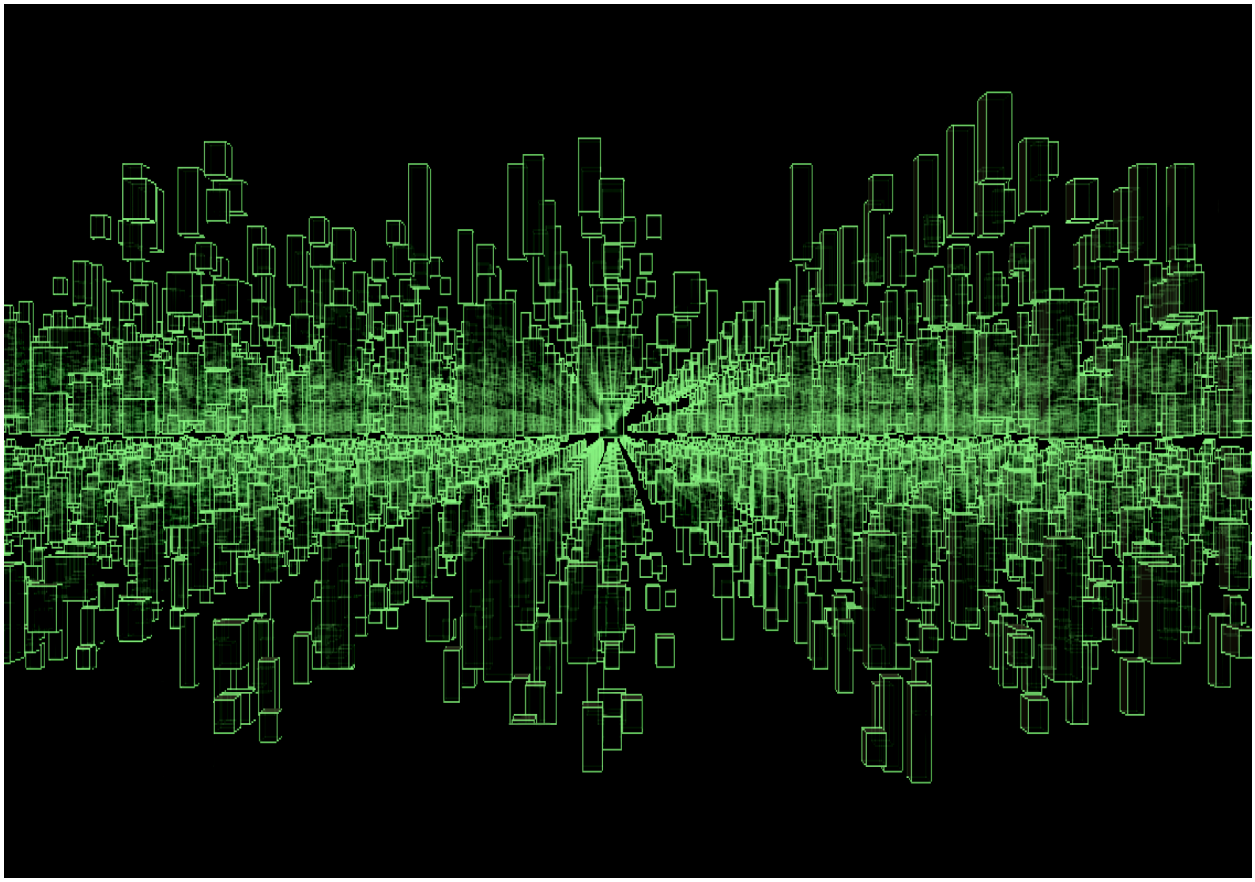

Evaluation of Acceleration Structures for Ray Casting in Physics Scenes

Master's thesis in Computer science and engineering

Chenxu Guo
Haowei Liao

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Evaluation of Acceleration Structures for Ray Casting in Physics Scenes

Chenxu Guo
Haowei Liao

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Evaluation of Acceleration Structures for Ray Casting in Physics Scenes
Chenxu Guo
Haowei Liao

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisors: Danylo Ierkaiev, Andrii Nikolaiev, Axel Hornay, Massive Entertainment
Examiner: Erik Sintorn, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of axis-aligned bounding boxes in a uniform scene

