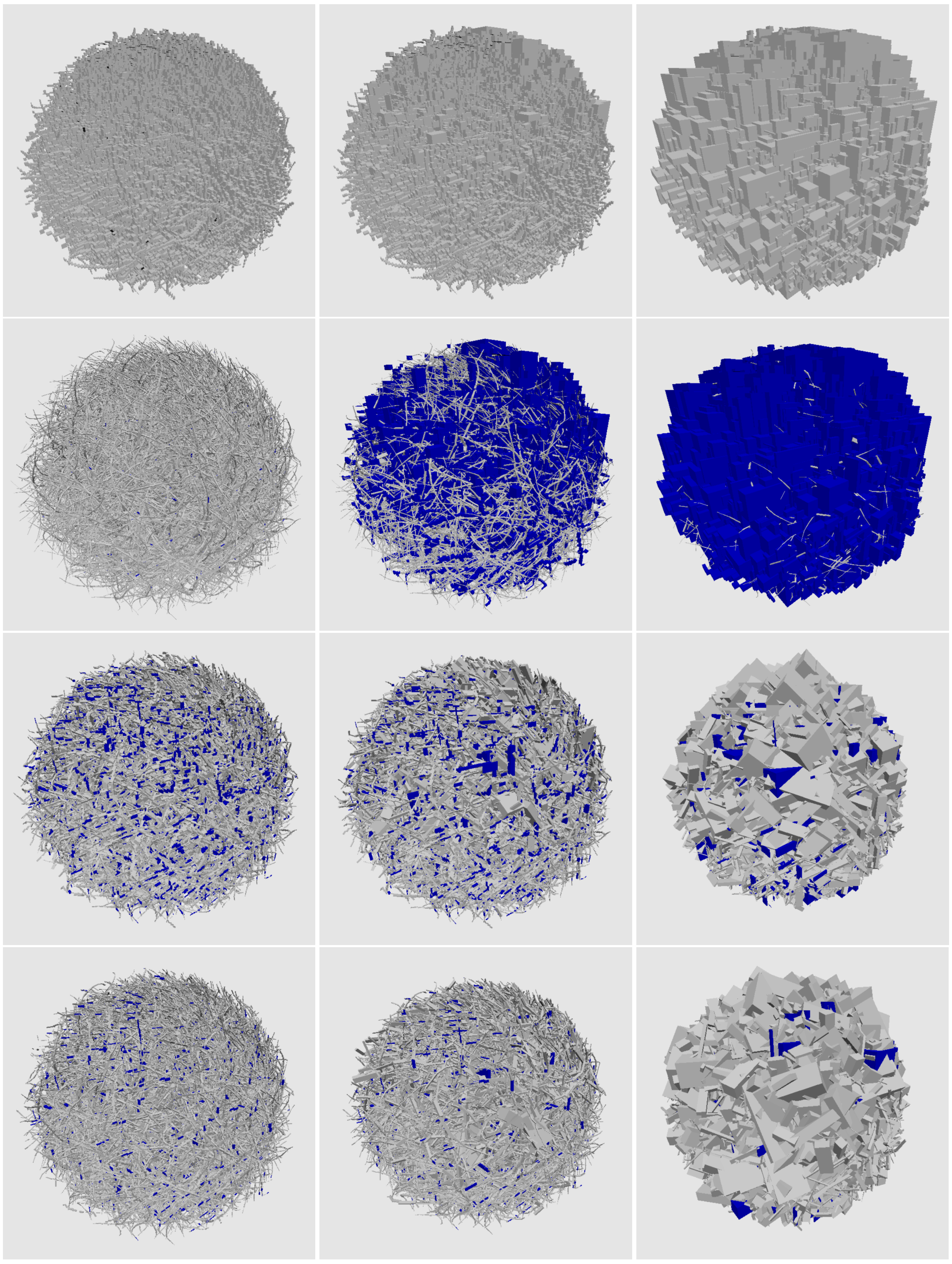


Figure 6.13: BVH hierarchy visualised for BVHs constructed via different methods over the Hairball scene. Each row represents a different method (AABB PLOC builder (1st row), OBB DiTO method (2nd row), 2-ODOP method (3rd row), and the 3-ODOP method (4th row)) and each column a different hierarchy level (level 30 (out of 33 for the AABB BVH), level 18, and level 15). For the OBB hierarchy, all nodes with the axis-aligned bounding boxes are blue-colored. AABB BVH is left gray for visibility. Noticeable fast increase of AABB nodes for the DiTO method is visible in higher hierarchy levels. The DiTO method shows the best fitting of the scene primitives in hierarchy levels closer to leaves than the ODOP, while the ODOP stays consistent over all hierarchy levels.

6.2.4 Measured performance metrics

For the measuring, special metrics were selected to best represent the quality of the BVH. These metrics are presented below. All measuring is summarised in Table 6.1.

Tree quality is an important parameter of the hierarchy that reflects how well the hierarchy is constructed and how effectively it optimizes ray tracing. This is determined by the number of intersection tests needed to be performed during the traversal of the ray in the hierarchy. BVH quality is represented by the BVH cost computed according to the surface area heuristic (SAH) described in subsection 2.4.1. In the SAH computation, the c_T, c_I constants are set to $c_T = 3$, $c_I = 2$ for AABB and $c_T = 4$, $c_I = 2$ for OBB BVH.

Another indicator of tree quality is the summed surface area of all BVH node volumes. This is shown in the **Increase of BVH area** row. The summed area of the AABB BVH is 100%, and the areas of the subsequent BVHs are shown as percentages of the original AABB BVH summed-area showing the overall decrease of the surface area of the BVH.

The next metrics are the summed area of all leaves and the summed area of all inner nodes. These values are normalized by division by the AABB scene box surface area.

Other observed metrics are the average number of triangles per leaf node and the percentage of nodes where the axis-aligned box turned out to be the most fitting one out of all nodes of the BVH. Furthermore, it is the increase of the memory requirements for the construction of each BVH compared to the original PLOC algorithm. The value shown for the DiTO method consists of the memory requirements of the transformation summed with the construction of the AABB BVH.

The totalled time for each GPU kernel during the BVH construction/transformation in seconds is tracked in the **Built time** row. For the DiTO algorithm, the sum of the AABB construction with the following BVH transformation is shown.

Finally, it's the ray throughput for each primary, AO and diffuse rays shown as MRays per second. These values measure how many ray queries the BVH can handle per second. It is determined by dividing the total number of intersection test queries by the time taken to run them.

6.2.5 Tree quality

As visible in Table 6.1, in most of the scenes, the BVH cost increased for all the OBB BVHs compared to the AABB BVH. The biggest increase is recorded for the Rungholt scene. This is expected due to the Rungholt's axis-aligned nature. The decrease in the BVH cost for the DiTO method is visible only for the Rotated Rungholt scene. This is similarly expected as this scene majorly favours the OBB-bounding volumes. It doesn't show the same phenomenon for the ODOP method, which is given by the generated normals' orientation. The orientation assumably doesn't align with the (1, 1, 1) axis.

The decrease in the BVH cost for higher ODOP degrees is noticeable, where, in most cases, it gives better results than the DiTO method. This fact is given not only by the orientation of the normals but also by using a BVH collapser modified for OBB BVH. For most of the scenes, the c_T, c_I constant difference for AABBs and OBBs is responsible for the increase of the BVH cost even if the surface area of the nodes generally decreased.

A decrease in the summed area of all nodes of the BVH is evident in the row **Increase of BVH area (%)**. In every scene, the total surface area of all BVH nodes is significantly smaller than that of the original AABB BVH. This is an expected behaviour, as the area of a node cannot be larger, by the nature of the builders, than the area of an axis-aligned node. The most significant decrease in the BVH area is observed for the Rotated Rungholt scene, where the decrease is nearly by 30%. On the other hand, for the Rungholt scene, there is almost no decrease in the area size, with the nodes being around 99% of the same size.

The overall area of the BVH is related to the number of AABB nodes used during the building/transformation phase. This means how many nodes turned out to have the best representation in terms of the surface area in the axis-aligned bounding box. From the results, as well from the 6.13 visualisation, the DiTO method works best for leaves and the hierarchy levels close to the leaves, resulting in a low number of AABB nodes. As the hierarchy levels get closer to the root, the number of used AABB nodes increases until almost all nodes in the hierarchy level are represented by the AABB nodes. The highest number of nodes is generally in leaves, which explains the low number of the AABB node representation in the whole BVH. This is evident from the results and the visualization in Figure 6.13 where the AABB nodes are colored blue. The AABB BVH is left gray for better comparison. In the Figure, the increase of the AABB nodes in levels near the root is noticeable for the DiTO method. In level 30, almost all nodes are represented by oriented bounding boxes, as the DiTO algorithm works well for nodes near leaves. As soon as the refitting process happens across multiple levels, the bounding volume expands, resulting in a possibly larger volume than the volume of an axis-aligned box.

