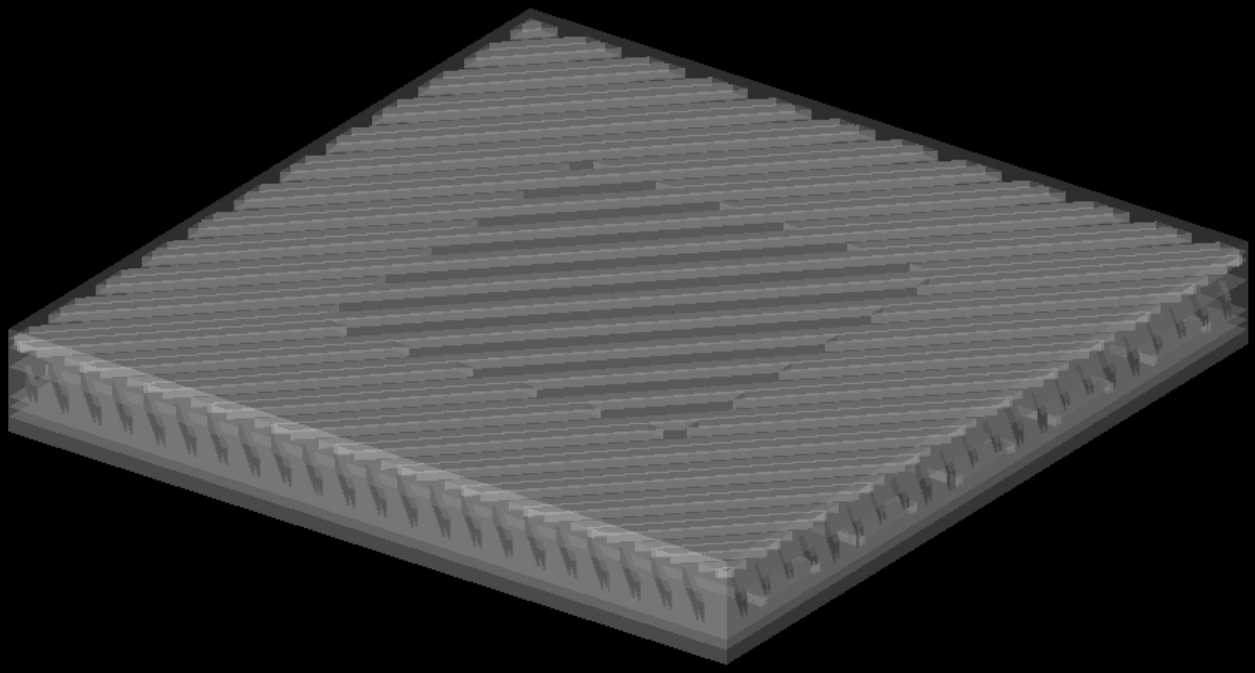
CPU-Based Ray Tracing For Semiconductor Structures

Yimin Zhou



Delft University of Technology

TU Delft ASML

CPU-Based Ray Tracing For Semiconductor Structures

by

Yimin Zhou

Student Name	Student Number
Yimin Zhou	5881870

Primary supervisor:	Ricardo Marroquim
Daily supervisor:	Yang Chen
Company supervisor:	Wouter Bijlsma & Ton Van Den Heuvel
Faculty:	EEMCS, TUDelft

Abstract

In this thesis, we investigate a CPU-based ray tracer tailored for semiconductor structures. This ray tracer addresses the need for high-quality visualization in environments where GPU acceleration is unavailable. Our primary objective is to achieve interactive visualization performance with quality levels on par with a basic triangle rasterizer employing a flat-shaded lighting model.

The research focuses on identifying optimal BVH configurations and developing efficient Oriented Bounding Box (OBB) ray tracing methods. By leveraging OBB clustering and pre-cached ray directions, we aim to mitigate the computational overhead associated with OBB tracing. The result is a prototype that demonstrates the feasibility of CPU-based ray tracing for real-time applications.

This work contributes to the broader discourse on rendering techniques in GPU-limited scenarios, offering valuable insights and laying the groundwork for future research and development.

Contents

1

Introduction

This thesis investigates an area of real-time graphics rendering often overshadowed by the advancements in high-fidelity GPU-accelerated rendering: real-time CPU-based raytracing for triangle-based models, which is, in the case of this thesis, intended for visualization of semiconductor structures. The motivation for this research stems from a practical challenge faced by ASML: the reliance on legacy systems where GPU acceleration is not an option for some of their customers.

The traditional method for rendering in such environments has relied on a legacy OpenGL-based renderer with software rasterization. However, this approach proves inefficient when the GPU is absent and requires transparency. This limitation challenges companies like ASML, whose customers require efficient visualization tools. Currently, they are constrained to using remote servers without GPUs to render semiconductor structures.

The primary objective of this research is to explore the feasibility of a CPU-based raytracing renderer for semiconductor structures that can match and potentially surpass the capabilities of legacy OpenGL renderers. The focus is on replicating existing visualization features and achieving interactive frame rates on the CPU.

This thesis introduces new tracing optimization approaches by exploring the application of Oriented Bounding Boxes (OBB) and Axis-Aligned Bounding Boxes (AABB) within Bounding Volume Hierarchies (BVH). The structure is as follows: In chapter 2, we begin with an introduction to semiconductor structures, ray tracing, and its optimizations, focusing specifically on BVH and additional methods. chapter 3 outlines the requirements set by ASML Brion and presents the research questions for this project. chapter 4 details the main techniques behind our implementation, giving the core ideas behind the researched methods. chapter 5 presents the results and analysis of our implementation. Finally, in chapter 6, we answer the research questions and conclude the study.

1.1. ASML Brion

ASML Brion offers software solutions for low-fidelity modeling and visualization of semiconductor structures. Flatrace, a research demo, is designed to investigate the potential of replacing a CPU-based OpenGL rasterizer (see Figure 1.1) with a CPU-based raytracer. The primary objective of Flatrace is to explore new optimization methods for ray-tracing semiconductor structures. Flatrace implements a comprehensive framework for a functional ray tracer, which includes several critical components that collectively enhance its performance and capabilities:

1.1.1. Flatrace

ASML Brion provides software solutions for low-fidelity modeling and visualizing semiconductor structures. Flatrace is a research demo developed to explore the possibilities of replacing a CPU-based OpenGL rasterizer with a CPU-based raytracer. The primary objective of Flatrace is to explore new optimization methods for ray-tracing semiconductor structures. Flatrace implements a framework for a functional ray tracer. This framework includes several critical components that collectively enhance its performance and capabilities:

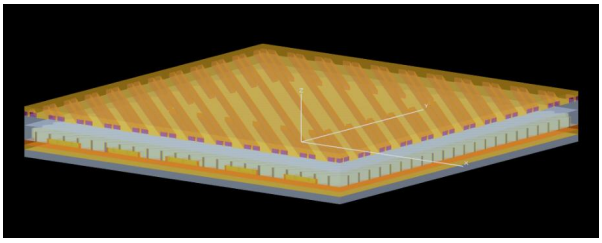


Figure 1.1: The current OpenGL rasterizer in ASML Brion.

AABB BVH with Binning

Flatrace employs an Axis-Aligned Bounding Box (AABB) Bounding Volume Hierarchy (BVH) using surface area heuristic (SAH) with binning [26] as acceleration structure for efficient ray tracing.

Parallel Processing and SIMD for AABB BVH

Flatrace leverages parallel processing and Single Instruction, Multiple Data (SIMD) instructions to accelerate the construction and traversal of the AABB BVH. By distributing the workload across multiple CPU cores and utilizing SIMD, Flatrace achieves substantial performance gains, enabling real-time rendering of complex scenes.

Transparency and Simple Shading

The ray tracer in Flatrace supports basic transparency effects based on ordered additive blending and shading effects. This feature allows for rendering semi-transparent materials and applying a simple Lambertian shading model [19].

2

Background & Related Work

This chapter explores the fundamentals of ray tracing on CPUs and GPUs, providing an overview of current ray tracing techniques. We will cover optimization techniques such as Axis-Aligned Bounding Boxes (AABB), Oriented Bounding Boxes (OBB), Bounding Volume Hierarchies (BVH), and Single Instruction, Multiple Data (SIMD). Additionally, we will discuss the current research gaps in OBB and how this research aims to address them.

2.1. Semiconductor Models

Flatrace is designed to render 3D semiconductor models rather than generic models. In 3D, semiconductor structures typically show boxy shapes, reflecting these semiconductors' layered and rectilinear nature. Their geometric simplicity is often defined by rectangular prisms and planar surfaces, making them distinct from more complex organic or freeform models.

Figure 2.1 illustrates this concept with a pattern that consists of uniformly spaced diagonal lines across two rectangular semiconductor layers. These diagonal lines represent different pathways within a semiconductor structure, such as metal interconnects or other functional materials. These diagonal lines' consistent, striped appearance emphasizes the precise, methodical design required in semiconductor models. Section 5.1 gives a more detailed analysis of the semiconductor models we used for evaluation.

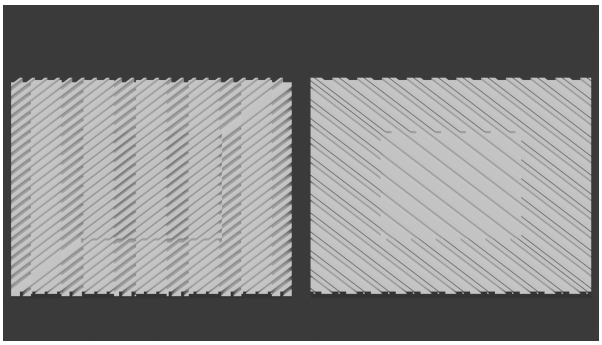


Figure 2.1: Layers of a semiconductor model.

2.2. Overview of Ray tracing

Ray tracing is a crucial component of many sophisticated image synthesis algorithms [15]. Partitioning a scene using bounding volume hierarchies (BVH) based on objects or primitives has become the state-of-the-art method for achieving fast ray tracing in recent years [17]. Various closed geometric shapes, including axis-aligned bounding boxes (AABBs) and oriented bounding boxes (OBBs) [13], have been used as bounding volumes in specialized applications.

2.2.1. GPU-Based Ray Tracing

GPU-based ray tracing leverages the parallel processing power of Graphics Processing Units (GPUs) to handle the computationally intensive task of ray tracing. With their thousands of smaller cores optimized for simultaneous processing, GPUs excel at handling large, repetitive tasks. This makes them particularly effective for ray tracing, which requires concurrently tracing multiple rays through a scene.

One of the primary advantages of GPU-based ray tracing is its speed. Modern GPUs can process vast numbers of rays in parallel, significantly reducing rendering times compared to CPUs [1]. Additionally, GPUs often include specialized hardware, such as NVIDIA's RT cores [21], which optimizes the BVH in GPU to minimize the number of ray-intersection tests, further boosting performance. This specialized hardware allows GPUs to handle heavy calculations efficiently, making them ideal for real-time rendering tasks in graphics-intensive applications like gaming and simulations.

2.2.2. CPU-Based Ray Tracing

CPU-based ray tracing has been a fundamental area of research for many years. Even with the advances in GPU-based ray tracing, many applications still use CPU-based ray tracing due to its flexibility in development. Most CPU-based ray tracers can only run offline because they usually try to render realistic, high-fidelity images. Since, in our project, we only render simplified, low-fidelity images, achieving an interactive frame rate is possible.

Recent advancements in CPU-based ray tracing have focused on optimizing performance and efficiency. Notable developments include the Embree framework for implementing advanced raytracing, which provides highly optimized ray tracing kernels for CPUs [8]. Embree utilizes modern CPU features such as Single Instruction and Multiple Data (SIMD) to accelerate ray intersection computations, offering significant performance improvements over traditional methods.