Typeset in LaTeX
Gothenburg, Sweden 2023

Evaluation of Acceleration Structures for Ray Casting in Physics Scenes
CHENXU GUO, HAOWEI LIAO
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Ray casting is a fundamental feature of game engines, and has a wide range of applications in the field of video game development. As a performance-critical task, a significant number of studies have proposed various acceleration structures to improve the efficiency of ray casting. In this study, we collaborated with Massive Entertainment to investigate optimal acceleration structures for ray casting within their in-house game engine, Snowdrop. We implemented several promising acceleration structures and developed a testing framework to evaluate their performance. The acceleration structures we implemented include uniform grids (UG), hierarchical hash grids (HG), dynamic bounding volume hierarchies (DBVH) and linear bounding volume hierarchies (LBVH). To obtain representative results, we tested these algorithms on a set of uniform scenes generated by the Unity3D engine, as well as on irregular scenes exported by the Snowdrop engine. The test items included the build time of the acceleration structure, the update time, and the time taken to perform 1000 ray castings. The results were used as a basis for evaluating the performance of the different acceleration structures. Furthermore, to gain a deeper understanding of the reasons for the differences in performance of these acceleration structures, we also introduced a performance model to analyze the details of the execution of these structures. Finally, we found that HG and DBVH achieved the best balance of query speed and update speed among all the acceleration structures involved in the comparison.

Keywords: ray casting, broad phase, acceleration structure, physics scene, comparative study.

Acknowledgements

We would like to thank our supervisor, Erik Sintorn, for his patient guidance and support throughout this project. His insights and input were instrumental in determining the direction of our research and the design of the experiments. He also helped us review the entire report and gave us detailed suggestions for revisions, which led to a huge improvement in the quality of the report. We would like to express our thanks to Danylo Ierkaiev, Andrii Nikolaiev, and Axel Hornay from Massive Entertainment, who answered many questions we had during the study and and deepened our understanding of the topic. They also provided us with valuable scene data, which was crucial for our experiments.

Chenxu Guo and Haowei Liao, Gothenburg, 2023-02-06

Contents

List of Figures

List of Tables

1

Introduction

As the video game industry continues to grow and evolve, video games have assimilated into people's lives [1]. The number of video gamers worldwide was estimated to be 3.03 billion in 2022 [2]. Advances in technology have allowed for the creation of more expansive and content-rich game environments, which in turn have enabled the development of immersive gameplay. An accurate physics simulation is crucial for creating a realistic game experience, and this requires the game to probe and recognize the game environment in real-time. Ray casting is a common technique used to accomplish this task that determines which objects a virtual ray or line segment will intersect with as it moves through the game world.

Ray casting allows the game to respond to player actions and enable more intricate game mechanisms. It is frequently used to assess an object's visibility from a specific angle and to trigger physical interactions between game objects. For example, the game can employ ray casting to trace the path of a bullet from the gun to the target object, and determine whether the bullet hits the target or not. Since the game logic may update states or cast new rays depending on the result of the previous ray, ray casting queries should be answered as near instantly as possible. Otherwise, the player may suffer a considerable dip in frame rate which will detrimentally affect the gaming experience.

As a versatile and fundamental functionality, ray casting is widely supported in many game engines and physics engines, such as Bullet [3], PhysX [4] and Havok [5]. This study investigates high-performance ray cast solutions in collaboration with Massive Entertainment [6], a Swedish video game studio that has developed its own in-house game engine, Snowdrop [7]. Snowdrop has been used to created a number of well-known video games, including World in Conflict, Tom Clancy's The Division and The Division 2 [6]. These games typically feature large, dense physics scenes with hundreds of thousands entities. The ray casting performance of Snowdrop needs to be further enhanced in order to meet the requirements of current and potential future video games.

1.1 Problem Overview

Ray casting is a searching problem that is commonly encountered in the field of computer graphics and video game development. It involves searching through a large number of objects in the scene to determine which ones intersect a given ray. A brute force approach to this problem, which simply checks the intersections for all objects, has a time complexity of $O(n)$ where n is the number of objects in the scene. However, in real-time applications such as video games, this method can be prohibitively slow and may lead low frame rates. To address this issue, various acceleration structures have been proposed, which can greatly improve the efficiency of the ray casting task by reducing the number of objects that need to be considered during the search. These structures use techniques such as spatial partitioning and object hierarchies to pre-process the game world and enable faster and more efficient ray casting.