This suggests the DiTO method would work better on shallow hierarchies. This is supported by the fact that for the Powerplant scene the DiTO method does not perform well. This scene consists of a high number of long thin axis-aligned primitives with a low average number of triangles per node suggesting deep constructed hierarchy. This reflects in the decrease of the summed area of the BVH by only 5.6% with a high percentage of area taken by the AABB nodes (84%). On the contrary, for all the versions of the ODOP method, the percentage of AABB nodes stays fairly consistent throughout the hierarchy levels.

Figure 6.19 visualizes the summed normalized area of each node in every level of the hierarchy for all scenes individually. The area is normalized by the axis-aligned scene box surface area for each BVH builder. The graphs demonstrate that the summed area in each level is generally lower for the OBB BVHs than for the AABB BVH. This difference is most noticeable in the higher hierarchy levels, which are closer to leaves. In the Rungholt scene, there is no large decrease in the area size due to most of the nodes in the OBB BVH being axis-aligned boxes. For comparison, for the Rotated Rungholt scene, the difference is most significant from all the scenes.

6.2.6 BVH building performance

As mentioned, the BVH building performance can be estimated by the time needed to construct the BVH and the GPU memory requirements. These two metrics are heavily related as every memory access is performance-demanding operation. The measured results for both metrics are shown in Table 6.1 in the **Memory increase compared to AABB** row and in the **Built time** row.

The results show that the construction time significantly increases for all versions of the OBB BVH across all scenes. This is due to the fact that the BVH needs to be built first and then transformed, which contributes to the overall building time of the BVH for the DiTO algorithm. The reason for the increased BVH construction time in all versions of the ODOP method is the number of normals used during the BVH building process. Instead of projecting a scene point onto three axes and comparing it to six values, as in the AABB BVH builder, the vertex is projected onto all generated normals and compared to twice the number of the normals maximal and minimal values. This also explains the increase in the memory requirements for the building/transforming process. For the ODOP method, the increase is given only by the increase in the size of buffers, which store maximal and minimal values for each normal. This means that with a higher ODOP degree, the memory requirements increase exponentially with the cubic speed. The ODOP method is significantly more resource-intensive in terms of memory and time when compared to the

DiTO method.

The final size of the OBB node is the same for all OBB BVHs, requiring 112 bytes of memory for each of them. In comparison, the AABB node only requires 64 bytes of memory. The main difference is that for the AABB BVH node, the bounds of the children's bounding boxes are stored as six float values for each child. For the OBB BVH, two 3x4 transformation matrices are stored in the node. These matrices transform a ray from world space to the local space of the children nodes' bounding volumes.

6.2.7 Ray tracing performance

The ray tracing performance is measured by two metrics: the number of intersection tests performed for each ray during the ray traversal in the hierarchy and the time it takes to render the scene. This time is given by ray throughput for simplified comparability. The number of all intersection tests performed per ray is visualised in Figures 6.14 and 6.15. The intersections were tracked only for one batch of primary rays. This metric stores the sum of all ray-triangle and ray-AABB/OBB intersection tests performed during the ray traversal in the BVH. A number of intersection tests of a ray with only triangles for each scene and different BVH are mapped by colour and shown as heatmaps in Figures 6.16 and 6.17. The red color indicates the higher number of intersection tests while the blue color indicates the opposite.

Graphs in Figure 6.18 show average number of intersections per primary ray for each scene and BVH constructed by the different methods. It is noticeable, that for most of the scenes, the number of all intersection tests is smaller for the OBB BVHs compared to the AABB BVH. Additionally, the decrease of the intersection tests for the 3-ODOP OBB BVH in the Hairball scene compared to the DiTO OBB BVH reflects higher ray throughput of the primary rays traced in the scene

From the heatmap visualisation, it is noticeable that the number of intersections needed for the OBB BVHs decreases in most of the scenes. In some cases, the DiTO method outperforms all the versions of the ODOP method, for example, in the *Conference*, the *Crown* and the *Powerplant* scenes. In the *Powerplant* scene, the 4-ODOP could have potentially outperformed the DiTo method. However, because of its extensive memory requirements, it could not be tested. In a few scenes, the ODOP method noted a smaller number of intersections; for example, in the *hairball* scene, the 4-ODOP BVH outperformed any other tested BVH.

On the other hand, for the *Crown* scene, the 3-ODOP method noted a higher number of intersections than the 2-ODOP method. This is given by the different sets of normal vectors used for the BVH building. The generation given for the 2-ODOP favoured the *Crown's* geometry orientation.

The Hairball scene, which, based on the oriented nature of its geometry and supported by Vitsas et al. (2023) [34], is a scene where OBB BVH performance significantly increases over the AABB BVH. In the measuring, this was also recorded by the increase of the ray throughput for at least two OBB BVH versions for each primary, AO and diffuse rays. The ray throughput was over 1.25 times higher for the 3-ODOP than for the AABB BVH for primary rays and 1.13 times higher for the diffuse rays. For the DiTO method, it was lower for each ray type than for the ODOP method but increased over 1.11 times for primary rays compared to the AABB BVH, 1.19 times for AO rays and 1.06 times for diffuse rays.

To compare the Rungholt and the Rotated Rungholt scenes, the Rungholt scene exhibited similar number of intersection tests for all the built BVHs. However, there was a significant decrease in the ray throughput for the DiTO method, which performed nearly 35% worse than the AABB BVH.

Nevertheless, for the rotated version of this scene, the difference in the number of intersection tests for AABB BVH compared to the OBB BVHs is significant. This fact is reflected by the ray tracing speed, where the improvement for the DiTO method is 1.42 times. However, for the ODOP method, despite the lower number of intersection queries, the increase in ray tracing performance is not that visible for primary rays and even decreases for AO and diffuse rays.

Generally, fewer intersection tests performed per ray, result in higher ray throughput. This is related to the tree quality measurements, such as BVH cost and BVH area, where better tree quality (smaller BVH cost and smaller BVH surface area) results in faster ray tracing. For the DiTO method, for the primary rays, there is a noted increase in the ray throughput for most of the scenes despite the higher BVH cost. This, however, resembles the number of intersection tests performed.

For the ODOP method, the ray throughput increases with the ODOP degree, which is accompanied by a reduction in the BVH cost as the ODOP degree increases. The ODOP method over-performs the DiTO method in the Tree, the Dragon, and the Hairball scene. For the AO and diffuse rays, the DiTO method generally results in better results than the ODOP method, especially for the larger scenes, which might result from not testing the higher ODOP degrees on these scenes. The AABB BVH noted better performance over any of the OBB BVH only in the Powerplant and Rungholt scene, which is an expected scenario.

6.3 Final Discussion

Both the DiTO and the ODOP methods aim to construct a high-quality OBB BVH which optimizes the ray tracing process. Both methods have different approaches, resulting in different outcomes on different scenes. As noted in the measured metrics results, the DiTO and the ODOP methods are very comparable, with one method performing better in number of scenes and the other performing better in the rest of the scenes. The scenes could be separated as follows: the DiTO method recorded better performance results in the Crown, the Conference, the Rotated Rungholt, and the Powerplant scene across all ray types. The ODOP method for any ODOP degree performed better on the Tree, Dragon, and the Hairball scene across all ray types. The last scene, the Rungholt scene, noted similar results for both methods.

The decrease in the ray throughput of the OBB BVHs despite the tighter fitting bounding volumes used in the BVH compared to the AABB BVH might be given by using a project that was highly optimised for the use of AABBs which favors the results of the AABB BVH. The higher memory accesses for the OBB BVHs might poorly affect the ray tracing performance.

6.3.1 Bottlenecks

The quality of the BVH constructed using the ODOP method is largely determined by the orientation of the generated normal vectors. The tightness of the volumes of nodes in the final BVH is heavily dependent on how closely these normals align with the scene's geometry. If the generated normals' orientation highly resembles the scene geometry's orientation, the BVH significantly outperforms the BVH constructed using the DiTO algorithm. On the contrary, if the geometry's orientation falls, for example, squarely between two normal vectors of the OBB BVH, it leads to a lower quality BVH. This was evident in the results recorded on the Rotated Rungholt scene.

The method is affected by high memory requirements during the construction. With the linear increase of the ODOP degree, the memory requirements increase exponentially, which also impacts the construction time. This creates a challenge for GPU as the GPU memory is very limited. The allocated buffers would need to be split to fit in the GPU memory. This fact affects the possibility of running the method with a higher ODOP degree which restricts the ability of the method for better representation of the scene geometry on larger scenes.