CPU-based ray tracing faces challenges, primarily due to the lower degree of parallelism than GPUs. This can impact performance for large-scale rendering tasks. However, the flexibility and ease of programming for complex scene management and dynamic data structures continue to make CPU-based ray tracing a valuable area of research and development where GPUs are not available [25].

2.2.3. Bounding Volume Hierarchies (BVH)

While direct intersection tests between rays and primitives are straightforward, they become impractical for complex geometric models due to their slow performance. Therefore, acceleration structures are essential for efficient ray intersection tests.

The Bounding Volume Hierarchy is a fundamental data structure used in ray tracing to accelerate intersection tests between rays and scene geometry [2]. BVH organizes primitives in a hierarchical tree structure, where each node represents a bounding volume that encapsulates its child nodes, and leaf nodes contain the actual geometry. Figure 2.2 shows how BVH divides the space and stores the primitives. This hierarchical arrangement allows for efficient culling of large portions of the scene that do not intersect with a given ray, thereby reducing the number of intersection tests needed to render the scene.

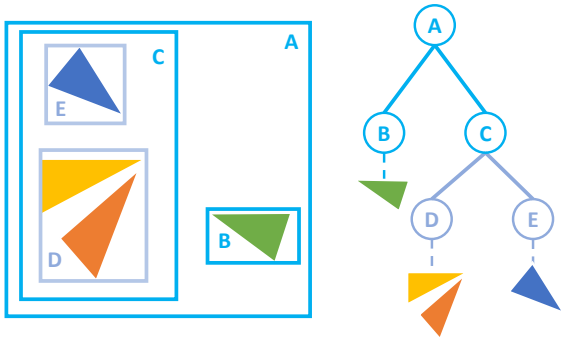


Figure 2.2: A BVH with AABB volumes (not the minimal volumes).

Construction

The BVH construction phase of a Bounding Volume Hierarchy (BVH) plays a vital role in improving the efficiency of ray tracing, primarily due to its significant influence on intersection tests. In the following, we will briefly overview several influential BVH construction strategies.

Surface Area Heuristic (SAH) Construction [26]: The SAH is one of the most popular BVH-building strategies (a BVH split method). It minimizes the expected traversal cost by carefully choosing where to split the nodes. Although SAH can be computationally expensive during construction, it often leads to highly efficient BVHs for ray tracing. Linear BVH (LBVH) [14] constructs the BVH more straightforwardly by sorting primitives based on their Morton codes (derived from their centroid positions). LBVH is generally faster to build than SAH-based methods but may result in less optimal BVHs. Agglomerative Clustering [9]: This approach builds the BVH by treating each primitive as a separate leaf node and then merging the closest nodes until the hierarchy is complete. While this can create high-quality BVHs, it is often more computationally expensive. Parallel BVH Construction [12]: With the rise of multi-core processors and GPUs, parallel BVH construction algorithms have been developed to take advantage of hardware parallelism. It aims to build BVHs faster by distributing the work across multiple threads or processors.

Traversal

BVH traversal aims to quickly identify the potential intersections between rays and scene primitives by traversing the BVH's hierarchical structure.

The traversal process begins at the root node and progresses through the hierarchy by evaluating whether a ray intersects the bounding volume of each node. If the ray intersects the node's bounding volume, the process continues recursively into the node's children. This process repeats until leaf nodes are reached, where the actual intersection tests with the scene primitives are performed [27].

An essential optimization in BVH traversal is using traversal algorithms designed to minimize the number of nodes visited. Techniques such as early termination, where the traversal stops as soon as the nearest intersection is found, can significantly reduce the computational load [10].

Parallel traversal methods have also been explored to take advantage of modern multi-core and many-core architectures [1]. These methods distribute the workload across multiple processing units, speeding up the traversal process. Another notable approach is using SIMD (Single Instruction, Multiple Data) operations to perform parallel intersection tests for multiple rays, further enhancing traversal efficiency [8].

2.2.4. Bounding Volumes

Bounding volumes in BVH simplify the representation of complex objects, enabling faster and more efficient computations by providing an easily manageable boundary. They also make object intersection checks easier, reducing the computational overhead associated with directly interacting with intricate models. Typical bounding volumes include spheres, axis-aligned bounding boxes (AABBs), and oriented bounding boxes (OBBs). In the following sections, we briefly talk about the three bounding volumes. Figure 2.3 depicts the difference of the bounding volumes within the same triangle.

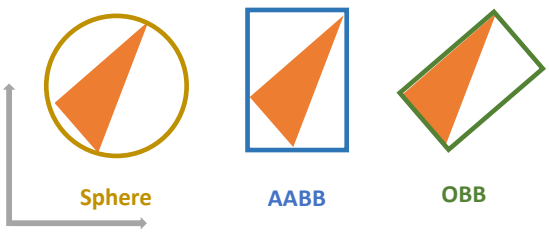


Figure 2.3: BVH volumes in 2D (Sphere vs. AABB vs. OBB).

2.2.5. Sphere Volume

Sphere volume is among the simplest and most commonly used bounding volumes in collision or intersection tests [7]. A center point and a radius define it and enclose an object within the smallest possible spherical boundary. Spheres offer computational simplicity and rotational invariance but may not tightly fit non-sphere shapes, leading to less efficient intersections.

2.2.6. Axis-Aligned Bounding Boxes (AABB)

Many BVH algorithms widely use the box-like AABB as a bounding volume [17]. It consistently aligns with the coordinate axes and streamlines intersection tests. This alignment simplifies the mathematical operations needed, resulting in improved performance for spatial queries and rendering processes in ray tracing. Nevertheless, AABB has certain limitations. A notable disadvantage is their limited flexibility in orientation, resulting in suboptimal bounding volumes for non-axis-aligned objects. This can lead to larger volumes that contain more empty space, which may result in an increased number of false positives in intersection tests and a decrease in overall efficiency in spatial partitioning. Despite the challenges, the balance between the simplicity and speed of AABB against their limitations is often considered advantageous, particularly in applications where rapid processing and efficiency are crucial.

2.2.7. Oriented Bounding Boxes (OBB)

Adopting OBBs in a BVH leads to a more precise bounding volume fit and a decrease in traversal steps [20]. Unlike AABBs, OBBs can rotate and take on any orientation in 3D space based on the geometry vertices. Due to its arbitrary orientation, an OBB has more attributes than an AABB. While OBBs offer a tighter fit around objects or primitives, enhancing spatial efficiency and reducing unnecessary traversals, this precision comes with complexities. The construction time for OBB tends to be longer than for AABB due to the complexity of determining the optimal orientation that minimizes the volume. Additionally, performing intersection tests with OBB is more computationally intensive. Despite these drawbacks, the precision and flexibility of OBB in accommodating complex shapes make them a valuable tool in specific applications.

Techniques such as Principal Component Analysis (PCA) or Ditetrahedron OBB (DiTO) can generate OBB for primitives.

Principal Component Analysis (PCA)

In general, Principal Component Analysis (PCA) is a statistical method that employs an orthogonal transformation to turn a set of observations of potentially correlated variables into a set of linearly uncorrelated variables known as principal components. The number of principal components is smaller or equal to that of the original variables. PCA is commonly used in data analysis and dimensionality reduction while preserving the data's most important features. PCA works by identifying the directions (principal components) along which the variance of the data is maximized. This is achieved by computing the eigenvectors and eigenvalues of the covariance matrix of the data. The eigenvectors (principal components) represent the directions of maximum variance, and the corresponding eigenvalues indicate the magnitude of this variance [22].

OBB is represented as a matrix when using the PCA method, and rays can be transformed into a unit space using this matrix. To generate OBBs, first compute the mean and covariance matrix of the mesh's vertex set P. Perform eigenvalue decomposition on this covariance matrix to obtain eigenvectors, which form the rotation matrix R defining the OBB's orientation. Transform the vertices P using R to align them with the principal components, and compute the minimum and maximum coordinates (V'_{min} and V'_{max}) of the transformed vertices. The center V'_{center} is the midpoint of these coordinates. Compute the scale S as the difference between V'_{max} and V'_{min}, and calculate the translation T using the mean P and V'_{center}. Combine S, R, and T to form the transformation matrix $M_{\text{OBB}} = TRS$, which can transform the ray into the AABB space defined as a unit cube centered at the origin, with $P_{\text{min}} = [-0.5, -0.5, -0.5]$ and $P_{\text{max}} = [0.5, 0.5, 0.5]$ [20].

Ditetrahedron OBB algorithm (DiTO)

The second method, DiTO, generates tighter fitting OBBs than PCA [13]. The DiTO algorithm processes a small, constant number of extremal vertices selected from input models to construct a representative shape called the ditetrahedron, from which the orientation of the bounding box can be derived efficiently.

Different instances of the algorithm, called DiTO-k, vary based on the number of selected vertices k. The ditetrahedron comprises two irregular tetrahedra connected along a shared interior side, known as the base triangle. This polyhedron has six faces, five vertices, and nine edges, with seven triangles, including the base. It is distinct from the triangular dipyramid, which consists of two pyramids with equal heights and a shared base. The ditetrahedron's triangles are expected to be characteristic of the orientation of a tight-fitting OBB for most input meshes.

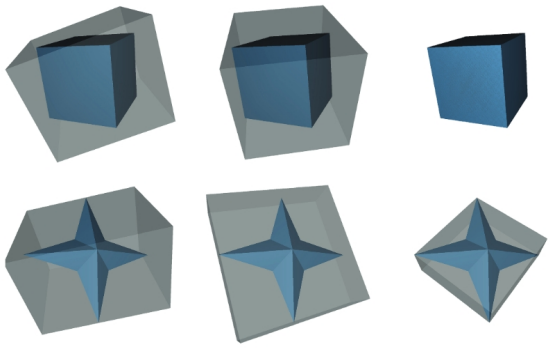


Figure 2.4: first column: AABB, second column: OBB(PCA), third column: OBB(DiTO) [13].

To illustrate, consider a randomly rotated cube with eight vertices and twelve triangles and a randomly rotated star shape with ten vertices and sixteen triangles as shown in Figure 2.4. For these shapes, the DiTO algorithm finds the minimum volume OBBs. In contrast, the PCA algorithm computes an excessively large OBB for the cube, with a volume two to four times larger than the minimum, depending on the cube's orientation. Similarly, PCA produces a loose-fitting OBB for the star shape.

2.3. Optimization Techniques for Ray Tracing

Ray tracing is a computationally intensive process that significantly benefits from various system-level optimization techniques besides BVH. These optimizations are essential for achieving real-time performance and handling complex scenes efficiently in CPU-based ray tracing. This section reviews critical optimization techniques applied in ray tracing, focusing on those relevant to CPU-based processes, including Single Instruction, Multiple Data (SIMD) and parallel processing.

2.3.1. Single Instruction, Multiple Data (SIMD)

SIMD is a powerful optimization technique used in CPU-based ray tracing to enhance performance [6]. SIMD allows a single instruction to process multiple data points simultaneously, which is particularly useful in ray tracing, where the same operation is often applied to multiple rays or ray bundles. Modern CPUs are equipped with SIMD extensions, such as SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions), which enable parallel processing of data in small batches [11]. By leveraging SIMD, ray tracing algorithms can achieve significant speedups in tasks such as ray-triangle intersection tests and shading calculations [28].