Despite the numerous studies that have developed various acceleration structures and compared existing solutions, it remains challenging to reach a consensus on which algorithm is the most efficient. These studies have indicated that different acceleration structures can be more suitable for different scenarios and applications, depending on factors such as the complexity of the scene, the performance constrains of the application, and the specific characteristics and limitations of the acceleration structure. For instance, uniform spatial subdivision techniques are highly effective for uniformly distributed scenes, but they are not as efficient as object hierarchy techniques in irregularly distributed scenes. Consequently, selecting the appropriate acceleration structure for a given application necessitates fair and thorough analysis.

As the game scene constantly undergoes changes across frames, the acceleration structures must also be updated in order to maintain their effectiveness. Therefore, when evaluating the performance of these structures, it is important to consider the time consumption for updates in addition to query speed. For instance, in the current implementation of the Snowdrop engine, the Bounding Volume Hierarchy (BVH) has been found to have a query speed that is on average 30% faster than the uniform grid(UG). However, the update time for BVH is significantly higher than that of the UG. This emphasizes the need for further optimization of acceleration structures in order to balance the conflicting demands of query speed and structure update time.

1.2 Research Question

Given the importance of acceleration structures in optimizing ray casting performance, we will investigate the following research question:

> **Which acceleration structure is the most efficient in terms of ray casting queries and structure updates in the physics scenes that are typically used in the Snowdrop engine?**

To answer the research question, we have implemented several acceleration structures, including spatial subdivision schemes and object hierarchy schemes, and conducted a comparative study to identify the pros and cons of each. In order to systematically evaluate the efficiency of these different acceleration structures, we have developed a performance model. Through this approach, we aim to gain a deeper understanding of the trade-offs involved in the choice of acceleration structure for ray casting in large physics scenes.

1.3 Delimitation

This study focuses on the use of ray casting in physics simulation for video games, as opposed to its more commonly seen application in ray tracing. While both tasks involve searching for intersections between rays and objects, there are significant differences in terms of their requirements and performance constraints. In ray tracing, the number of rays used is typically in the millions, which is significantly higher than the number of rays used in physics simulation, which may only be a few hundred. Given the lower demand for rays in physics simulations and the frequent updates to the simulated scene, this study does not focus on acceleration structures that prioritize query performance at the expense of update/rebuild time. Another difference between the ray tracing and ray casting algorithms is that the former tends to find only the nearest collision point, while the latter needs to find all points of intersection with the ray. This can result in slower execution speeds for the ray casting algorithm, especially in scenes with a large number of objects.

Another delimitation of this thesis is that it only covers the acceleration of ray casting on the CPU. However, many state-of-the-art studies in this area have focused on optimizing acceleration structures for use on the GPU, which can provide a significant performance boost due to the parallel processing capabilities of the GPU. Since it is a common practice for game engines to handle ray casting on the CPU, this thesis only focus structures designed for the CPU, which may have different performance characteristics and limitations. As a result, the techniques and conclusions of this thesis may not be directly applicable to the optimization of ray casting on the GPU.

2

Previous Work

2.1 Acceleration Structures

Over the past several decades, a significant number of studies have been conducted with the goal of improving the performance of ray casting. This has led to the development of a wide range of acceleration structures. In general, these acceleration structures can be grouped into two main categories: space partitioning schemes and object hierarchies [8].

Fujimoto, Tanaka, and Iwata [9] mentioned early on UGs, which is one of the representatives of space partitioning acceleration structures. They also discussed octrees and compared the traversal details between octrees and UG. The octree is described thoroughly by Jackins and Tanimoto [10]. Wald, Ize, Kensler, *et al.* [11] found that when the positions of objects in the scene are scattered or when there are large empty areas, hierarchical grids generally have better performance. When the scene is very big or the grid size is very small on a UG, storing the grid information requires a large amount of space. Cleary and Wyvill [12] suggest a hashing method for storing grid data to reduce this space. Fuchs, Kedem, and Naylor [13] proposed BSP tree, recursively dividing the space with planes, a non-uniform space partitioning schemes. A KD tree [14] is a special case of a BSP tree, the difference being that the plane used for division in a KD tree is perpendicular to the axes [15].