The main time performance bottlenecks for the DiTO method is the computational complexity of determining the best representing orientation for each node which contributes to the overall construction time. This is because multiple orientations have to be tested. Additionally, the AABB BVH may have already collapsed, favoring the AABB volumes, which could lead to an unbalanced tree for the OBB BVH, impacting the BVH quality. A major bottleneck of the method occurs during the refitting stage of the algorithm, which extends the pre-computed bounding volumes, often resulting in substituting these volumes with axis-aligned boxes as the better option. This undermines complex computations performed earlier during the selection of the best orientation for the set of vertices. Consequently, the resulting bounding volumes aren't as fit as they could be using OBBs.

6.3.2 Possible improvements

A significant improvement for the ODOP method would involve implementing a more complex algorithm to select specific normals, possibly with pre-processing of the scene geometry. Although this would increase pre-processing time, the ability of selecting fewer number of normals that would better match the scene geometry orientation would result in a reduce of the memory requirements of the method during the construction phase. Additionally, this would also shorten the construction time. However, this would bring a possible complexity challenges and optimization difficulties, particularly for scenes of an axis-aligned nature.

Running the DiTO method for a higher number of pre-selected normals (or different sets of normals) during the selection of projection coordinates phase could enhance its performance. The additional candidate orientations could better resemble the actual geometry, which would prevent the downfalls of the extensive refitting.

Additional possible testing could include both sets of generated/pre-selected normal vectors being switched between both presented methods. For the DiTO method to use the generated normals during the coordinate projection phase of the algorithm and for the ODOP method to use the pre-selected normals specified in the DiTO algorithm [17].

Another option of testing would be testing different constant values of the c_T, c_I constants, which could result in a different topology of the hierarchy for the ODOP method. Testing hardware can also influence the results as newer GPUs use better optimisation techniques, which could favour the intense data fetching from memory for the OBB BVH.

The DiTO method shows improved tighntness of the volumes at lower hierarchy levels near the leaves, mainly due to the refitting phase of the algorithm, which expands the bounding boxes of nodes closer to the root. This is noticeable in the visualisations of the BVH, where there was a significant fast increase of the number of the AABB node representations as the hierarchy level decreased. Beyond a certain point, the axis-aligned bounding volumes represented most of the BVH nodes.

One possible improvement of the method would be tracking this point during the BVH transformation by setting a specific threshold. During the transformation, after the percentage of AABB nodes exceeded this threshold, a different method of OBB construction could be implemented. This could result in a strategic switch between different methods resulting in improved quality of the final BVH.

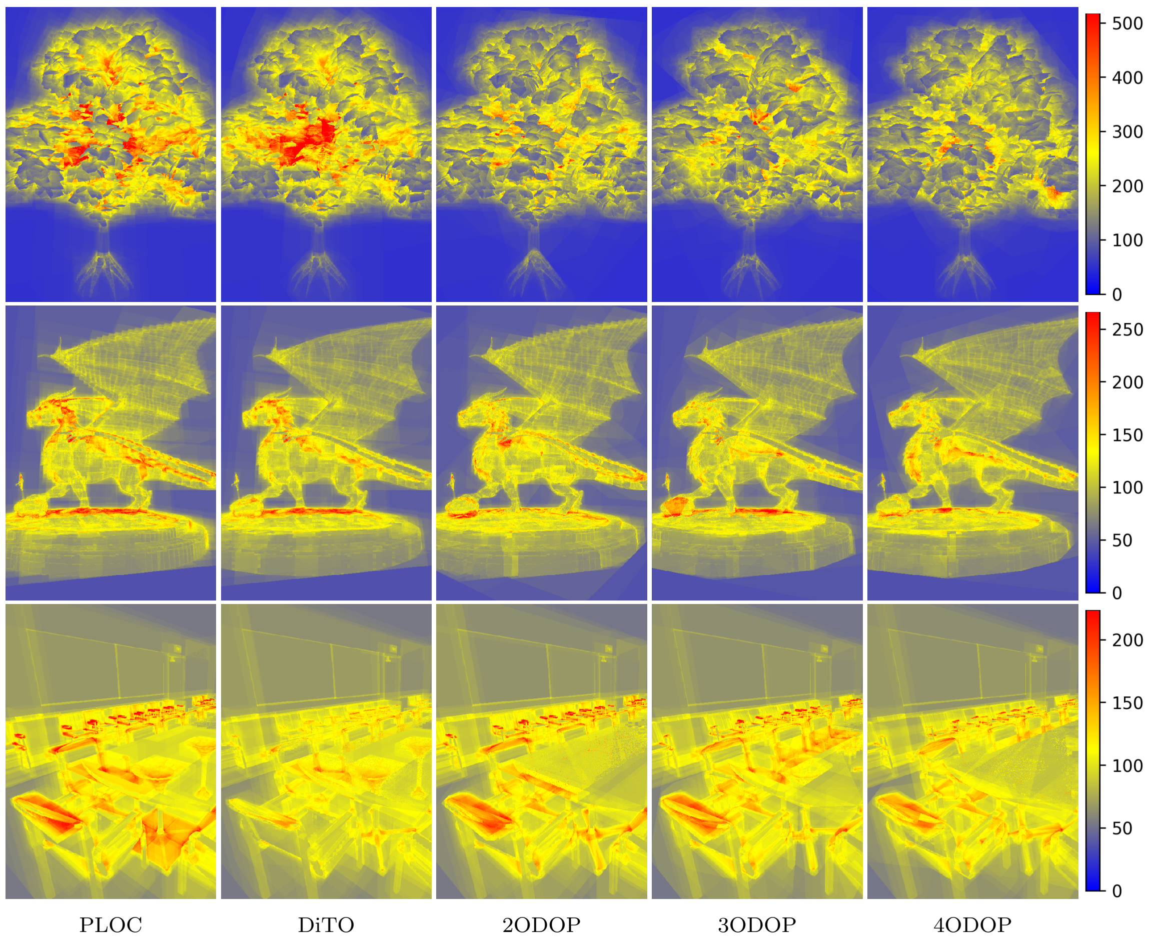


Figure 6.14: Comparisons of AABB BVH constructed by the PLOC algorithm and OBB BVH built using the DiTO and the 2-ODOP, 3-ODOP, and 4-ODOP methods, respectively. The number of intersections for primary rays is mapped to colour and shown as a heat map. The red colour indicates the highest number of intersections per ray. The number of intersections is represented as a sum of all ray-AABB/OBB and ray-triangle intersection tests Compared scenes are the *Tree*, *Dragon*, *Conference* scene in respective rows.