2.3.2. Parallel Processing

Parallel processing is another critical optimization technique that takes advantage of the multiple cores available in modern CPUs. In ray tracing, parallel processing can be implemented at various levels, from distributing rays across different cores to parallelizing the construction and traversal of acceleration structures like BVH. Frameworks such as Intel's Threading Building Blocks (TBB) [18] and OpenMP [5] are commonly used to facilitate parallelism in CPU-based ray tracing. By parallelizing workloads, these frameworks help to maximize CPU utilization and reduce rendering times.

2.3.3. Algorithmic Optimizations

Algorithmic optimizations play an important role in improving the efficiency of ray tracing. One of the most significant algorithmic techniques is using acceleration structures, such as BVH, which help minimize the number of intersection tests required during ray traversal, which we briefly introduce in subsection 2.2.3. These structures partition the 3D space in a way that allows for quick elimination of

large portions of the scene that do not intersect with a given ray. Additionally, techniques like adaptive sampling, where the number of rays per pixel is adjusted based on the scene's complexity, and importance sampling, which focuses rays on significant parts of the scene, enhance performance and image quality.

In conclusion, optimization techniques for ray tracing, particularly in CPU-based processes, encompass a wide range of strategies from SIMD and parallel processing to sophisticated algorithmic and data structure optimizations. These techniques are critical for achieving high-performance ray tracing capable of handling the demands of real-time applications and complex scene rendering.

2.4. Critique and Gap Identification

Recent advancements in ray tracing have predominantly concentrated on high-fidelity rendering powered by GPUs, leading to significant improvements in visual realism across applications such as gaming and simulation. However, this intense focus on GPU-based solutions has overshadowed research into CPU-based ray tracing, particularly for low-fidelity scenarios where computational resources are limited. This thesis aims to address this gap by developing ray tracing techniques to meet the needs of ASML's customers, optimizing performance in CPU-based raytracing.

3

Research Questions

3.1. Requirements

ASML provides the following requirements: 1) CPU ray tracing is required to render the whole geometry. 2) Interactive framerate. 3)Lambertian lighting model, no shadows, orthogonal camera, simple transparency. For ASML customers, the render time is more important than the BVH build time for ray tracing. As those are general requirements, specific methods are decidable. So, we have the following two main research questions.

3.2. Research Question: Optimal BVH Techniques

Which bounding volume or BVH split strategy is optimal for ray tracing within the context of our 3D semiconductor models? In chapter 2, we introduced several advanced techniques for accelerating ray tracing. This research seeks to identify the most effective method, or combination of methods, specifically for the unique structural characteristics of our 3D semiconductor models.

3.2.1. Hypothesis

Given the layered and rectilinear structure of semiconductors, combined with diagonal patterns that are not axis-aligned, it is hypothesized that an OBB will produce a tighter fit and utilize space more efficiently than an AABB. This hypothesis will be tested and evaluated in chapter 5.

3.3. Research Question: BVH Tracing Improvements

How can BVH traversal techniques be optimized to improve ray tracing performance in 3D semiconductor models? Given the orthogonal camera requirement and the different bounding volumes that could be used, the traversal and intersection within the BVH could be improved.

3.3.1. Hypothesis

Using an orthogonal camera, where all rays share the same direction, it is hypothesized that operations related to the ray direction can be precomputed and cached to accelerate traversal. For instance, caching the ray direction transformation for OBB traversal could reduce computational overhead and improve performance. This hypothesis will be tested and evaluated in chapter 5.

4

Methodology

This chapter focuses on the key techniques implemented in Flatrace, including those discussed in subsection 1.1.1, and a new method—clustering, will be discussed in subsection 4.2.1. The chapter begins by discussing the construction of BVHs and bounding volumes, then transitions to optimization strategies, mainly clustering BVH nodes using OBB volumes. An overview of Flatrace is illustrated in Figure 4.1. The ray tracer is developed in C++, with more detailed explanations provided in the subsequent sections.

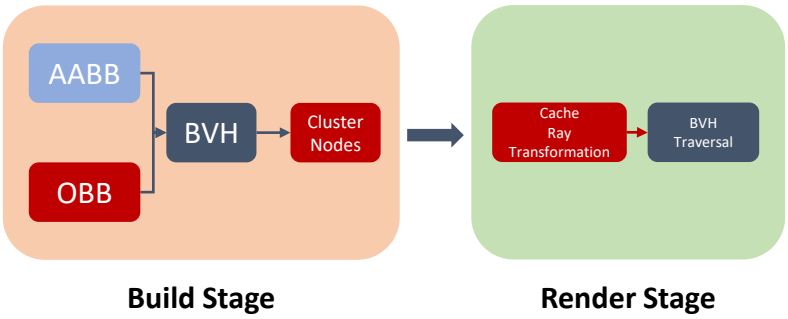


Figure 4.1: Flatrace has two main stages: Build Stage and Traversal Stage. All the OBB-specific operations are in red. In the build stage, BVH is built with AABB or OBB, and then the nodes in BVH are clustered for optimization(only for OBB). In the render stage, ray direction transformations are cached(only for OBB), and then the traversal phase will traverse the BVH and perform intersection tests between ray and bounding volumes or ray and triangles.

4.1. Acceleration Data Structure For Ray Tracing

We need to perform many ray-triangle intersections to render the geometry with ray tracing. Directly testing triangle intersections can be highly time-consuming. To speed up ray traversal and reduce the number of intersection tests, we utilize a Bounding Volume Hierarchy (BVH), the current state-of-the-art method for optimizing ray traversal [17]. A BVH uses bounding volumes, such as Axis-Aligned Bounding Boxes (AABB), to enclose triangles. However, using Oriented Bounding Boxes (OBB), which fit more tightly around triangles, can further enhance efficiency [13]. Therefore, we integrated OBB into Flatrace, which initially only supported AABB. We aim to determine the optimal BVH configuration for our semiconductor models by experimenting with different BVH construction methods, split strategies, and bounding volumes.

4.1.1. Bounding Volume Hierarchy

BVH organizes objects in a hierarchical tree structure, where each node represents a bounding volume that encapsulates its child nodes, and leaf nodes contain the actual triangles. This hierarchical

arrangement allows for efficient culling of large portions of the scene that do not intersect with a given ray, thereby reducing the number of intersection tests needed to render the scene.

In Flatrace, the BVH construction involves several key steps:

- **Node Splitting:** The scene is recursively divided into smaller sub-bounding volumes using the splitting methods.
- **Bounding Volume Calculation:** Appropriate bounding volumes will be calculated for each node while a node is being split.
- **clustering:** Optionally, similar nodes are grouped for BVHs with OBB volumes(clustering will be introduced in subsection 4.2.1).

4.1.2. BVH Construction

The construction process of a BVH directly influences the efficiency and effectiveness of ray traversal. This process involves organizing scene geometries into bounding volumes. We can create a BVH that minimizes overall computational costs by carefully selecting split strategies.

Node Splitting

How the scene's space is partitioned is crucial for constructing an optimal Bounding Volume Hierarchy (BVH). In Flatrace, we have implemented two splitting methods, which are illustrated in Figure 4.2 and subsequently evaluated in Table 5.2 to determine which approach yields the most efficient BVH structure.

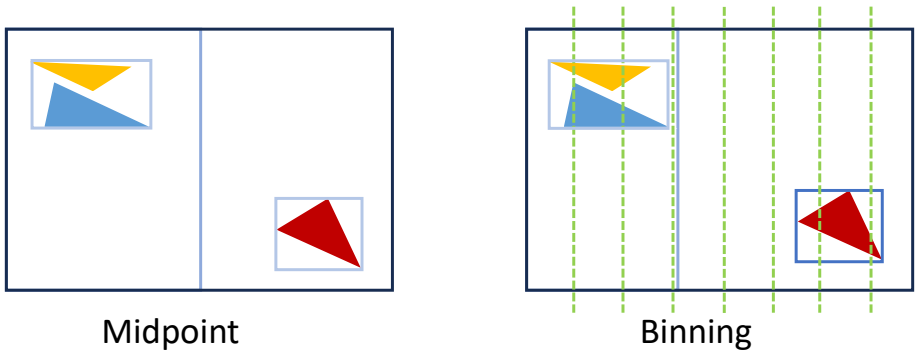


Figure 4.2: Split methods in 2D, we can see that binning creates fewer and tighter nodes(the green dash lines are here to represent bins).

The first splitting method is the Midpoint Split [29], which divides the node space at the midpoint along one of the BVH node's axes. Specifically, the algorithm selects the axis with the largest spatial extent and splits the geometries at the midpoint point or the median of the triangle position along that axis. While the Midpoint Split ensures a balanced spatial partitioning, it may not always result in the tightest possible bounding volumes if the triangles are not evenly located in the node, potentially leading to increased traversal and intersection costs.

The second splitting method is the Binning Split [26], which leverages the SAH cost to determine the optimal split position for a BVH node. Instead of computing the SAH cost for every possible split, binning involves dividing the node space into a predefined number of bins along each axis and evaluating the SAH cost for each possible split within these bins, reducing computation time. The split that minimizes the overall SAH cost is then selected, resulting in a more balanced and efficient BVH. This method typically produces tighter-fitting bounding volumes and reduces the number of intersection tests during traversal, enhancing overall performance. However, the Binning Split is more computationally intensive than the Midpoint Split.

Bounding Volume Calculation

After splitting the node into two spaces, the next crucial step in constructing the BVH is calculating the bounding volumes of the newly created child nodes based on the geometry in the child nodes. The choice of bounding volume plays a significant role in determining the BVH's efficiency, as it directly influences the tightness of the fit around the geometry, which affects the speed of ray traversal.

In Flatrace, we have both AABB and OBB as options for bounding volume calculations, allowing us to compare their performance within the context of our 3D semiconductor models. In Figure 2.3, the difference between AABB and OBB are illustrated in 2D. We aim to balance computational cost and traversal efficiency by experimenting with both bounding volumes. The evaluation of these bounding volumes, in conjunction with different split methods, will be discussed in chapter 5, where we analyze their impact on the overall performance of the BVH and determine the most effective approach for our specific application.

AABB is the simplest and most commonly used bounding volume [17]. An AABB is defined by the minimum and maximum points along each axis, creating a box aligned with the coordinate axes. The primary advantage of AABBs is their computational simplicity; they are easy to compute and require minimal processing during ray traversal since intersection tests are straightforward and fast. However, the simplicity of AABBs can also be a limitation. Because AABBs are always aligned with the axes, they may not always provide the tightest possible fit around the geometry, especially when dealing with objects that are not axis-aligned. This can lead to larger bounding volumes and more intersection tests during traversal, reducing overall efficiency.

On the other hand, an OBB offers a more flexible and tighter fit by aligning the bounding box with the object's orientation rather than the coordinate axes [13]. A center point, a set of orientation axes, the extents along these axes, and an inverse matrix define an OBB. The flexibility of OBBs allows them to conform closely to the geometry, especially in cases where objects are rotated relative to the coordinate system. This tighter fit typically reduces unnecessary intersection tests during traversal, improving efficiency. However, the calculation of OBBs is more complex and computationally expensive than AABBs. Additionally, OBB-ray intersection tests are more complicated, which could increase the traversal cost. The OBB traversal will be discussed in subsection 4.1.5.

4.1.3. OBB Generation using DiTO

Unlike AABBs, OBBs are more challenging to generate. To construct the OBBs, we employ the ditetrahedron OBB algorithm (DiTO) [13], since it produces a tighter bounding box than PCA as shown in Figure 2.4. The DiTO algorithm was developed for point clouds, polygon meshes, or polygon soups without requiring an initial convex hull generation process. The algorithm is founded on processing a limited and unchanging quantity of distinctive vertices from the input model. The chosen points are subsequently utilized to construct a representative geometric figure known as a double tetrahedron, or ditetrahedron. Determining an optimal alignment for a tight-fitting bounding box can be efficiently achieved by utilizing the edges of this particular shape. The DiTO algorithm can be summarized in four steps which can be seen in Figure 4.3.