In object hierarchies, objects are hierarchically organized instead of dividing the space into disjoint regions [16]. These techniques mainly refer to bounding volume hierarchies (BVH) and their variants. BVHs can be constructed using top-down methods that splits at the spatial median [17] or the object median to get a balanced tree [18]. The surface area heuristic (SAH) is usually used in the construction of BVHs to achieve higher tree quality. This method employs a greedy strategy to recursively evaluate potential partitions in an attempt to find the minimum possible traversal cost [19][20]. Although top-down build is intuitive to implement, it can be hard to parallelize. Most parallel BVH construction algorithms sort points along a space-filling curve and build the tree in a bottom-up fashion. Then the bounding volumes are computed with top-down methods [21]. Linear bounding volume hierarchies (LBVH) use Morton codes for sorting. The primitives are divided into clusters that can be handled in parallel. The construction time of LBVH is linear to the

number of primitives [22]. BVHs can also be constructed incrementally, enabling dynamic insertion of objects into the hierarchy without reconstructing the entire structure. [23].

2.2 Performance Model

Performance is often the main factor in choosing an acceleration structure. Due to the diversity of test scenes and hardware platforms, it is often difficult to draw universally applicable conclusions from these studies. For example, Fujimoto, Tanaka, and Iwata [9] revealed that uniform grids have better performance than octrees in several test scenes, but some other studies [24] have come to the opposite conclusions in scenes with arbitrarily distributed objects [16]. In order to fairly and in-depth compare the structures, Havran [25] suggested a performance model in his PhD thesis, which established the relationship between running time and different arithmetic operators involved in the algorithm. Meanwhile, the factors that can affect the performance of acceleration structures were also pointed out, including the complexity of the input scenes, the idea behind the algorithm, testing procedure, the hardware platforms, the compiler and the implementation.

There have been many other studies focusing on using analytical models to evaluate acceleration structures. Cleary and Wyvill [12] analyzed uniform grids in the context of ray tracing. The time consumption of a single ray-casting query was divided into three parts: the preparatory steps for the traversal process, the ray-object intersection tests and the grid traversal. Kay and Kajiya [17] discussed the theoretical performance of bounding volume hierarchies in the context of ray tracing. The overhead of the BVH was split into ray-object intersection tests and the traversal of the underlying hierarchy structure. The cost of intersection tests were represented by the arithmetic operations involved. Havran [16] extended this model by the cost of moving data blocks from memory to registers and the cost of pre-processing steps, such as locating the starting step for the traversal of UGs.

In order to measure and compare the execution time of the acceleration structures, Serpa and Rodrigues [26] developed Broadmark, a framework for assessing broad-phase collision detection algorithms. This framework contains two parts: a test data generator that runs in the Unity3D game engine, and an algorithm simulator implemented with C++. The generator exports the bouding volumes of collidable objects in the scene as well as the results of collision detection frame-by-frame. The recorded bounding volumes is used as the input to construct the acceleration structures in the simulator. And the precomputed collision detection results serve as the ground truth for testing the reliability of new algorithms. In that way, the framework can take the advantages of the flexibility of the game engine to generate test data for any types of scenes. It is noteworthy that Broadmark was specifically intended for evaluating collision detection algorithms rather than ray casting. As such, it could not be applied directly in our study. To address this limitation, we developed a framework for evaluating the performance of ray casting algorithms, utilizing Broadmark as an architectural reference.

3

Background

This chapter provides the background and introduces the terminology relevant to the present study. We begin by giving a general introduction to collision-detection systems in game engines, highlighting the role of ray casting as a core function within these systems. We then clarify the distinction between ray casting and collision detection. Subsequently, we provide a brief overview of the acceleration structures covered in this study.