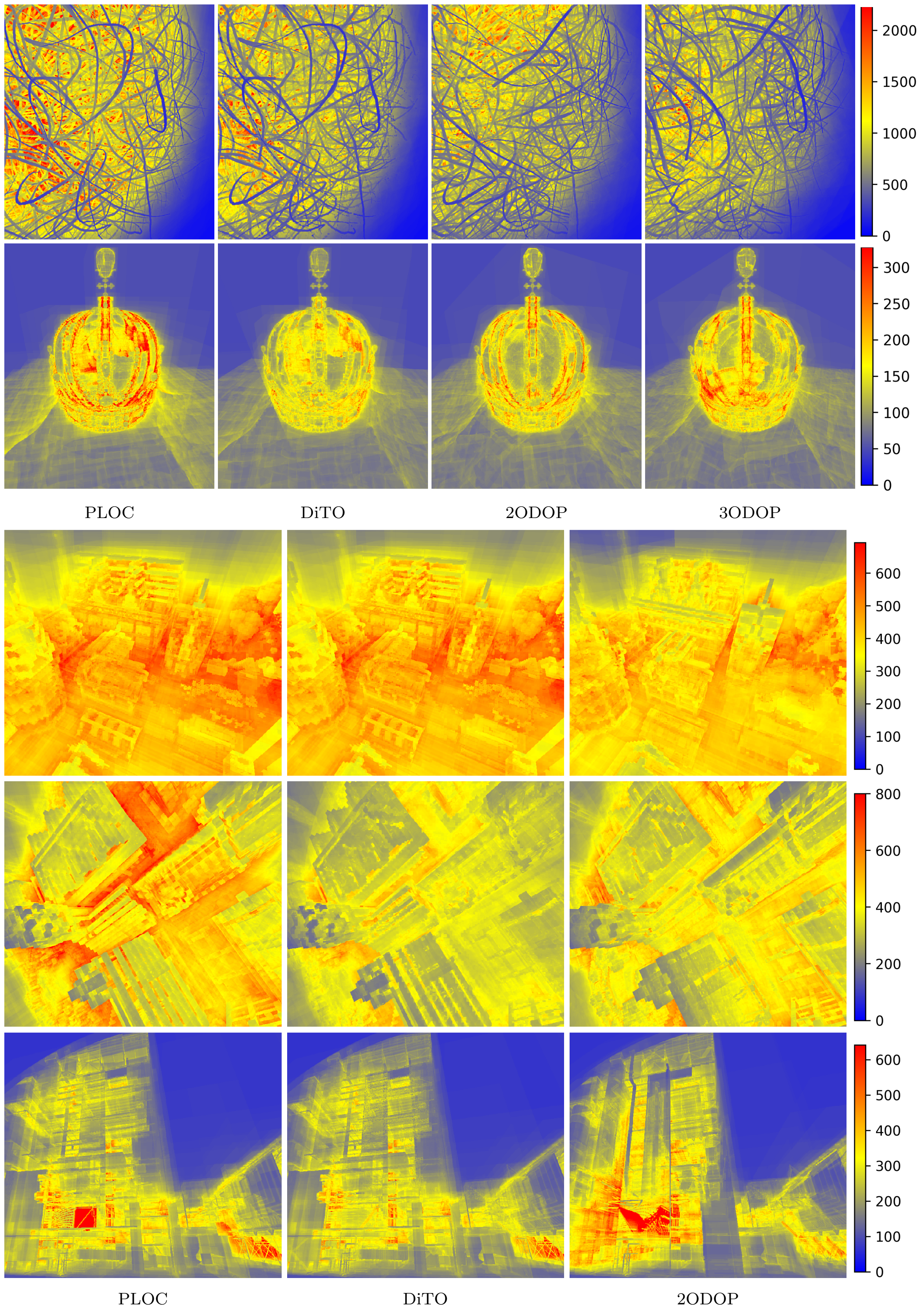


Figure 6.15: Comparisons of AABB BVH constructed by the PLOC algorithm and OBB BVH built using the DiTO and the 2-ODOP and 3-ODOP method, respectively, for the first part. For the second part, the 3-ODOP method was not used. The number of intersections for primary rays is mapped to colour and shown as a heat map. The red colour indicates the highest number of intersections per ray. The number of intersections is represented as a sum of all ray-AABB/OBB and ray-triangle intersection tests. Compared scenes are the *Hairball* and the *Crown* scene in respective rows in the first part and the *Rungholt*, *Rotated Rungholt,* and the *Powerplant* scene in the rows in the second part. These are the largest scenes from the collection.

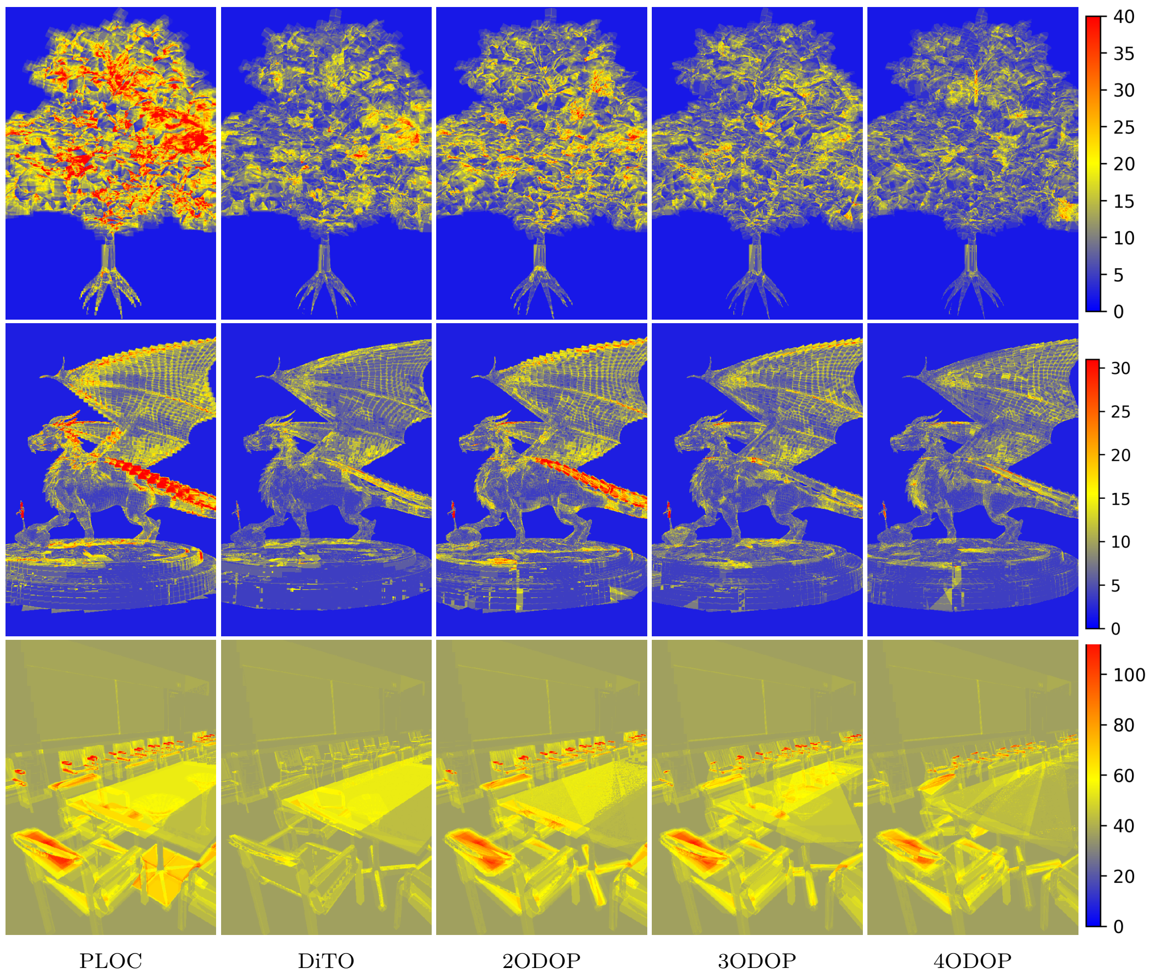


Figure 6.16: Comparisons of AABB BVH constructed by the PLOC algorithm and OBB BVH built using the DiTO and the 2-ODOP, 3-ODOP, and 4-ODOP methods, respectively. The number of ray-triangle intersection tests for primary rays is mapped to colour and shown as a heat map. The red colour indicates the higher number of intersections per ray. Compared scenes are the *Tree*, *Dragon*, *Conference* scenes in respective rows.