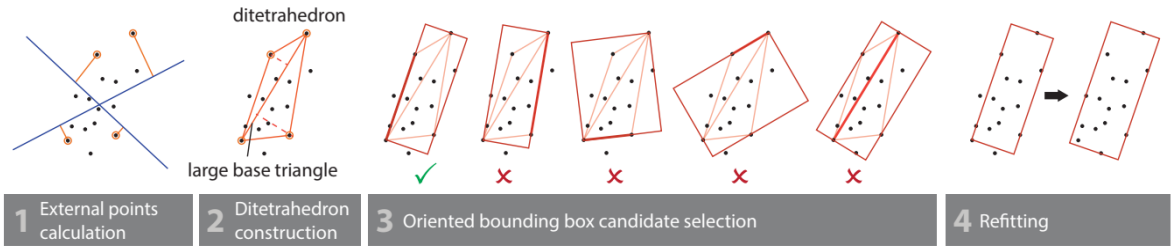


Figure 4.3: DiTO algorithm in 2D[24].

Extremal Vertices Calculation

The algorithm begins by selecting a small, constant number of extremal vertices from the input geometry. Extremal vertices are farthest along specific directions or axes(manually selected). The number of vertices chosen, denoted by k in DiTO-k, affects both the running time and the quality of the resulting

OBB. Based on the experiment by the DiTO authors [13], we choose $k = 14$ in our implementation, which means seven pairs of extremal vertices will be selected, and seven candidate axes will be used.

Ditetrahedron Construction

The second step is to use these extremal vertices to construct a shape called the ditetrahedron. This shape is not a regular geometric figure but a polyhedron formed by two irregular tetrahedra sharing a base triangle. The ditetrahedron has six faces, five vertices, and nine edges. It includes seven triangles, counting the interior base triangle, formed from 3 of the selected vertices. It uses the farthest vertices as one edge of the triangle and the farthest vertex to that edge as the third triangle vertex. Using this base triangle, the algorithm chooses the opposite farthest vertices to create two joint tetrahedra to form a diterahedron.

OBB Candidate Selection

Next, the algorithm looks at all seven sides of the tetrahedron, checking each triangle-aligned local reference frame as a possible axis for an OBB that fits well and has an axis-aligned box. The goal is to determine the configuration with the smallest surface area.

OBB Refitting

The previous phase has established the alignment of a satisfactory OBB, which needs to be readjusted to encompass all points in the geometry. This is necessary since the orientation of the OBB's faces is determined by the facets of the ditetrahedron, rather than the minimum and maximum projections along the present local reference frame's axes. The OBB extents are computed by repeatedly identifying the smallest and largest projections of all points in P onto the local OBB axis.

4.1.4. BVH Traversal

The BVH traversal is an essential phase of our rendering process and is designed to efficiently identify potential intersections between rays and geometries within the scene. This method utilizes a stack-based approach to navigate through the BVH nodes.

Initialization

The traversal begins with the initialization of a stack to track the BVH nodes that need to be visited. The root node of the BVH is the starting point and is initially pushed onto this stack. This setup ensures that the traversal starts from the top of the hierarchy, allowing the algorithm to explore the entire tree structure if needed.

Traversal Loop

The core of the traversal process is a loop that continues until the stack is empty. During each iteration, the node at the top of the stack is popped for processing. This node is then analyzed to determine whether it is a leaf or an internal node. If it is a leaf node, the algorithm checks for intersections between the ray and the triangles within that node. This step is crucial for identifying actual intersections.

Internal Node Processing

When an internal node is encountered, the algorithm retrieves its left and right child nodes. It then checks for intersections with these child nodes (intersections with the bounding volumes), determining the distance to the intersection points. The nodes are sorted based on their intersection distances to ensure that the closest nodes are processed first. The child nodes are pushed onto the stack for further processing if a valid intersection is found. This approach ensures that all potential intersections are explored efficiently.

Bounding Volume Intersection

Intersection with AABB is different from OBB. AABB offers better computational efficiency because its simple structure makes intersection tests straightforward and fast. In contrast, OBB fits objects more tightly with arbitrary orientations, which will be discussed in more detail in subsection 4.1.5.

4.1.5. OBB Traversal

To simplify intersecting OBBs, we transform rays from OBB space into unit AABB space and then perform intersection tests between the transformed ray and unit AABB. When constructing the OBB, we determine the transformation matrix that converts a unit AABB (with dimensions from -0.5 to 0.5) into OBB space. Each OBB then stores the inverse of this matrix. Then, this inverse matrix can transform the ray into unit AABB space during the actual traversal, facilitating easy and quick intersection tests. For every node traversal, both child nodes of the node are tested, as detailed in subsection 4.1.4. Consequently, the ray origin and ray direction are transformed twice, once for each child node. Therefore, an internal node intersection cost for OBB involves four matrix operations and two unit AABB intersection tests.

4.2. Clustering BVH Nodes with OBB

Because the semiconductor models have many repeating patterns, we can simplify the traversal computations by grouping similar BVH nodes (only leaf nodes for slower construction time) and creating a representative OBB for a cluster to reduce the OBB ray transformation calculation. This optimization process involves several key steps, which will be discussed in the following sections, also presented in Figure 4.4.

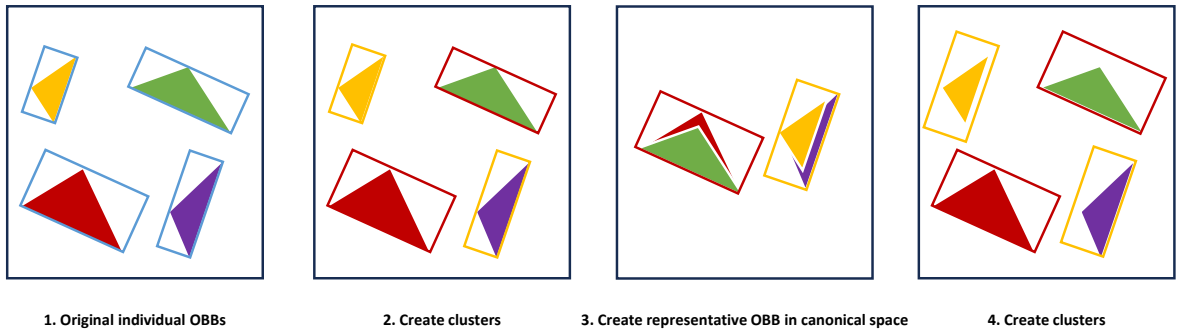


Figure 4.4: Process of clustering. The blue bounding box is the original OBB, and red and yellow are two clusters and the representative OBBs.

4.2.1. Group Similar Nodes

First, we utilize the midpoints and orientations of the OBBs as feature inputs for the clustering algorithm. This allows us to create clusters of the leaf nodes without considering the spatial distance. We use K-means clustering for the clustering algorithm.

K-means Clustering

K-means clustering is a method used to group data points into a specified number of clusters based on their features [16]. In the context of BVH optimization, we use the midpoints and directions of all leaf nodes' OBBs as the data points. The K-means algorithm starts by initializing a set number of cluster centroids. Each leaf node is assigned to the nearest centroid based on the Euclidean distance between their midpoints and directions. Once all nodes are assigned to clusters, the centroids are recalculated as the mean of all points within each cluster. This assignment and centroid recalculation process iterates until the centroids stabilize, meaning they no longer change significantly with further iterations. The final clusters group similar leaf nodes together, allowing for the creation of a representative OBB for each cluster. We use K-means by MLpack [4] to enable fast CPU clustering in our implementation.

4.2.2. Generate Representative OBB

After forming clusters, we generate a representative OBB for each cluster. This representative OBB is designed to encompass all triangles within the node by aligning them in a canonical space, where the triangles are translated to a common reference point. This alignment allows the representative OBB to fit arbitrary nodes in the cluster.

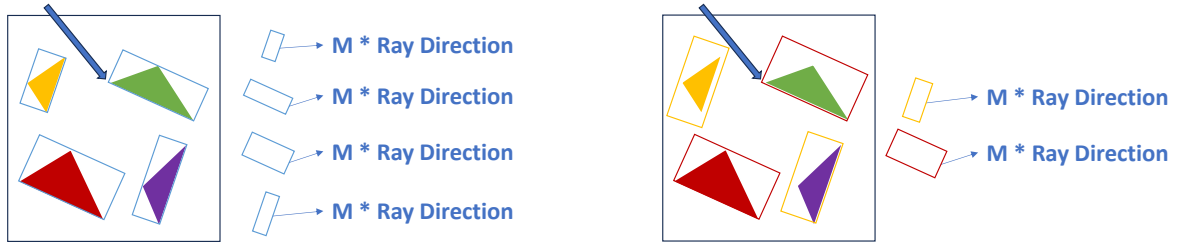
4.2.3. Replace Individual OBBs in Clusters

Next, we replace the OBB in each node within the clusters with its corresponding representative OBB. During the traversal stage, these representative OBBs are used instead of the individual node's OBBs. By clustering the BVH nodes, we do not need to transform the rays for each OBB but only for the representative OBBs. As a result, we can see in Figure 4.4 in step 4 that we replace the original OBBs with the representative OBB, larger than the original OBBs. The following section will explain how and when this larger representative OBB will help the tracing performance.

4.2.4. Transformed Ray Direction Caching

In our ray tracer, as explained in chapter 1, we use an orthogonal camera. An essential aspect of this setup is that all rays have the same direction due to the camera's orthogonality. As highlighted in subsection 4.1.5, it is necessary to transform rays into the unit AABB space, including transforming their origin positions and directions, which is computation-consuming.

Since all rays share the same direction, and we have generated representative OBBs as outlined in subsection 4.2.3, we can optimize the traversal phase. As shown in Figure 4.5, without representative OBBs, a matrix transformation of the ray direction is required for each node during traversal. However, with representative OBBs, this transformation only needs to be performed once per representative OBB before node traversal begins, with the result used during traversal. This reduces the number of matrix operations in internal node intersection to two.



(a) Without clustering, we need to perform matrix calculation for each individual node.

(b) With clustering, we only need to perform matrix calculation for representative OBBs.

Figure 4.5: Comparing the number of matrix operations during the internal node traversal.

Given the nature of clustering and the caching of transformed ray directions, we anticipate a trade-off depending on the number of clusters used. The representative OBBs will be less accurate with fewer clusters, leading to fewer ray direction transformations but more intersection tests. Conversely, with more clusters, the representative OBBs will be more precise, resulting in more ray direction transformations but fewer intersection tests. Therefore, finding the optimal balance between the number of clusters is crucial to maximize efficiency and the results are shown in section 5.5.

4.2.5. Types of BVH

Depending on the bounding volume used for BVH node splitting and the splitting method, the BVH tree can take on different shapes. Three types of BVH are implemented in Flatrace to explore the optimal approach. Each can have various bounding volume or split method configurations.

The first type, AABB-based BVH, incorporates both AABB and OBB volumes within its structure. Initially, the scene is split based on AABB volumes and with the binning split method [26]. OBB and AABB volumes are constructed utilizing the same geometric information in the BVH nodes, ensuring that the OBBs encapsulate geometry identical to that of the AABBs. Even though the AABB and OBB volumes are constructed for the AABB BVH, only one is enabled for tracing. This BVH works mainly as a baseline for Flatrace render results.