3.1 The Collision Detection System

The collision-detection system is an essential module of game engines, as almost all 3D and 2D games will need it. Its primary functionalities are collision detection and collision queries. Since the system requires frequent collision detection and queries, it usually has its own "collision world", including collidable entities of each collidable object. This design is common among physics systems, like $hkpWorld$ in the Havok engine and $NxScene$ in the PhysX engine. A collidable entity, also known as a collidable primitive, contains a transform and a shape. Common collidable primitives are spheres, capsules, axis-aligned bounding boxes, oriented bounding boxes, and arbitrary convex volumes since the collision detection between these primitives is relatively more straightforward than between actual object meshes [27].

- **Collision Detection**: As the name suggests, a major function of the collision detection system is to detect whether collisions occur between objects in the scene. In order to give an object the ability to collide with other objects, every collidable object needs to have its collision representation. The collision representation of an object may contain single or multiple collidable entities, depending on the complexity of the object. The collision detection event not only returns whether the objects intersect, but also provides more specific information about the contact, such as the collision point and the separating vector. The separating vector can be used to help move objects out of the collision zone. At the same time, collision detection has many other uses. It can trigger a reaction when a sword hits the player or allow the character to pick up coins when he touches them.

- **Collision Queries**: Collision query, as another primary function of collision detection system, mainly solves the hypothetical problem. Because, unlike collision detection, it detects collisions with imaginary objects that do not exist in the scene instead of actual objects. The most commonly used queries is collision cast, which determines what a hypothetical object would collide with while moving alone a line segment [27]. However, objects placed into the collision world by a collision cast are not physically present in the world and therefore have no real effect on the world. The collision casts are usually divided into ray casts and shape casts.

3.2 Ray Cast

Ray casting is a method of finding out whether a virtual ray intersects with any collidable entities in a scene. A ray is a directed line segment, defined by an initial point $\boldsymbol{P_0}$ and a direction vector $\boldsymbol{d}$, terminating at a point $\boldsymbol{P_1}$ after traversing a distance t. The equations for the rays are usually as follows:

$$\boldsymbol{P_1} = \boldsymbol{P_0} + \boldsymbol{d} \cdot t \tag{3.1}$$

If intersections occur, the ray casting will find the intersecting point or set of points and return relevant information, usually including the parameter t corresponding to their contact point and identifier of the collidable entity. Figure 3.1 shows how a ray casting algorithm works. Ray casts have a wide range of applications in games. They can, for example, detect which part of the enemy a bullet is likely to hit, detect whether the player is within the enemy's line of sight, and detect whether the player has both feet on the ground.

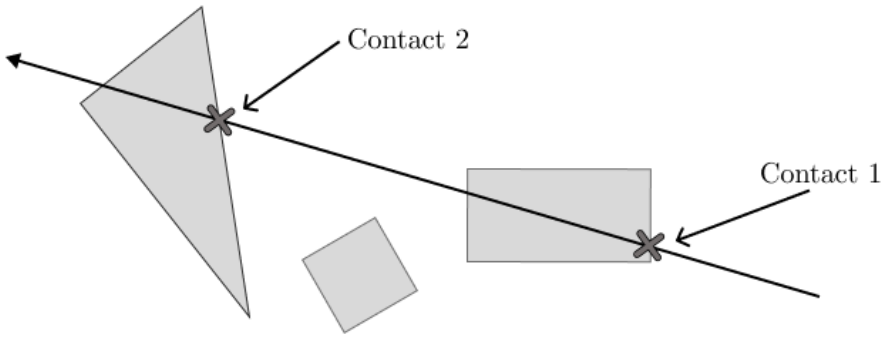

Figure 3.1: The illustration of a ray casting algorithm.

3.3 Bounding Volumes

Direct collision detection or collision query can be very time-consuming when objects consist of many polygons. In order to reduce this time, many approaches propose to use bounding volumes(BV) instead of the objects themselves, as simpler volumes often require only a few steps to complete the collision calculation. A bounding

volume is a simple alternative volume that completely encloses the object. This alternative method is quick but not precise enough. If more accurate information is required such as the collision point of an object, further collision calculations with the original object are required. A more detailed acceleration process is discussed in the later section.