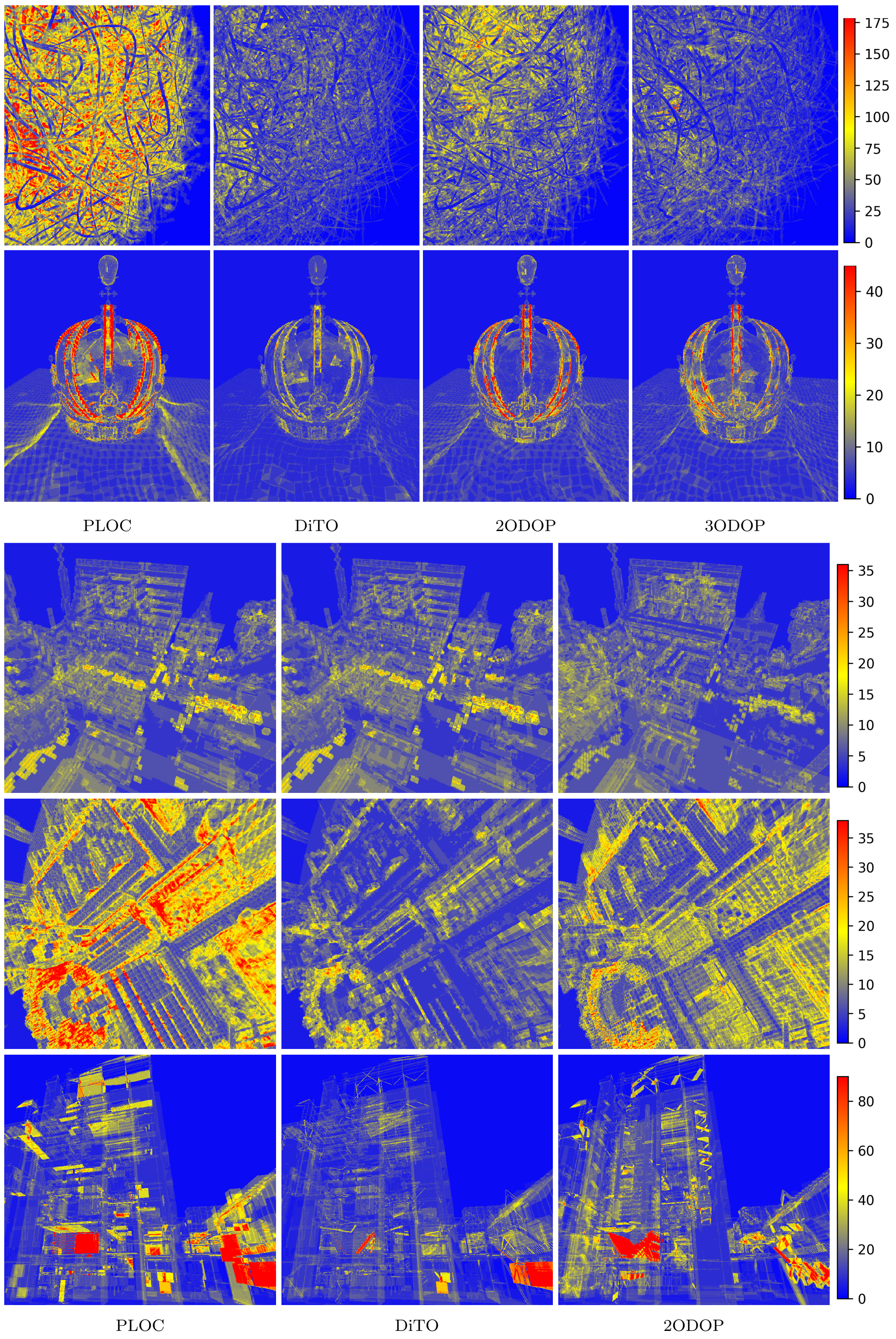


Figure 6.17: Comparisons of AABB BVH and OBB BVH constructed using the DiTO and the 2-ODOP and 3-ODOP method, respectively. The number of ray-triangle intersection tests for primary rays is mapped to colour and shown as a heat map. The red colour indicates the higher number of intersection tests per ray. Compared scenes are the *Hairball* and the *Crown* scene in respective rows in the first part and the *Rungholt*, *Rotated Rungholt,* and the *Powerplant*.

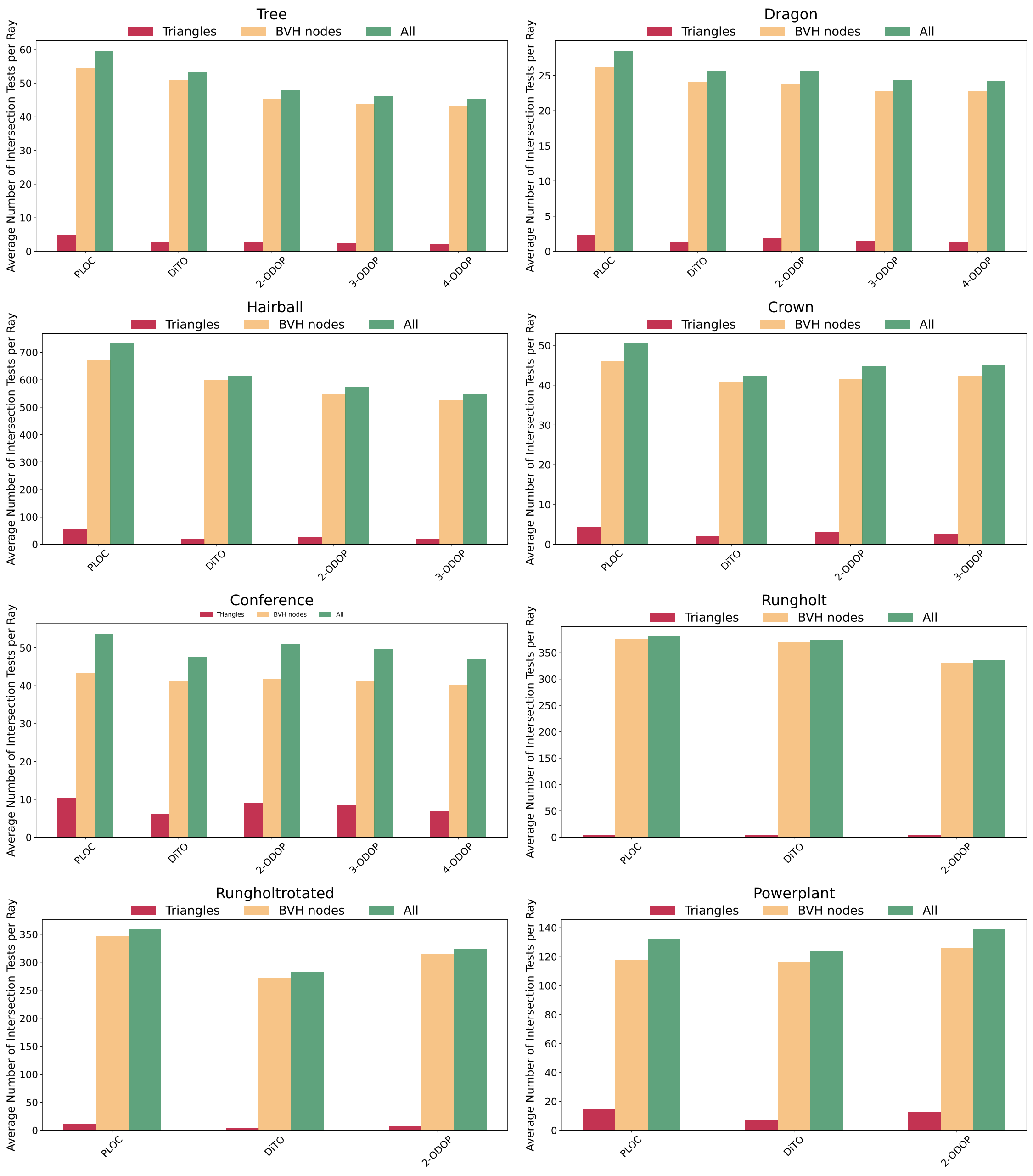


Figure 6.18: Graph visualisations of number of intersection tests performed during ray tracing of primary rays for each BVH builder (PLOC, which produces AABB BVH, and DiTO, 2-ODOP, 3-ODOP, and 4-ODOP OBB BVHs) and for every scene. 3-ODOP and 4-ODOP cannot be tested on larger scenes due to their high memory requirements which results in not including these BVH builders in the specific graphs. The graphs visualise the individual builders on the horizontal axes and the number of intersection tests on the vertical axes. The red bar indicates number of ray-triangle intersection tests. The beige bar indicates number of ray-bounding volume(AABB/OBB) intersection tests. The green bar represents the summed value, meaning all intersection tests performed for primary rays during rendering of one frame. Name of the specific scene is written above the graph.