The second type is OBB-based BVH, which exclusively incorporates OBB volumes within its structure. The scene split in this BVH is based solely on the OBB volumes. Midpoint/Median splitting and Binning split methods are employed for this type. Given the distinct characteristics of OBBs in comparison to AABBs, this method yields BVH shapes that differ from those generated by the AABB-based approach. By concentrating solely on OBBs, this BVH aims to enhance the tightness and effectiveness of the bounding volumes throughout the hierarchy.

The third type is the hybrid BVH, which is fundamentally AABB-based BVH. The Hybrid BVH is implemented assuming it can avoid the overhead introduced by OBB traversal when the OBB is aligned with the world axis and an AABB with faster tracing is a better choice. Unlike the AABB-based BVH, it uses both bounding volumes for tracing. During the splitting process, both volume types are compared based on SAH cost in each node to determine which volume should be used for tracing the node, and a flag will be set to inform the traversal phase which bounding volume to use. This approach aims to leverage the advantages of AABB and OBB volumes, optimizing for the most efficient bounding volume at each node.

4.3. SIMD And Parallel Processing

We employ SIMD and parallel processing with TBB (Threading Building Blocks) [18] to further optimize the tracing process. SIMD enables the simultaneous processing of multiple rays using a single instruction, thereby significantly accelerating tracing operations. In our implementation, we leverage SSE [3] to perform 4x4 ray tracing, which bundles 16 rays together to conduct intersection tests with the bounding volume concurrently.

In addition to SIMD, TBB is employed to parallelize the workload across multiple CPU cores. TBB provides a high-level abstraction for parallel programming, enabling efficient task scheduling and load balancing. By dividing the tracing tasks of rays into smaller bundles and distributing them across available cores, TBB ensures that computational resources are utilized effectively, reducing the overall render time.

5 Results

In this chapter, we present and analyze the results of our experiments to evaluate the efficacy of the various methods discussed in chapter 4. First, we examine the geometry patterns of the semiconductor models to identify each model's unique characteristics. The BVHs' heatmap visualization highlights the differing efficiencies of the bounding volumes. Next, we assess the impact of varying bin sizes on the performance of AABB and OBB BVHs in the binning split method. Following this, we evaluate the efficiency of AABB and OBB BVHs and the bounding volumes across various metrics. We then demonstrate how implementing clustering methods enhances performance and examine the performance of hybrid BVH. Subsequently, we show the improvement by SIMD. Finally, we evaluate render time cost during different phases of tracing. All tests were conducted on an AMD Ryzen 9 7940HS CPU with a render resolution of 1280x720 and 1 sample per pixel (ssp).

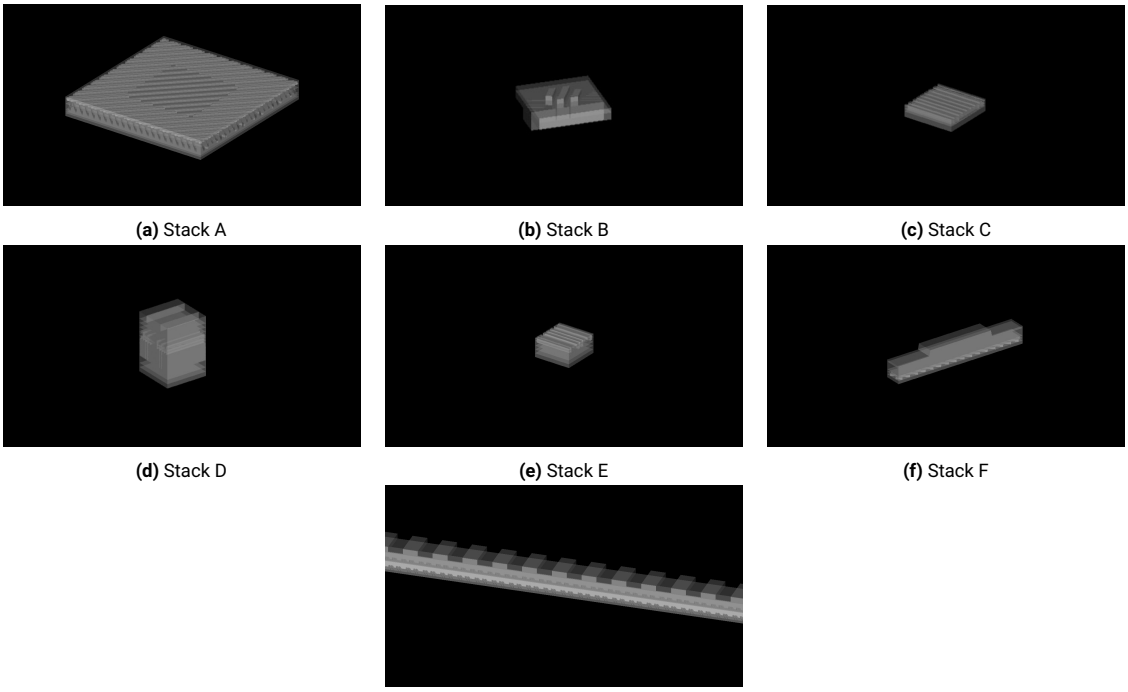
5.1. Semiconductor Models

The semiconductor models used for the evaluation are shown in Figure 5.1, and Table 5.1 lists the triangle counts of each model. Stack A stands out with its large, flat, rectangular prism adorned with uniformly spaced diagonal lines. Similarly, Stack C and Stack E emphasize simpler geometry and lower triangle counts; Stack C features minor, evenly spaced diagonal lines, while Stack E has a compact, cube-like form consisting of several horizontal layers.

In contrast, Stack B and Stack D introduce more complexity. Stack B features multiple layers and distinct raised elements on the top layer. Meanwhile, Stack D presents a vertically oriented, boxy structure composed of multiple stacked layers; despite its compact appearance, Stack D features a dense arrangement with many overlapping axis-aligned geometry and a high triangle count. Stack F and Stack G further illustrate the diversity in semiconductor architecture. Stack F displays an extended rectangular prism with a stepped profile. At the same time, Stack G showcases a long, continuous structure with a repeating pattern of rectangular elements, resulting in the largest triangle count. In summary, they all show boxy shapes with patterns across layers.

Model	**Triangle Count**
Stack A	42252
Stack B	1548
Stack C	5616
Stack D	515824
Stack E	6604
Stack F	6912
Stack G	653544

Table 5.1: Semiconductor Models



(a) Stack A **(b)** Stack B **(c)** Stack C

(d) Stack D **(e)** Stack E **(f)** Stack F

(g) Stack G

Figure 5.1: All the testing stack models.

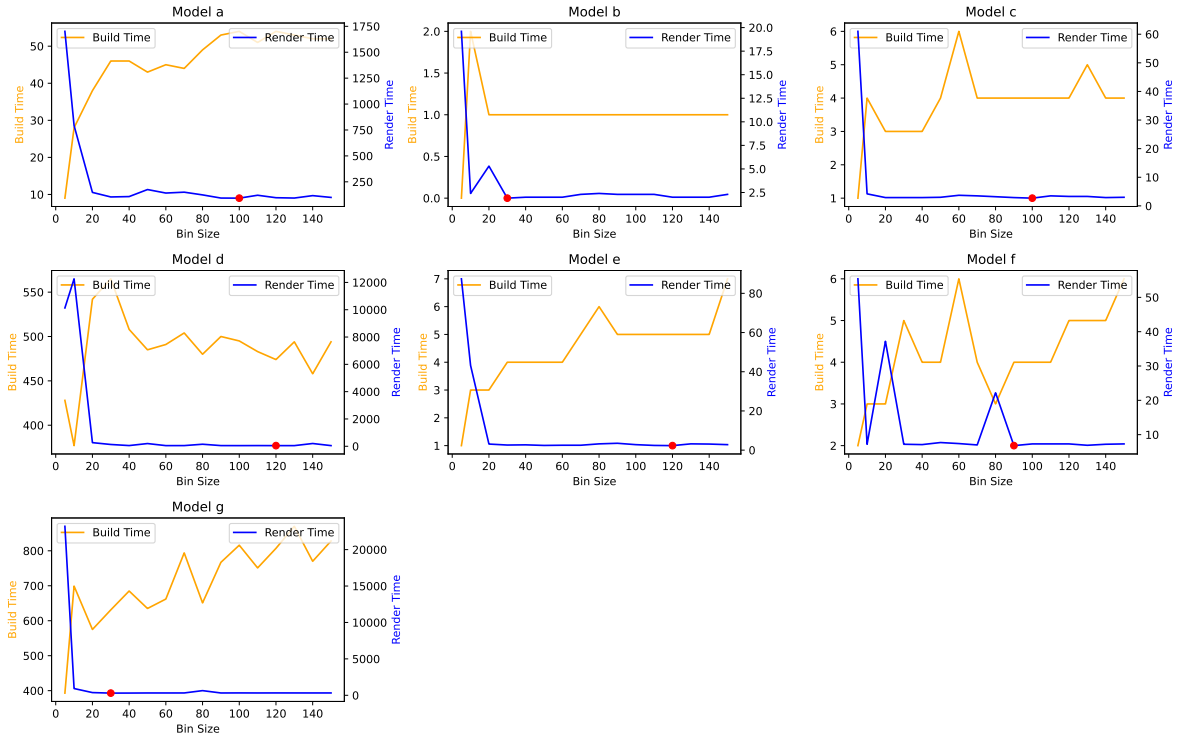
5.2. Bin Size of Binning

In subsection 4.1.2, we discussed the binning split method, where the node space is divided into a predetermined number of bins along each axis. The process then evaluates the SAH cost for every potential split within these bins. Therefore, it is vital to determine the number of bins when using this method to construct BVHs.

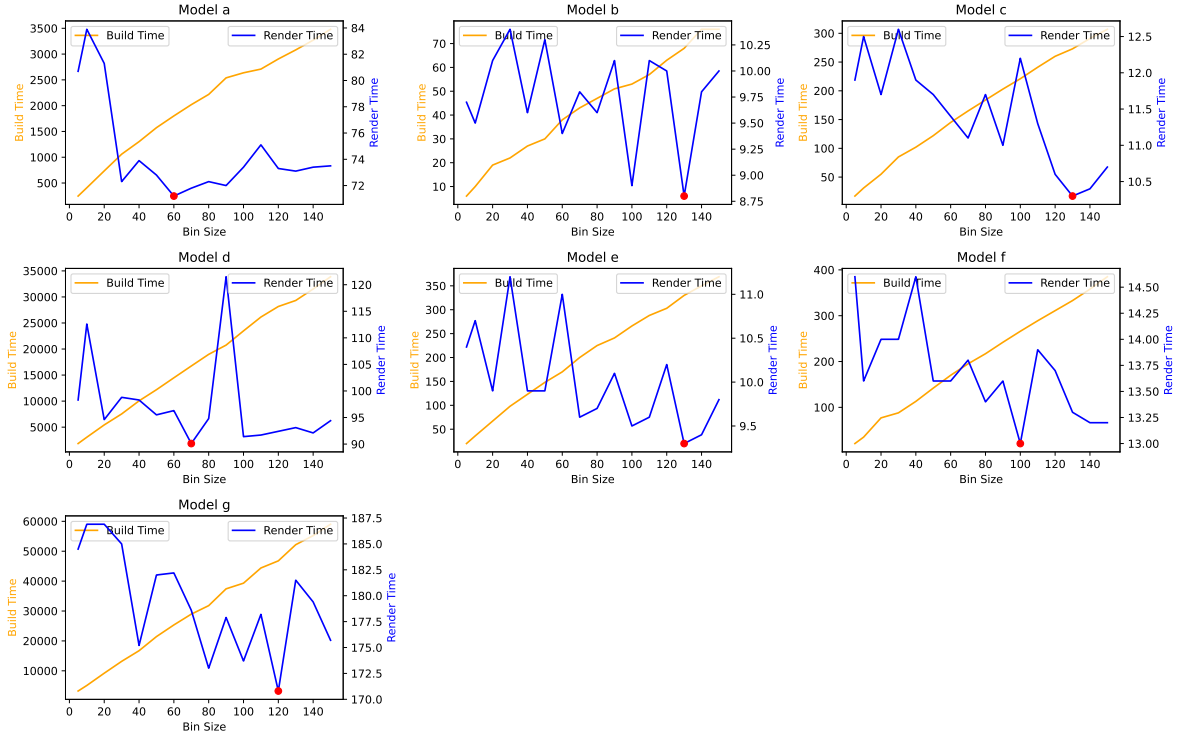
Both Figure 5.2a and Figure 5.2b illustrate how different bin sizes (number of bins) impact the performance of AABB BVH (tracing OBB) and OBB BVH, respectively. For both types of BVH, we observe a consistent trend where build time and render time are significantly influenced by the chosen bin size. Smaller bin sizes generally result in lower build times but lead to higher render times due to less optimal splits. Conversely, larger bin sizes significantly increase build time while improving render performance.