The axis-aligned bounding box (AABB) is a widely-used bounding volume [28]. It is a six-sided rectangular box (four sides in 2d), and all sides are parallel to one of the axes of the coordinate system in which it is located. It has the advantage that the intersection calculation is efficient, but it is not always a tight fit for the original object. Another type of bounding volume that can be used is the Oriented Bounding Box (OBB), which is similar to the AABB, but can be oriented in any direction. OBBs are particularly useful for slender objects that are not parallel to the axes, as they provide a better fit than AABBs. [29]. There are also other types of bounding volumes such as cones, cylinders, and spherical shells, each with their own features and applications. [30] [31].

3.4 Acceleration Process

In complex scenes which contain a vast amount of entities, the expense of collision detection or collision query against all objects directly is unacceptable. To achieve smoother physics simulation and gaming experience, the collision detection system must be fast enough. So as to address the performance issue, this system is practically divided into two successive phases: the broad and narrow phase [32].

- **Broad-phase**: The Broad phase serves to minimise the number of objects that need to enter the narrow phase for further check. Detection in this phase is usually based on the BV of the object, rather than the collidable entities of the object, as it is much faster. At the same time, the introduction of BV can lead to misjudgements, resulting in the detection of more objects as colliding than is actually the case. Despite the addition of a new detection phase and the potential for false positives, the overall operation is more efficient than without the broad phase.

- **Narrow-phase**: After eliminating a large number of objects that are not found to intersect during the broad-phase, the narrow-phase collision detection or collision query will determine precisely whether individual objects intersect by examining their collidable entities. The result will be used for collision response, as well as contact determination [32].

Figure 3.2 demonstrates the above process, where the objects represented in translucent form have been filtered out after the broad phase. The remaining objects are then processed in the narrow phase, with their intersections with the ray being identified.

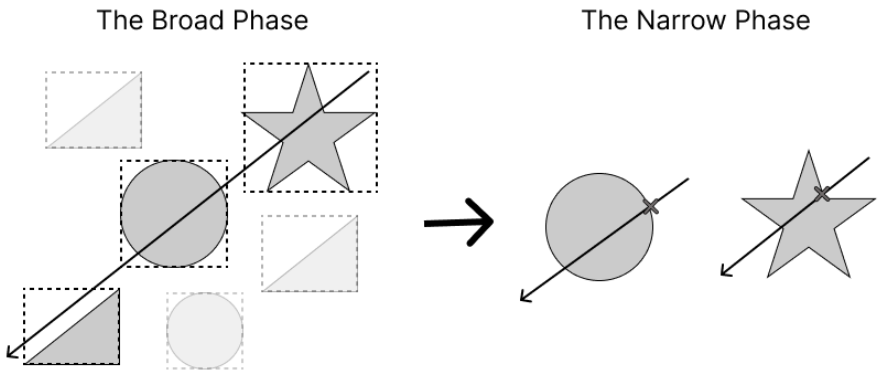

Figure 3.2: A diagram illustrating the acceleration process.

3.5 Acceleration Structures

Currently, the Snowdrop engine utilizes 2D uniform grids, a spatial partitioning scheme, for its ray casting algorithm [33]. Additionally, the engine also incorporates BVHs (Bounding Volume Hierarchies) as an alternative solution, which is an object partitioning scheme. We therefore present here in detail the concepts of these acceleration structures.

3.5.1 Uniform Grids

As shown in the left panel of Figure 3.3, the concept of uniform grids is quite straightforward; Space is simply divided into a number of equal-sized cells. Each object is assigned to the cell it overlaps. Since the cells are independent regions and can be predicted at run-time, updates to the objects in the cells can be done in parallel, which can significantly reduce execution time. Ray casting over uniform grids often relies on the 3D-DDA algorithm, in which cells are traversed by drawing a 3D straight line in the grid [34]. Since the space is uniformly subdivided, the coordinates of a specific cell can be determined based on the size of the cells. Thus, associating objects to cells is relatively fast. However, a typical game scene always contains objects of various sizes and complexity. So there is no universal cell size that is suitable for any scene. An inappropriate cell size will make the resolution of the grid too fine or too coarse, which might cause serious memory footprint or performance issues [35].