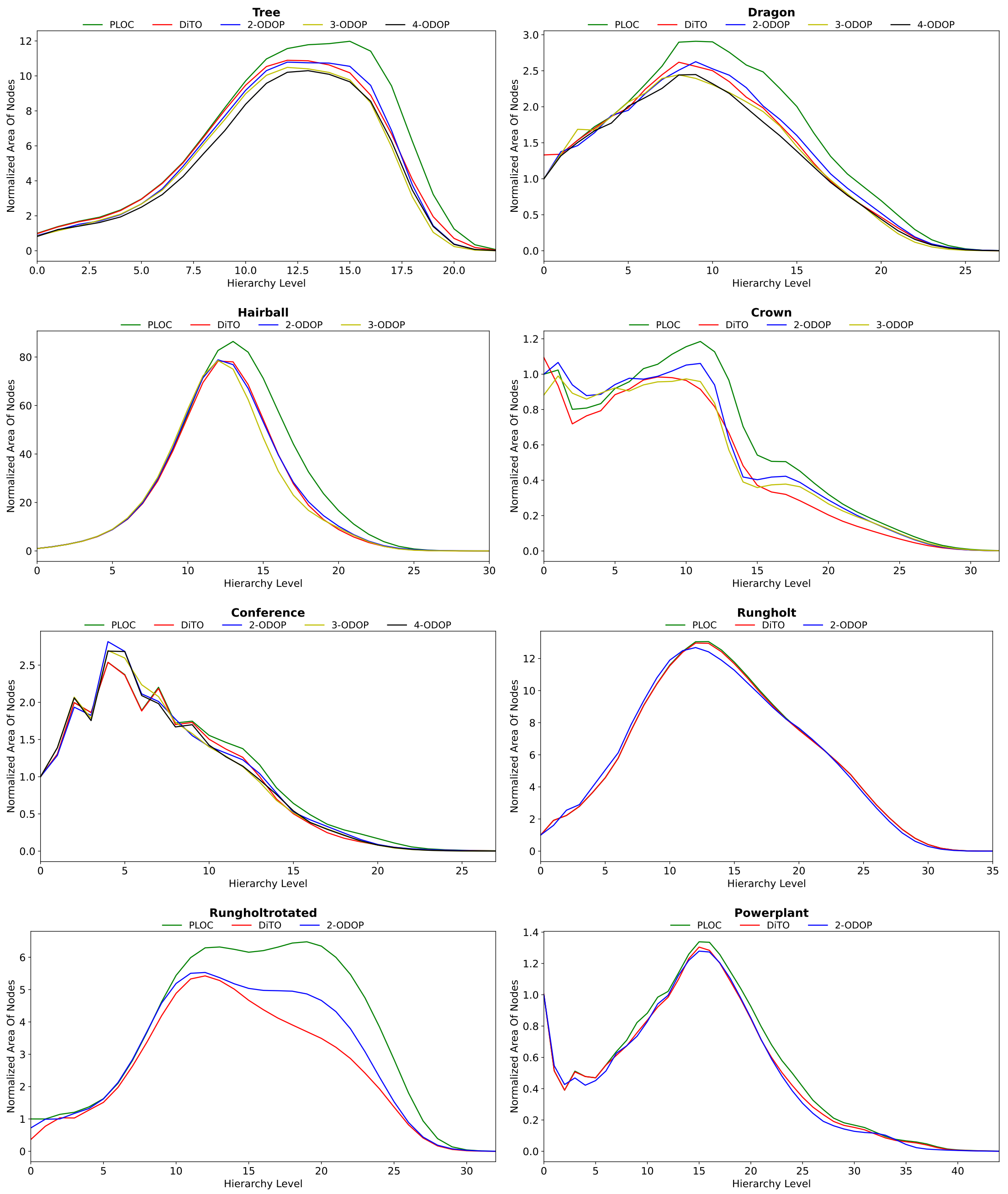


Figure 6.19: Normalized summed-area for all nodes in every hierarchy level for each scene. The area is normalized by the axis-aligned scene box's surface area. With the higher hierarchy level, the summed area of nodes gets smaller for the OBB BVHs compared to the AABB BVH.

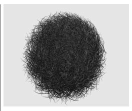
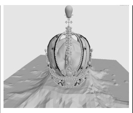
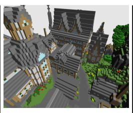
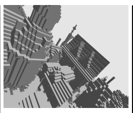
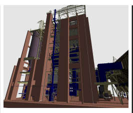
								
Scene #Tris	**Tree** 313k	**Dragon** 800k	**Hairb.** 2.8M	**Crown** 3.5M	**Conf.** 331k	**Rungh.** 6.7M	**RotRun.** 6.7M	**Powerp.** 12.7M
BVH cost								
AABB	439.21	138.75	2717.51	59.6	95.2	623.12	358.62	90.41
DiTO	489.18	149.91	2724.01	61.14	108.18	816.6	329.21	92.4
2-ODOP	496.45	159.63	2858.73	70.86	116.05	814.45	381.17	97.13
3-ODOP	466.41	149.7	2667.27	66.09	113.72			
4-ODOP	452.78	144.64			112.69			
Leaves summed area/Inner nodes summed area								
AABB	16.01/118.87	4.03/37.8	29.43/748.65	1.58/16.95	3.82/23.65	6.89/197.4	7.9/107.0	2.03/21.2
DiTO	8.66/111.0	2.24/34.28	11.57/640.21	0.78/14.56	3.1/22.93	6.89/196.47	2.98/78.71	1.28/20.67
2-ODOP	12.03/104.66	3.01/33.85	18.42/643.5	1.3/15.7	3.63/23.04	6.86/195.68	5.45/87.54	1.56/19.98
3-ODOP	10.44/99.96	2.73/32.42	17.24/617.11	1.14/14.82	3.48/22.81			
4-ODOP	9.59/98.16	2.57/31.73			3.41/22.9			
Average number of triangles per leaf								
AABB	3.6	2.26	6.3	2.42	2.46	2.21	2.27	2.53
DiTO	3.6	2.26	6.3	2.42	2.46	2.21	2.27	2.53
2-ODOP	4.23	2.9	6.32	3.0	3.29	2.24	2.56	3.71
3-ODOP	4.19	2.86	5.03	2.9	3.04			
4-ODOP	4.13	2.82			3.05			
Percentage of AABB nodes out of all nodes (%), AABB = 100%								
DiTO	16.16	4.88	4.66	12.69	64.8	99.73	2.41	35.85
2-ODOP	16.16	32.51	9.7	20.46	35.42	98.96	0.14	46.79
3-ODOP	5.31	19.54	2.76	8.23	22.62			
4-ODOP	2.31	14.25			17.15			
Percentage of area of AABB nodes (%), AABB = 100%								
DiTO	60.51	54.79	60.92	36.59	85.53	95.67	30.62	84.72
2-ODOP	19.61	44.14	11.04	52.33	90.7	86.1	2.36	82.42
3-ODOP	7.15	26.86	2.84	19.76	85.99			
4-ODOP	2.9	20.62			83.64			
Increase of BVH area (%), AABB = 100%								
DiTO	88.59	86.17	83.73	81.68	94.77	99.52	71.01	94.4
2-ODOP	86.52	88.11	85.07	91.8	97.12	99.14	80.92	92.71
3-ODOP	81.85	84.02	81.53	86.16	95.75			
4-ODOP	79.89	82.01			95.81			
Memory increase compared to AABB (%), AABB = 100%								
DiTO	164.14	202.34	136.73	195.68	194.0	204.8	201.91	191.23
2-ODOP	456.23	456.23	456.24	456.24	456.2	456.24	456.24	456.24
3-ODOP	1168.7	1168.69	1168.71	1168.71	1168.61			
4-ODOP	2556.13	2556.11			2555.93			
Built time (s)								
AABB	0.01	0.02	0.04	0.07	0.01	0.2	0.13	0.18
DiTO	0.04	0.05	0.15	0.23	0.02	0.42	0.43	0.81
2-ODOP	0.51	1.53	3.76	7.13	0.57	17.55	8.58	16.8
3-ODOP	2.72	8.1	21.62	39.7	3.0			
4-ODOP	13.94	42.97			15.27			
Primary rays (MRays/s)								
AABB	176.60	255.59	16.61	168.54	508.02	**233.62**	74.37	**75.81**
DiTO	239.43(1.36x)	270.18(1.06x)	18.49(1.11x)	**216.14**(1.28x)	**639.10**(1.26x)	135.36(0.58x)	**105.92**(1.42x)	74.48(0.98x)
2-ODOP	285.50(1.62x)	260.29(1.02x)	18.02(1.08x)	157.30(0.93x)	438.15(0.86x)	140.35(0.60x)	75.83(1.02x)	69.35(0.91x)
3-ODOP	307.17(1.74x)	265.12(1.04x)	**20.68**(1.25x)	172.19(1.02x)	457.88(0.90x)			
4-ODOP	**338.14**(1.91x)	**285.66**(1.12x)			489.93(0.96x)			
AO rays (Mrays/s)								
AABB	167.50	**197.31**	17.15	141.34	614.07	**360.70**	135.93	**221.89**
DiTO	160.43(0.96x)	187.74(0.95x)	20.37(1.19x)	**156.97**(1.11x)	**642.99**(1.05x)	232.65(0.64x)	**162.89**(1.20x)	172.89(0.78x)
2-ODOP	161.64(0.97x)	178.08(0.90x)	17.69(1.03x)	136.88(0.97x)	460.73(0.75x)	227.66(0.63x)	122.14(0.90x)	159.86(0.72x)
3-ODOP	**186.24**(1.11x)	193.79(0.98x)	**20.76**(1.21x)	150.95(1.07x)	486.49(0.79x)			
4-ODOP	181.06(1.08x)	197.29(1.00x)			512.93(0.84x)			
Diffuse rays (Mrays/s)								
AABB	79.17	109.06	6.36	75.62	223.34	**67.74**	24.21	**54.25**
DiTO	78.57(0.99x)	105.87(0.97x)	6.72(1.06x)	**84.95**(1.12x)	**238.74**(1.07x)	42.26(0.62x)	**26.69**(1.10x)	48.21(0.89x)
2-ODOP	80.00(1.01x)	98.18(0.90x)	5.89(0.93x)	75.53(1.00x)	171.62(0.77x)	42.45(0.63x)	20.81(0.86x)	45.86(0.85x)
3-ODOP	**98.57**(1.25x)	105.93(0.97x)	**7.19**(1.13x)	78.71(1.04x)	183.92(0.82x)			
4-ODOP	90.03(1.14x)	**112.49**(1.03x)			193.71(0.87x)			

Table 6.1: Comparison of each BVH constructed using different builder. AABB BVH represents a baseline, built using the PLOC algorithm. DiTO represents an OBB BVH built by method using the DiTO algorithm for transforming an AABB BVH (built by PLOC algorithm). 2-ODOP, 3-ODOP, 4-ODOP represent OBB BVH constructed using method utilizing ODOPs with different ODOP degree (2, 3, and 4). Measured metrics for every BVH are written in row sections. Each scene is represented by their specific column.