In the case of AABB BVH, a bin size of 100 was found to offer a balanced compromise across all models. This number maintains relatively low build times while minimizing render times, making it an ideal choice for consistency in performance evaluation across different models. Similarly, for OBB BVH, a bin size 32 emerged as the most balanced choice. It provides a reasonable trade-off between build and good render times, ensuring efficient performance across various scenarios while avoiding extreme values in either metric.

Therefore, even though build time is not the most critical metric, balancing build time and render time is essential. With this in mind, we selected 100 as the standard bin size for AABB BVH and 32 as the standard bin size for OBB BVH in all subsequent evaluations. These selections ensure fair and comparable results across different models and conditions, balancing computational efficiency with rendering quality.



(a) AABB binning bin size

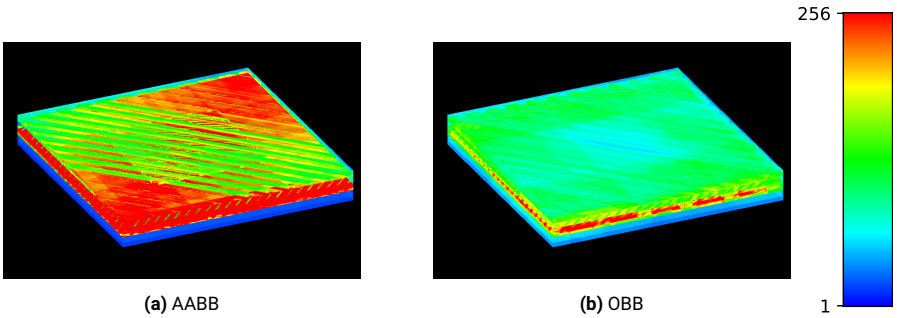


(b) OBB binning bin size

Figure 5.2: Both AABB BVH and OBB BVH can utilize the binning split method.

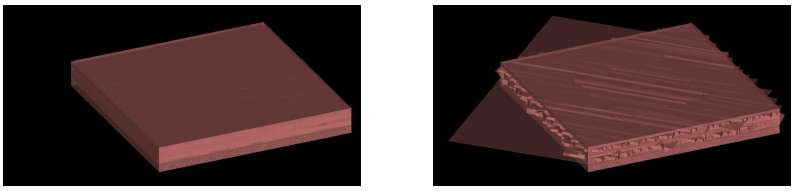
5.3. AABB Volume vs. OBB Volume

In Figure 5.3, we visualize the number of bounding volume intersections during traversal. The heatmap shows that the OBB BVH has more green areas than the AABB BVH, indicating fewer intersections before hitting the geometry. Although OBBs are generally tighter, the bottom layer shows fewer intersections for AABBs. This is likely because the bottom layer's structure is more axis-aligned, and OBBs in higher layers might occlude lower layers from certain angles. Figure 5.4 further illustrates that some OBB volumes in the top layers can occlude the base layers due to their rotation, potentially causing higher intersection counts for the bottom layers. This demonstrates that while OBBs typically reduce the number of intersections during traversal, the specific structure and orientation of the layers can influence the intersection count.



(a) AABB **(b)** OBB

Figure 5.3: Comparison of AABB/OBB BVH intersection numbers. From 1 to 256 (blue-green-yellow-red) as shown on the right gradient map.



(a) AABB volumes **(b)** OBB volumes

Figure 5.4: The AABB visualization exhibits a high degree of alignment, whereas the OBB displays a more varied orientation.

5.4. BVH Performance

As detailed in subsection 4.2.5, Flatrace utilizes three types of BVH. In this section, we will analyze AABB and OBB BVHs, while the evaluation of hybrid BVH will be discussed later in section 5.6. Table 5.2 presents performance metrics across these BVH configurations, examining various combinations of split methods and bounding volumes to identify the most effective BVH structures for our rendering pipeline.

Key metrics, including SAH cost, balance factor, build time, and render time, are employed to assess the efficiency and performance of various BVH structures. These metrics are instrumental in evaluating the impact of different BVH construction methods on rendering performance.

5.4.1. SAH Cost

The SAH Cost metric measures the computational cost of using the SAH during BVH construction. Lower SAH costs indicate more efficient tree structures, leading to faster traversal times and improved rendering performance. The data in Table 5.2 demonstrates various SAH costs with different configurations. The OBB methods significantly reduce SAH costs in most models, especially in the relatively complex Models A and G. OBB BVH with SAH outperforms AABB BVH multiple times. However, although the results are close for smaller models like C and E, the AABB method provides better outcomes. In the complex Model D (in which structures are mainly aligned with axes and have many overlaps), the OBB BVH with SAH has a higher SAH Cost than the AABB BVH.

OBBs generally provide a tighter fit around objects compared to AABBs. This tighter fit reduces the volume of empty space within each bounding box, leading to fewer unnecessary intersection tests

during ray tracing. The improved efficiency is most noticeable in complex models, where non-axis-aligned structure shapes benefit significantly from OBB's adaptive orientation. Conversely, smaller or less complex models, like C, E, and F, show less dramatic improvements or even better performance with AABB. This is because, in simpler geometries where most shapes are already aligned with the axes, AABBs are highly effective. In these cases, the models are small, and the formed structures are mostly aligned with the axes, making the axis-aligned bounding volume a better fit. When triangles within a BVH node are long and thin, OBBs may rotate away from the axis-aligned orientation, leaving empty spaces to accommodate the elongated triangles, causing high SAH costs. This means the SAH cost of OBBs versus AABBs depends significantly on the specific geometry and triangle rotations within the BVH nodes. Still, OBBs generally reduce the empty spaces of complex models.

5.4.2. Balance Factor

The balance factor, represented by the standard deviation, evaluates the distribution of nodes within the BVH. A well-balanced BVH ensures that traversal paths are evenly distributed, preventing performance bottlenecks and evenly spreading the computational load across the tree.

Table 5.2 also indicates a range of balance factors across different models and BVH configurations in column **STD**. The OBB BVH with the Midpoint or Median scores performs the best in most models. Model G, however, exhibits an exceptionally high balance factor with the AABB BVH, suggesting significant disparities in node distribution. In contrast, other models display closer scores. As seen in Model G, high balance factors can indicate an uneven distribution of nodes, where certain paths within the BVH may be significantly deeper or shallower than others. This uneven distribution can lead to inefficiencies during traversal, as some paths may consistently require more computation time, resulting in potential bottlenecks. Overlaps in Model G likely cause this disparity.

The OBB BVH with Midpoint and Median methods usually produce the best STD, primarily because these two methods divide the space evenly. Stack models often contain many overlapping triangles, which leads to difficulties in finding optimal split positions during construction for both methods. As a result, Midpoint and Median methods may place many triangles in leaf nodes, causing low tree depth. Despite achieving the best STD, the SAH costs remain high for both methods, highlighting the necessity for multiple metrics.

5.4.3. Build Time

The choice between AABB and OBB trees and the split method (SAH, Midpoint, or Median) greatly influences build times. According to Table 5.2, using the SAH Binning split method, AABB BVH builds significantly faster than OBB BVH. This is because OBB BVH, particularly with the SAH split method, typically requires more time due to the complexity of optimizing orientations (generating OBBs) and calculating the best splits based on the SAH. In contrast, AABB trees with the SAH split method build more quickly due to the simplicity of generating AABBs. Additionally, the Midpoint and Median split methods consistently yield the fastest results because of their simplicity. The 0s in smaller models mean it took less than 1 ms to build.

Moreover, the complexity of the model plays a crucial role. More complex models, such as Stack D and G, require longer to process due to the increased number of primitives and the complexity of their spatial distribution, which complicates BVH construction.

5.4.4. Render Time

Render time is a critical metric for evaluating the performance of BVH structures as it directly reflects their efficiency. Lower render times indicate a more effective BVH capable of quickly computing ray intersections with minimal computational overhead. Table 5.2 highlights significant variations in render times across different models and configurations. Notably, the BVH with OBB volumes achieves the best render times in complex models (e.g., Model A and Model G), underscoring its effectiveness in minimizing traversal steps and intersection tests. This efficiency arises from the tighter bounding volumes of OBBs, which more accurately encapsulate the geometry and reduce unnecessary ray intersections. However, the AABB BVH (with AABB volumes) performs better for Model D. Most geometry in Model D is aligned with the world axis, making the OBB volumes similar in shape and surface area to the AABB volumes. Consequently, the OBB volumes do not provide a significant advantage and even introduce additional intersection overhead compared to the more straightforward AABB intersection.

Conversely, for simpler models such as Model B and E, AABB volumes render faster than OBB volumes. This indicates that for geometries with fewer primitives, the advantages of using OBBs may not compensate for their additional computational overhead during construction. Due to its faster intersection tests, an AABB volume can offer comparable or even better performance in these scenarios.

BVH	SAH Cost	STD	B-Time	R-Time
AABB	5314.6	6.9	41	86.7
AABB (OBB)	1426.7	6.9	65	94.3
OBB (Mid)	6641.1	2.1	**25**	376.0
OBB (Median)	11312.6	**0.7**	36	836.4
OBB (Binning)	**527.6**	2.1	594	**72.2**

(a) Model a

BVH	SAH Cost	STD	B-Time	R-Time
AABB	12.4	2.1	**0**	**1.9**
AABB (OBB)	**12.2**	2.1	**0**	4.3
OBB (Mid)	25.6	1.5	**0**	10.7
OBB (Median)	41.0	**1.0**	**0**	11.7
OBB (Binning)	**12.2**	2.3	12	8.9

(b) Model b

BVH	SAH Cost	STD	B-Time	R-Time
AABB	**43.6**	1.8	**1**	**3.1**
AABB (OBB)	50.8	1.8	3	6.6
OBB (Mid)	99.3	1.3	**1**	18.0
OBB (Median)	289.0	**1.1**	**1**	27.5
OBB (Binning)	50.8	1.6	39	10.7

(c) Model c

BVH	SAH Cost	STD	B-Time	R-Time
AABB	**7078.0**	2.0	303	**44.1**
AABB (OBB)	7927.4	2.0	478	99.6
OBB (Mid)	58675.7	2.3	**163**	13459.3
OBB (Median)	44497.6	2.0	248	7591.3
OBB (Binning)	7849.8	**1.7**	4388	94.5

(d) Model d

BVH	SAH Cost	STD	B-Time	R-Time
AABB	**47.1**	2.2	2	**2.4**
AABB (OBB)	55.0	2.2	3	5.3
OBB (Mid)	785.6	**0.8**	**0**	69.7
OBB (Median)	997.5	1.2	**0**	95.0
OBB (Binning)	55.0	2.0	49	9.9

(e) Model e

BVH	SAH Cost	STD	B-Time	R-Time
AABB	168.2	1.8	2	**6.9**
AABB (OBB)	135.6	1.8	3	8.7
OBB (Mid)	1836.5	**0.9**	1	124.3
OBB (Median)	2363.3	1.8	**0**	160.4
OBB (Binning)	**121.0**	1.9	47	13.4

(f) Model f

BVH	SAH Cost	STD	B-Time	R-Time
AABB	26279.5	18.3	426	328.2
AABB (OBB)	6780.0	18.3	785	213.8
OBB (Mid)	510607.0	1.8	**324**	3610.8
OBB (Median)	971066.0	**1.5**	339	26035.4
OBB (Binning)	**5969.6**	1.7	7577	**194.0**

(g) Model g

Table 5.2: SAH Cost, STD, Build Time(ms) and Render Time(ms) by Model. **BVH** is the type of BVH used, AABB is AABB BVH split with AABB volume using Binning split method; AABB(OBB) is same as AABB but traced with OBB volume; OBB(Mid) is OBB BVH with Midpoint split method; OBB(Median) is OBB BVH with Median split method; OBB(Binning) is OBB BVH with SAH Binning split method.