3.5.2 Hierarchical Grids

Many papers have discussed the design of hierarchical grids. Cazals, Drettakis, and Puech [36] propose loosely nested grids, while Parker, Parker, Livnat, *et al.* [37] propose macrocells. We have chosen recursive grids [38], in which all objects are divided into a root cell and continuously subdividing the root cellas depicted in the right diagram of Figure 3.3. The difference is that the scale of subdivision in recursive grids is configurable and not necessarily 2x2x2. Subdivision will not occur

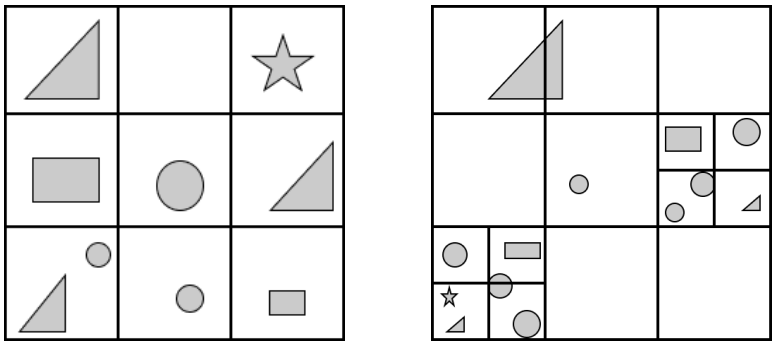

Figure 3.3: A uniform grid and a hierarchical grid

if the number of objects in a grid is less than a certain value or if the depth of the grid is greater than a characteristic value. Hierarchical Grids is efficient in dividing space when objects in a scene are not evenly distributed, such as in the presence of large empty areas or when a large number of objects are packed into a small area. It only requires a small number of grid strips to divide empty areas, while higher resolution grids are used in areas where objects are densely packed.

3.5.3 Bounding Volume Hierarchies

As an object partitioning scheme, bounding volume hierarchies don't subdivide the space, but wrap objects into Bounding Volumes (BV). The BVs usually have simple shapes such as boxes or spheres. Even though testing a BV is faster than a complex object, it is still very expensive to check all the BVs in the scene to locate the objects hitting by the ray. Using a BVH solves this problem with its hierarchical tree structure. As illustrated in Figure 3.4, the geometric primitives are held by the leaves, and BVs are held by internal nodes. A parent node holds a BV that encloses all the triangles of its child tree. So the top node of the tree is a big BV which contains all the geometric primitives of the scene. Testing against a BVH is much faster than checking all the objects in the scene. The process of testing a ray against a BVH usually starts with testing the head node, then test each of the children nodes and repeat this process recursively until reaching the leaf nodes. With this approach, the asymptotic time complexity can be reduced to logarithmic in the number of leaf nodes [33].

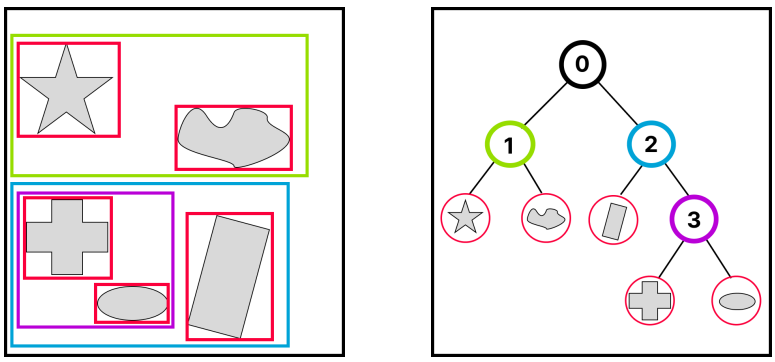

Figure 3.4: A bounding volume hierarchy and its tree representation

In a physics simulation, objects keep moving from one time step to the next. Bounding volume hierarchies can handle dynamic scenes in two main ways: by rebuilding the hierarchy from scratch at each simulation step or update the hierarchy to reflect the change of the scene. The Linear BVH (LBVH) and Dynamic BVH (DBVH) are representative algorithms that illustrate these two approaches, respectively [23][39]. The DBVH has the ability to adapt to changes in the scene, such as the insertion, removal, or movement of objects, without the need to completely rebuild the structure. To maintain the quality of the structure, the DBVH uses Surface Area Heuristics(SAH) to minimize the cost while inserting new objects to the tree. Unlike DBVH, which uses an incremental update strategy, the LBVH maintains tree quality by completely reconstructing the tree every frame. LBVH is constructed in parallel, which allows it to achieve certain tree quality with an acceptable time overhead. Additionally, the tree nodes of LBVH are stored linearly in memory, potentially resulting in faster traversal speeds.