Chapter 7

Conclusion

This thesis aims to implement and compare methods for the efficient construction of the high-quality bounding volume hierarchy (BVH) with the use of oriented bounding boxes (OBBs) as the bounding volume.

In the first part of the thesis, the theoretical background was set, which is necessary for understanding the technical terms used throughout this thesis. The ray tracing and bounding volume hierarchies were described, and the BVH construction algorithms were presented. This includes the *PLOC* algorithm [20] along with the *DiTO* algorithm [17] and Morton Codes used in the implementation of the PLOC algorithm.

The method proposed by Vitsas et al. 2023 [34] and the method developed by Sabino et al. 2023 [28] were presented in chapter 4 as methods for high-quality OBB BVH construction.

To unify these methods inside a common framework, the project's source code with the PLOC algorithm implemented by Daniel Meister and Jiří Bittner was utilized [5]. The project uses the *GPU Ray Traversal Framework*[2], which offers efficient scene rendering methods described in the *Understanding the Efficiency of Ray Traversal on GPUs* paper [1]. The methods were implemented in C++ and CUDA 11.

Using the PLOC algorithm and the project with the already implemented PLOC algorithm simplifies the comparison of different BVH structures. This includes the OBB BVH constructed using ODOPs, OBB BVH built with the use of the DiTO algorithm executed over AABB BVH, and AABB BVH itself. These structures are built within this project, ensuring that the hierarchies are constructed in a consistent manner using identical methods and strategies. This uniformity benefits the effective evaluation of the presented methods for OBB BVH construction.

The resulting BVHs were tested on eight different scenes with various complexities. The BVHs were then compared in terms of their construction and ray tracing speeds. Other performance-related metrics were also measured, such as the BVH cost, the surface area of all the nodes of the BVH summed together, memory requirements, and the percentage of AABB nodes in the hierarchy. The AABB hierarchy constructed by the PLOC algorithm was used as a baseline for the measuring. For the ODOP method, different ODOP degrees were tested: 2-ODOP, 3-ODOP and 4-ODOP. The DiTO algorithm transforms an already generated AABB BVH into an OBB BVH. For this, the AABB BVH generated by the PLOC algorithm was utilised.

Compared to the AABB BVH, the OBB BVHs offer tighter fitting bounding volumes, which results in a smaller BVH surface area and a decrease in the number of intersection tests performed during the ray traversal of the hierarchy. The DiTO method results in significantly tight fitting volumes in the higher hierarchy levels, levels near leaves. The decrease in the tightness of the volumes of nodes closer to the root is due to the refitting part of the method. This part is performed from the leaf nodes up the hierarchy, which expands the bounding boxes of nodes closer to the root.

The ODOP method offers a constant improvement of the tightness of the bounding volumes at each hierarchy level. With the higher ODOP degree, the surface area of the BVH decreases. However, the heavy dependency of the method on the generated normals influences the overall effectiveness and efficiency of the method. As a result, while the ODOP method can generate more optimized bounding volumes in all hierarchy levels, it depends on the orientation of the generated normals aligning with the orientation of the scene primitives.

The built-time and memory requirements of the methods are much higher than those of the PLOC algorithm constructing the AABB BVHs. With the higher ODOP degree, the memory and built-time requirements increase exponentially, which results in more memory and time-demanding method out of the two.

Despite the expected higher ray throughput for the OBB BVHs given by the lower number of intersection tests and the BVH summed area, the AABB BVH overperforms the OBB BVHs in several scenes. This might be given by the use of the project that was originally heavily optimised for the use of AABB bounding volumes.

In the Hairball scene which is a scene that highlights the advantages of the uses of OBB BVH over the AABB BVH, the ODOP method demonstrated better results. The method proposed by Vitsas et al. 2023 [34] improved the ray tracing performance by 1.06 times for diffuse rays. While the 3-ODOP version of the method developed by Sabino et al. 2023 [28] recorded a performance increase of 1.13 times. On the other hand, the 2-ODOP method recorded a decrease which illustrates how significantly can the selection of normals impact the final results.

All of the tested OBB bounding volume hierarchies resulted in lower surface area than the AABB BVH, and the number of intersection tests performed during the ray traversal decreased. On the contrary, the cost of the hierarchy increased, which resembles the increased complexity of the ray-OBB intersection tests. For scenes with many differently oriented primitives, such as the Hairball scene, the OBB significantly outperformed the AABB BVH. For these types of scenes, the OBB BVH is a better option. Additionally, if the construction time and the memory requirements weren't a problem, the ODOP method using a high ODOP degree would outperform the DiTO BVH.

7.1 Future Work

Both presented methods used for constructing high-quality OBB BVH showed their advantages and disadvantages across the eight tested scenes. These findings could be used to improve the methods, potentially leading to the creation of a bounding volume hierarchy construction algorithm that combines the advantages of both methods.

One possible strategy of combining the methods would be using the DiTO method for constructing the OBB BVH mainly in the higher hierarchy levels where it records the best results. This approach would consist of constructing the initial BVH using the ODOP method. After constructing the BVH, the DiTO algorithm would be called over every node in a hierarchy level higher than a certain threshold. During the refitting phase, the DiTO algorithm would monitor the number of nodes where the surface area didn't improve. Once this number surpasses another threshold, the algorithm could stop processing nodes above this hierarchy level (closer to the root node), potentially reducing the time needed for the BVH transformation.

Appendix A

Application manual

This project implements two methods for construction of OBB BVH. The project originially contains an implementation of the PLOC-BVH builder based on parallel locally ordered clustering [20] by Meister and Bittner. The builder is integrated into the ray tracing code of Aila et al. [2].

As an addition, the project was modified to include two methods for OBB BVH construction proposed by Vitsas et al. [34] and Sabino et al. [28].

The following appendix contains specifics around running the application and measuring similar data as the added measuring results. The specifics and configuration is written in this application manual.

The project requirements are:

- CUDA 11.x, GPU with compute capability 5.x
- Windows: Microsoft Visual Studio 2019+
- Microsoft Windows 10+

Instructions to Run Test Application

- As the CUDA files are compiled during runtime, specific macros need to be set in the `gpu-ray-traversal/src/Configure.hpp` file. These are: **ALG_TO_RUN** which selects specific builder to build the BVH.
  - 0 for the DiTO method
  - 1 for the ODOP method
  - 2 for the PLOC method

  If ALG_TO_RUN is set to be 1, the **ODOP_DEGREE** variable needs to be set additionally. Number selected runs the ODOP method with specific degree of ODOP (for example, **ODOP_DEGREE = 2** runs the builder with 2-ODOP).

Bin executables

As the cuda files are compiled at runtime, the bin executables have to be placed inside the src directory. Specifically, in `src/gpu-ray-traversal` directory. Specific executable should be run only with specific values set in the `Configure.hpp` file to avoid errors as the CUDA files are compiled during runtime.

There are {builder}_Release.exe files to build the BVH using the specific builder (builder = PLOC, DiTO, 2ODOP, 3ODOP, 4ODOP) and render the Conference scene which is set as default. Other scenes can be selected in the App.

The {builder}_Visualise.exe binary serves to visualise the built BVH. In that order, the **VISUALISE** macro in the `Configure.hpp` file needs to be set to 1 and the specific hierarchy level in which the BVH will be visualised as well.

These executables were generated using Microsoft Visual Studio 2019 and CUDA 11.4.

Added **benchmark.cmd** file measures the performance of a BVH built according to the `Configure.hpp` file. The specific builder executable is run in order to replicate the measuring on eight scenes which are stored in `gpu-ray-traversal/scenes` folder. The logs are stored in `benchmark.log` file.

To run benchmark for specific builder and specific scene with camera coordinates, example command is shown below:

```
DiTO_Release benchmark --mesh=scenes/rt/conference/conference.obj
--camera="6omr/04j3200bR6Z/0/3ZEAz/x4smy19///c/05frY109Qx7w////m100"
```

As most of the scenes are too large to include in the project, they can be given out on demand.