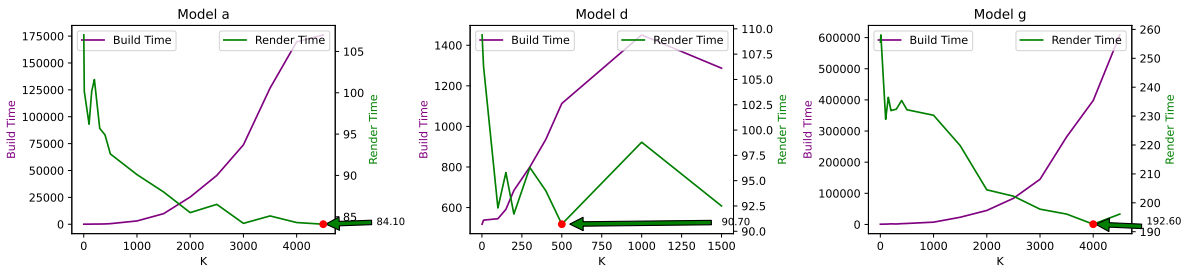
5.5. Clustering BVH OBB Nodes

In this section, we explore the effects of clustering BVH nodes using k-means clustering, which helps reduce the computational load by grouping similar leaf OBB nodes, as described in subsection 4.2.1. Specifically, we evaluate the BVHs with OBB volumes: AABB BVH with OBB, and OBB BVH configured with the binning split method. The analysis focuses on more complex models A, D, and G, as simpler models have too few nodes to benefit from clustering.

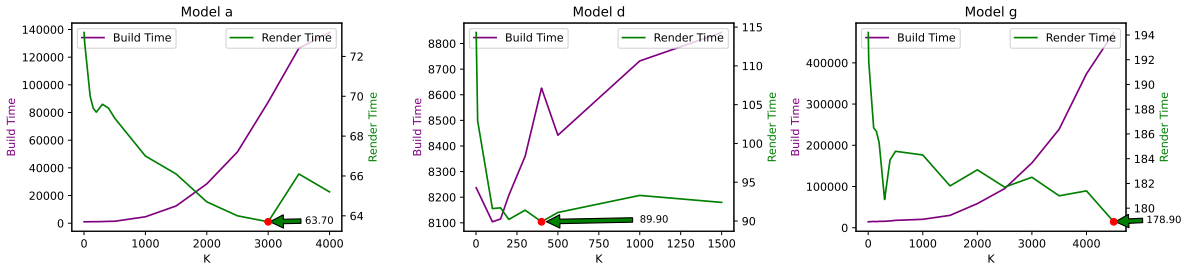
Figure 5.5 displays the performance results for models A, D, and G with clustering across the AABB BVH (tracing with OBB volume) and OBB BVH. Each of these BVHs is tested with different numbers of clusters (K), and the maximum tested K value is adjusted based on the number of leaf nodes in each model. For instance, in Figure 5.5a, model D contains around 1500 leaf nodes, so the graph concludes at K=1500.

In the AABB BVH (with OBB volume) clustering results (Figure 5.5a), all models show a clear trend: increasing the number of clusters (K) leads to longer build times but shorter render times. The optimal point is generally achieved when K is very high; however, for model D, the optimal K is around 200. Compared to the AABB BVH (with OBB volume) results in Table 5.2 without clustering, it is evident that clustering reduces render time, indicating improved tracing efficiency for OBB volumes.

The OBB BVH clustering results (Figure 5.5b) reveal a trend similar to that observed with AABB BVH clustering. Clustering effectively reduces render times across all models. The improvements are even more pronounced than the clustered AABB BVH results. This indicates that using OBB BVH and OBB volumes in conjunction with clustering not only enhances the efficiency of the BVH structure but also provides better performance benefits.



(a) AABB (OBB Volume) clustering



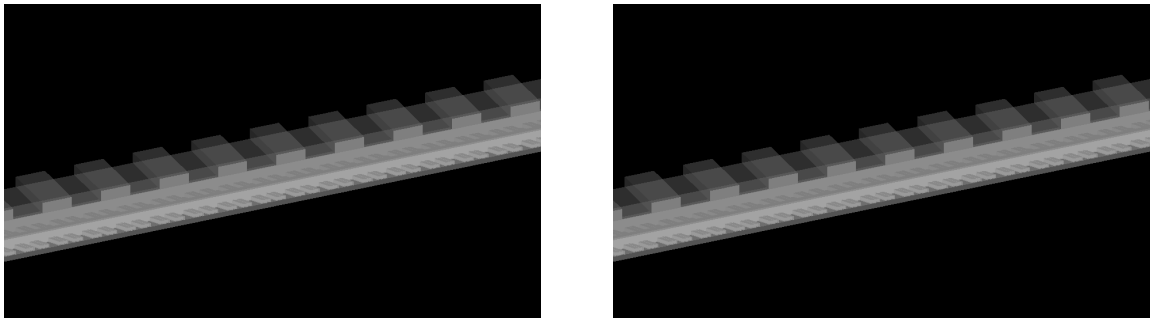
(b) OBB clustering

Figure 5.5: Comparison of different numbers of clusters.

When examining the rendered images, as shown in Figure 5.6, it is evident that the final output remains consistent regardless of whether clustering is applied. This demonstrates that the visual quality of the clustering method matches that of the non-clustering results while providing improved efficiency.

5.6. Hybrid BVH

The Hybrid BVH aims to leverage the advantages of both OBB and AABB volumes. As shown in Table 5.3, the Hybrid BVH achieves lower or comparable Surface Area Heuristic (SAH) costs relative to AABB and OBB binning in both Model B and Model D. Additionally, in Model A, the Hybrid BVH achieves a render time close to the best clustering result (63.70 ms), as seen in Figure 5.5b. It performs similarly to the AABB BVH (with AABB volumes) in Model D, as shown in Table 5.2. Moreover, the Hybrid BVH achieves the best render time among all BVHs and the clustering method in Model G. This indicates



(a) AABB BVH (AABB volume), clustering not applicable

(b) OBB BVH, with clustering

Figure 5.6: Clustering produces the same render results.

that the Hybrid BVH can effectively select the optimal volume for intersection tests, thereby leveraging the advantages of both OBB and AABB volumes and improving overall performance.

Since the Hybrid BVH uses both AABB and OBB, typically resulting in a limited number of OBB leaf nodes, and clustering only utilizes leaf nodes, it has not been tested with the clustering method.

The render times for the smaller models B, C, E, and F are close to or higher than the render time of the AABB BVH (with AABB volumes) for Model D, as shown in Table 5.2. This indicates that OBB volumes introduce more overhead than benefits during tracing models with few nodes, even in Hybrid BVH.

Model	**SAH Cost**	**Hybrid BVH** (ms)
A	1405.0	65.4
B	12.0	3.1
C	43.6	3.4
D	7077.3	45.4
E	52.3	4.8
F	132.5	8.1
G	6463.8	162.0

Table 5.3: SAH Cost and Render Times with Hybrid BVH

5.7. SIMD Render Time

Using various BVH methods, we evaluate the impact of SIMD optimizations on the render times of three 3D models—A, D, and G. Specifically, we compare AABB BVH, OBB BVH, and OBB BVH with clustering, where the clustering parameter $K = 1000$ was selected based on preliminary experiments that indicated it offers a favorable balance between BVH build time and rendering performance. Table 5.4 presents the millisecond render times for each model and BVH method combination. This demonstrates that SIMD optimizations significantly enhance performance across all BVH methods, achieving interactive frame rates suitable for real-time applications.

Model	**AABB BVH** (ms)	**OBB BVH** (ms)	**OBB BVH with clustering** ($K = 1000$) (ms)
A	12.8	9.5	**8.8**
D	**18.4**	35.0	35.3
G	53.1	**32.5**	34.1

Table 5.4: Render Times with SIMD Optimizations

For models A and G, the OBB BVH methods outperform the AABB BVH. Specifically, the OBB BVH with clustering ($K = 1000$) achieves the fastest render time for model A at 8.8 ms, a significant improvement over the AABB BVH's 12.8 ms. This performance gain is attributed to the OBB's ability to more tightly enclose non-axis-aligned geometries, reducing the number of unnecessary ray-box intersection tests during rendering. The clustering further enhances performance by grouping geometrically similar primitives, optimizing the traversal efficiency of the BVH. In contrast, model D performs better with the

AABB BVH method, rendering at 18.4 ms compared to 35.0 ms and 35.3 ms for the OBB BVH methods. This is likely due to model D's predominantly axis-aligned geometry, as discussed in Section 5.1. In such cases, the AABB BVH provides a more efficient representation because its bounding volumes align perfectly with the geometry, minimizing the volume of space within the bounding boxes.

The application of SIMD optimizations substantially impacts rendering performance. SIMD processes multiple data elements simultaneously, reducing the computational overhead associated with BVH traversal and intersection tests. This parallelism is particularly beneficial when rendering complex models like model G, where the OBB BVH method reduces render time from 53.1 ms (AABB BVH) to 32.5 ms.

6

Conclusion

This thesis has investigated the performance of BVHs performance in CPU-based ray tracing of semiconductor models, focusing on various split methods and types of bounding volumes. The primary objectives were identifying the optimal BVH configurations, comparing the performance differences between AABB and OBBs, and implementing techniques specifically designed for semiconductor structures, such as clustering BVH nodes with OBBs. In the context of semiconductor models, clustering was considered a promising technique due to the inherently detailed nature of these models, which often consist of patterns and uniform geometries causing many similar BVH nodes. By grouping similar OBBs, the clustering method can reduce the number of matrix operations for OBBs, leading to more efficient OBB intersection tests and improved BVH performance. As illustrated in Figure 5.5b, for Model A, the clustering method produced the best results compared to other methods. This demonstrates the potential of clustering BVH nodes with OBBs to enhance the efficiency of ray tracing in semiconductor models significantly.

From the results presented in chapter 5, it is evident that different BVH configurations yield varying performance across the semiconductor models tested. This variability underscores the importance of selecting an appropriate BVH structure tailored to the specific characteristics of the geometry. The findings highlight that no single BVH configuration universally excels in all scenarios. Instead, the efficiency of a BVH depends on factors such as the complexity and alignment of the model's primitives.

Specifically, OBB BVHs demonstrate superior performance in complex, non-axis-aligned geometries due to their tighter fitting around rotated shapes, which reduces unnecessary intersection tests during ray tracing. Conversely, AABB BVHs are more effective for simpler, axis-aligned structures where their alignment with the coordinate axes results in more efficient bounding volumes and faster computations due to the more straightforward and faster intersection tests of OBBs. Additionally, the Hybrid BVH approach effectively combines the advantages of AABBs and OBBs during the intersection, optimizing performance across various models.

The key contributions and findings of this thesis are summarized as follows:

1. **Effectiveness of OBB BVHs:** OBB BVHs generally achieved lower SAH costs than AABB BVHs, particularly in complex models. For instance, in Model A, the SAH cost was reduced from 5314.6 (AABB) to 527.6 (OBB), representing an improvement of approximately 90%. Similarly, in Model G, the SAH cost dropped from 26279.5 (AABB) to 5969.6 (OBB), an improvement of about 77%. OBB BVHs also demonstrated reductions in render time for complex models. In Model A, render time decreased from 86.7 ms (AABB) to 72.2 ms (OBB), a 17% improvement. For Model G, the render time was improved from 328.2 ms (AABB) to 194.0 ms (OBB), a 41% reduction.
2. **Performance Enhancement through Clustering:** The implementation of clustering methods enhanced the performance of OBB BVHs by reducing render times without compromising visual quality. For example, in Model A, clustering reduced the render time from 72.2 ms (OBB BVH) to 63.7 ms (Clustering OBB BVH), a 12% improvement. As demonstrated in Figure 5.5, clustering generally improves the rendering efficiency for OBBs by grouping similar BVH nodes and caching transformed ray directions.

3. **Advantages of Hybrid BVH:** The Hybrid BVH approach achieved optimal render times across various models, combining the strengths of AABBs and OBBs. For instance, the Hybrid BVH reduced the render time to 162.0 ms in Model G, outperforming both AABB and OBB BVHs alone. In models with predominantly axis-aligned structures, such as Model D, the Hybrid BVH it produced results similar to those of the AABB BVH, indicating its adaptability to different geometric characteristics.
4. **Impact of SIMD Optimizations:** The application of Single Instruction, Multiple Data (SIMD) techniques significantly improved rendering speeds, achieving interactive frame rates even for complex models. In Model A, SIMD reduced the render time with the OBB BVH and clustering to 8.8 ms, down from 12.8 ms with the AABB BVH. Similarly, in Model G, SIMD decreased the render time with the OBB BVH to 32.5 ms, compared to 53.1 ms with the AABB BVH, achieving interactive frame rates.

6.1. Research Questions Answered

The research questions posed in chapter 3 have been addressed as follows:

Research Question 1: Optimal BVH Techniques

Which bounding volume or BVH split strategy is optimal for ray tracing within the context of our 3D semiconductor models?

The results in chapter 5 demonstrate that OBB BVHs using the binning split method outperform AABB BVHs regarding both SAH cost and render time for complex, non-axis-aligned geometries. This is attributed to the tighter fitting of OBBs around rotated shapes, reducing the number of intersection tests required during rendering. Conversely, AABB BVHs are more effective for simpler, axis-aligned semiconductor models, where their alignment with the coordinate axes results in more efficient bounding volumes and faster computations. Therefore, the optimal BVH technique depends on the specific characteristics of the rendered semiconductor structures.

Research Question 2: BVH Tracing Improvements

How can BVH traversal techniques be optimized to improve ray tracing performance in 3D semiconductor models?

Implementing the new clustering method with cached ray directions enhanced the performance of OBB BVHs. Clustering groups similar BVH nodes, allowing for the reuse of transformed ray directions and reducing redundant calculations during OBB traversal. This optimization reduces render times without compromising visual quality, making OBB BVHs more viable for real-time applications in complex models. Additionally, leveraging the orthogonal nature of the camera and the uniformity of ray directions further optimized traversal computations.

Overall, the findings suggest that a tailored approach in selecting and optimizing BVH types based on the specific characteristics of the semiconductor models is crucial for achieving optimal rendering performance in CPU-based ray tracing.

6.2. Limitations

While the research presents advancements in CPU-based ray tracing for semiconductor models, certain limitations were identified:

- **Model Specificity:** The effectiveness of the BVH configurations and optimization techniques depends on the semiconductor models' geometric characteristics. The conclusions drawn may not generalize to models with different structures or complexities.
- **Construction Time:** The OBB BVHs, especially with the binning split method, had longer construction times than AABB BVHs. This could offset the benefits gained during rendering in scenarios where the scene changes frequently.
- **Clustering Overhead:** While clustering improved render times, it introduced additional computational overhead during the BVH construction phase. The optimal number of clusters varied across models, requiring model-specific adjustments.

6.3. Future Work

Building upon the findings and acknowledging the limitations of this research, several avenues for future work are proposed:

- **Adaptive Clustering Strategies:** Implement adaptive clustering methods that automatically determine the optimal number of clusters based on the model's characteristics, reducing the need for manual tuning and ensuring consistent performance improvements across different models. Additionally, consider extending clustering to all nodes, as current methods only cluster leaf nodes.
- **Enhancing SIMD Implementation:** Investigate and address the noise artifacts introduced by SIMD optimizations in OBB BVHs. This could involve refining the intersection algorithms or exploring precision issues in SIMD computations to ensure both performance gains and visual accuracy.
- **Parallel BVH Construction:** Explore parallelization of the BVH construction phase, leveraging multi-core CPU architectures to mitigate the increased build times associated with OBB BVHs and clustering.
- **Adapting State-of-the-art BVH Construction Method:** Incorporate advanced methods like spatial splits into traditional BVHs to reduce bounding volume overlap, particularly for long triangles that span large areas. SBVH's [23] ability to split triangles into multiple parts and reference them in different child nodes helps minimize overlap and improve traversal efficiency in complex scenes.

References

[1] Timo Aila and Samuli Laine. “Understanding the efficiency of ray traversal on GPUs”. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG ’09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 145–149. ISBN: 9781605586038. DOI: 10.1145/1572769.1572792. URL: https://doi.org/10.1145/1572769.1572792.

[2] James H. Clark. “Hierarchical geometric models for visible surface algorithms”. In: *Commun. ACM* 19.10 (Oct. 1976), pp. 547–554. ISSN: 0001-0782. DOI: 10.1145/360349.360354. URL: https://doi.org/10.1145/360349.360354.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Volume 2: Instruction Set Reference, A-Z. Intel Corporation. Santa Clara, CA, July 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[4] Ryan R. Curtin et al. “mlpack 4: a fast, header-only C++ machine learning library”. In: *Journal of Open Source Software* 8.82 (2023), p. 5026. DOI: 10.21105/joss.05026. URL: https://doi.org/10.21105/joss.05026.

[5] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[6] Haitao Du et al. “Interactive ray tracing on reconfigurable SIMD MorphoSys”. In: *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. 2003, pp. 471–476.

[7] Christer Ericson. *Real-Time Collision Detection*. USA: CRC Press, Inc., 2004. ISBN: 1558607323.

[8] Valentin Fuetterling et al. “Efficient Ray Tracing Kernels for Modern CPU Architectures”. In: *Journal of Computer Graphics Techniques (JCGT)* 4 (Dec. 2015), pp. 90–111.

[9] Yan Gu et al. “Efficient BVH construction via approximate agglomerative clustering”. In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG ’13. Anaheim, California: Association for Computing Machinery, 2013, pp. 81–88. ISBN: 9781450321358. DOI: 10.1145/2492045.2492054. URL: https://doi.org/10.1145/2492045.2492054.

[10] Vlastimil Havran. “Heuristic Ray Shooting Algorithms”. PhD thesis. Nov. 2000.

[11] Hwancheol Jeong et al. *Performance of SSE and AVX Instruction Sets*. 2012. arXiv: 1211.0820 [hep-lat].

[12] Tero Karras. “Maximizing parallelism in the construction of BVHs, octrees, and k-d trees”. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG’12. Paris, France: Eurographics Association, 2012, pp. 33–37. ISBN: 9783905674415.

[13] Thomas Larsson and Linus Källberg. “Fast Computation of Tight-Fitting Oriented Bounding Boxes”. In: Feb. 2011, pp. 3–19. ISBN: 978-1-56881-437-7. DOI: 10.1201/b11333-3.

[14] 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.

[15] J.D. MacDonald and K.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.

[16] J. MacQueen. “Some Methods for Classification and Analysis of Multivariate Observations”. In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif.: University of California Press, 1967, pp. 281–297. URL: https://projecteuclid.org/euclid.bsmsp/1200512992.

[17] 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.

[18] Chuck Pheatt. “Intel® threading building blocks”. In: *J. Comput. Sci. Coll.* 23.4 (Apr. 2008), p. 298. ISSN: 1937-4771.

[19] Steven M. Rubin and Turner Whitted. “A 3-dimensional representation for fast rendering of complex scenes”. en. In: *Proceedings of the 7th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’80*. Seattle, Washington, United States: ACM Press, 1980, pp. 110–116. ISBN: 978-0-89791-021-7. DOI: 10.1145/800250.807479. URL: http://portal.acm.org/citation.cfm?doid=800250.807479.

[20] Rodolfo Sabino et al. “Fast and Robust Ray/OBB Intersection Using the Lorentz Transformation”. In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 519–528. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_32. URL: https://doi.org/10.1007/978-1-4842-7185-8_32.

[21] V. V. Sanzharov, V. A. Frolov, and V. A. Galaktionov. “Survey of Nvidia RTX Technology”. In: *Program. Comput. Softw.* 46.4 (July 2020), pp. 297–304. ISSN: 0361-7688. DOI: 10.1134/S0361768820030068. URL: https://doi.org/10.1134/S0361768820030068.

[22] Jonathon Shlens. *A Tutorial on Principal Component Analysis*. 2014. arXiv: 1404.1100 [cs.LG].

[23] Martin Stich, Heiko Friedrich, and Andreas Dietrich. “Spatial splits in bounding volume hierarchies”. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG ’09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 7–13. ISBN: 9781605586038. DOI: 10.1145/1572769.1572771. URL: https://doi.org/10.1145/1572769.1572771.

[24] N. Vitsas et al. “Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees”. en. In: *Computer Graphics Forum* 42.2 (May 2023), pp. 245–254. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.14758.

[25] I Wald et al. “OSPRay - A CPU Ray Tracing Framework for Scientific Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 931–940. DOI: 10.1109/TVCG.2016.2599041.

[26] Ingo Wald. “On fast Construction of SAH-based Bounding Volume Hierarchies”. In: *2007 IEEE Symposium on Interactive Ray Tracing*. 2007, pp. 33–40. DOI: 10.1109/RT.2007.4342588.

[27] Ingo Wald, Solomon Boulos, and Peter Shirley. “Ray tracing deformable scenes using dynamic bounding volume hierarchies”. In: *ACM Trans. Graph.* 26.1 (Jan. 2007), 6–es. ISSN: 0730-0301. DOI: 10.1145/1189762.1206075. URL: https://doi.org/10.1145/1189762.1206075.

[28] Run Yan et al. “RT Engine: An Efficient Hardware Architecture for Ray Tracing”. In: *Applied Sciences* 12.19 (2022). ISSN: 2076-3417. DOI: 10.3390/app12199599. URL: https://www.mdpi.com/2076-3417/12/19/9599.

[29] Robin Ytterlid and Evan Shellshear. “BVH Split Strategies for Fast Distance Queries”. In: *Journal of Computer Graphics Techniques (JCGT)* 4.1 (Jan. 2015), pp. 1–25. ISSN: 2331-7418. URL: http://jcgt.org/published/0004/01/01/.