Troubleshooting Build Errors

If an error is encountered during the build:

- Check the directory for build customizations:
  ```
  C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\V120\BuildCustomizations\
  ```
  It should contain a props file (e.g., `'CUDA 11.x.props'`). If not, find it and copy it into this directory.
- If you use a different version of CUDA, then check the project file:
  ```
  cub/cub.vcxproj
  ```
  Replace the CUDA version in this file with the appropriate version (e.g., replace `'CUDA 11.4'` with `'CUDA 11.6'`).
- If an error with cudacache files is encountered, the cudacache files usually need to be regenerated, so deleting the folder helps.

Additionally, the project can be built using Microsoft Visual Studio, this will ensure everything works as expected if an error is encountered.

PLOC and ODOP Configuration

The PLOC method and the ODOP method are configurable via `'gpu-ray-traversal/ploc.cfg'` file and `'gpu-ray-traversal/odop.cfg'` file respectively.

- radius: 1-128
- morton60: 0-1
- adaptiveLeafSize: 0-1
- maxLeafSize: 1-64

Appendix B

Content of electronic appendix

```
images/
src/
  scenes/
  gpu-ray-traversal/
    cub/
    benchmark.cmd
    DiTO_Release.exe
    DiTO_Visualise.exe
    PLOC_Release.exe
    PLOC_Visualise.exe
    20DOP_Release.exe
    20DOP_Visualise.exe
    30DOP_Release.exe
    30DOP_Visualise.exe
    40DOP_Release.exe
    40DOP_Visualise.exe
    src/
      framework/
      rt/
        bvh/
        cuda/
        dito/
        kernels/
        OBB/
        ODOP/
        ploc/
        ray/
        visualiser/
        App.cpp
        App.hpp
        Configure.hpp
        Scene.cpp
        Scene.hpp
        Util.cpp
        Util.hpp
```

Bibliography

[1] Timo Aila and Samuli Laine. “Understanding the Efficiency of Ray Traversal on GPUs”. In: vol. 2009. Aug. 2009, pp. 145–149. DOI: `10.1145/1572769.1572792`.

[2] Timo Aila and Samuli Laine. *Understanding the Efficiency of Ray Traversal on GPUs.* `https://github.com/matt77hias/GPURayTraversal`. 2016.

[3] Benedikt Bitterli. *Rendering resources.* https://benedikt-bitterli.me/resources/. 2016.

[4] Russell A. Brown. “Building a Balanced k-d Tree in O(kn log n) Time”. In: *ArXiv* abs/1410.5420 (2014). URL: `https://api.semanticscholar.org/CorpusID:16995974`.

[5] Jiri Bittner Daniel Meister. *Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction.* `https://github.com/meistdan/ploc`. 2018.

[6] Simena Dinas and Jose Bañón. “A literature review of bounding volumes hierarchy focused on collision detection”. In: 17 (June 2015), pp. 49–62.

[7] Leonardo R. Domingues and Hélio Pedrini. “Bounding volume hierarchy optimization through agglomerative treelet restructuring”. In: *Proceedings of the 7th Conference on High-Performance Graphics, HPG 2015, Los Angeles, California, USA, August 7-9, 2015.* Ed. by Michael C. Doggett et al. ACM, 2015, pp. 13–20. DOI: `10.1145/2790060.2790065`. URL: `https://doi.org/10.1145/2790060.2790065`.

[8] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3D Technology).* Dec. 2004, p. 77. ISBN: 1-55860-732-3.

[9] Jeremy Fisher et al. “Ray Tracing Visualization Toolkit”. In: (Aug. 2011).

[10] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. “Simpler and faster HLBVH with work queues”. In: Aug. 2011, pp. 59–64. DOI: `10.1145/2018323.2018333`.

[11] Andrew S. Glassner, ed. *An introduction to ray tracing.* GBR: Academic Press Ltd., 1989. ISBN: 0122861604.

[12] Lawrence Berkeley National Laboratory. *Conference Room.* `https://radsite.lbl.gov/mgf/scenes.html`.

[13] Samuli Laine and Tero Karras. “Two Methods for Fast Ray-Cast Ambient Occlusion”. In: *Comput. Graph. Forum* 29 (June 2010), pp. 1325–1333. DOI: `10.1111/j.1467-8659.2010.01728.x`.

[14] Elmar Langetepe and Gabriel Zachmann. *Geometric Data Structures for Computer Graphics.* Jan. 2006, p. 10. ISBN: 9780367803735. DOI: `10.1201/9780367803735`.

[15] Christian Lauterbach et al. “Fast BVH construction on gpus”. In: *Computer Graphics Forum* 28 (Apr. 2009), pp. 375–384. DOI: `10.1111/j.1467-8659.2009.01377.x`.

[16] Christian Lauterbach et al. “Fast BVH construction on gpus”. In: *Computer Graphics Forum* 28 (Apr. 2009), pp. 375–384. DOI: `10.1111/j.1467-8659.2009.01377.x`.

[17] Eric Lengyel. *Game Engine Gems 2*. 1st. USA: A. K. Peters, Ltd., 2011. ISBN: 1568814372.

[18] J. David MacDonald and Kellogg S. Booth. “Heuristics for ray tracing using space subdivision”. In: *The Visual Computer* 6.3 (May 1990), pp. 153–166. ISSN: 1432-2315. DOI: `10.1007/BF01911006`. URL: `https://doi.org/10.1007/BF01911006`.

[19] Morgan McGuire. *Computer Graphics Archive*. `https://casual-effects.com/data`. July 2017. URL: `https://casual-effects.com/data`.

[20] Daniel Meister and Jiri Bittner. “Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction”. In: (Feb. 2017).

[21] Daniel Meister et al. “A Survey on Bounding Volume Hierarchies for Ray Tracing”. In: *Computer Graphics Forum* 40 (May 2021), pp. 683–712. DOI: `10.1111/cgf.142662`.

[22] Planet Minecraft. *Neu Rungholt*. `https://www.planetminecraft.com/project/neu-rungholt/`.

[23] G. M. Morton. *A Computer Oriented Geodetic Data Vase; And a New Technique in File Sequencing*. Mar. 1966.

[24] T. Müller. *Visualisierung in der Relativitätstheorie, PhD thesis*. Eberhard-Karls-Universität zu Tübingen, 2006.

[25] NVIDIA. *What is Path Tracing?* 2022. URL: `https://blogs.nvidia.com/blog/what-is-path-tracing/`.

[26] NVIDIA. *What’s the Difference Between Ray Tracing and Rasterization?* 2018. URL: `https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/`.

[27] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128006455.

[28] Rodolfo Sabino et al. “Building Oriented Bounding Boxes by the intermediate use of ODOPs”. In: *Computers Graphics* 116 (Aug. 2023). DOI: `10.1016/j.cag.2023.08.028`.

[29] Rodolfo Sabino et al. “Fast and Robust Ray/OBB Intersection Using the Lorentz Transformation”. In: Aug. 2021, pp. 519–528. ISBN: 978-1-4842-7184-1. DOI: `10.1007/978-1-4842-7185-8_32`.

[30] Lawrence D. Stone. “Theory of Optimal Search”. In: 1975. URL: `https://api.semanticscholar.org/CorpusID:122235548`.

[31] L. Szirmay-Kalos and G. Márton. “Worst-case versus average case complexity of Ray-shooting”. In: *Computing* 61.2 (Oct. 1998), pp. 103–131. ISSN: 0010-485X. DOI: `10.1007/BF02684409`. URL: `https://doi.org/10.1007/BF02684409`.

[32] Timo Viitanen et al. “PLOCTree: A Fast, High-Quality Hardware BVH Builder”. In: 1.2 (Aug. 2018). DOI: `10.1145/3233309`. URL: `https://doi.org/10.1145/3233309`.

[33] Marek Vinkler, Jiri Bittner, and Vlastimil Havran. “Extended Morton codes for high performance bounding volume hierarchy construction”. In: July 2017, pp. 1–8. DOI: `10.1145/3105762.3105782`.

[34] N. Vitsas et al. “Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees”. In: *Computer Graphics Forum* 42.2 (2023), pp. 245–254. DOI: `https://doi.org/10.1111/cgf.14758`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14758`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14758`.

[35] Ingo Wald. “Realtime ray tracing and interactive global illumination”. PhD thesis. Jan. 2004.

[36] Gabriel Zachmann et al. *Real-Time Collision Detection for Dynamic Virtual Environments. IEEE VR Tutorials.* Bonn, Germany: IEEE Computer Society, 2005.

[37] Jiří Žára et al. *Moderní počítačová grafika, 2. vydání.* Computer Press, 2004. ISBN: 80-251-0454-0.