3.5.4 The Surface Area Heuristic

The efficiency of a BVH is depend on how the hierarchy is formed. Traditional top-down approaches frequently partition objects based on the spatial median or the object median. Many state-of-the-art algorithms utilize the Surface Area Heuristic (SAH), which have been demonstrated to generate hierarchies of higher quality [40]. A higher quality hierarchy implies that a smaller number of nodes need to be visited during ray casting queries.

The fundamental idea of SAH is that the probability of a random ray intersecting an object within the scene is proportional to the surface area of that object [19]. In consideration of this principle, SAH is commonly used to estimate the cost of traversing a bounding volume tree in the construction of the BVH. Higher tree quality can be achieved by minimizing the SAH cost. The SAH cost model can be described by the following equation

$$C(T) = \frac{1}{SA(T)} \left[C_T \cdot \sum_{N \in \text{ inner nodes}} SA(N) + C_I \cdot \sum_{N \in \text{ leaves}} SA(N) \cdot N_I \right], \qquad (3.2)$$

where $C(T)$ is the cost of traversing the BVH, $SA(T)$ is the surface area of the world bounding box, $SA(N)$ is the surface area of the bounding box of a inner or leaf node, C_T is the cost of traversal step, C_I is the cost of a intersection test, N_I is the number bounding volume intersection tests.

4

Acceleration Structures Implementation

To evaluate the performance of various acceleration structures, we developed a testing framework comprising of an algorithm simulator. This simulator uses precomputed data to simulate the execution of individual acceleration structures, assessing their performance in various aspects and generating output logs, as explained in more detail in Section 5.2.2. In this chapter, we present the details of each acceleration structure implemented in the simulator. These details primarily include the underlying ideas for the construction and traversal algorithms, as well as any variations in the strategies or implementation methods within each algorithm. For example, whether recursion is used in the construction and traversal of BVHs, the grid construction strategy for uniform grids.

4.1 Brute Force

Brute Force (BF) refers to the process of performing ray-bounding volume tests on all objects in the scene. The execution time of BF is linearly proportional to the number of objects present in the scene. In this study, BF is used as a baseline for comparison with other acceleration structures. It is simply an array that stores the objects in the scene and does not require special construction or traversal algorithms. As such, it does not introduce additional traversal overhead like other acceleration structures do.

4.2 Uniform Grids

Uniform grid is a representative acceleration structure in Spatial Partitioning. Another reason we chose it was that the Snowdrop engine chose it as one of the built-in acceleration structures. We felt the need to measure and analyse its performance. The basic concept of the uniform grid has already been described in the previous section, the following section will focus on the specific implementation of the uniform grid in our simulator.

4.2.1 Construction

The UG construction algorithm consists of two steps. First, the scene is evenly divided into cells according to some strategy, and then all objects in the scene are assigned to the cells. However, the density of the grid depends on the size of cell. If the cells are too small, the grid will have more cells, which means ray casting may spend more time on traversal. And if the cells are too big, the result of broad phase will contain many false positive, resulting in a greater cost to reject these objects. When there is a large difference in the size of the objects in the scene, the size of the cell will be more difficult to decide. It is therefore very difficult, even impossible, to choose a universal size for cells in uniform grid.

$$cells\_per\_dimension = \left(\frac{number}{density}\right)^{\frac{1}{3}}; size = \frac{longest\_edge}{cells\_per\_dimension} \tag{4.1}$$

As for our implementation, we choose a relatively generic construction strategy, where large objects will exist in multiple cells. It provides an external parameter *density* that allows the user to adjust the grid density, or the number of objects stored in each cell. It is important to note that this parameter is only used to calculate the cell size and does not guarantee the actual number of objects present in each cell. For UG, the formula for cell size is shown in Equation 4.1. *longest_edge* refers to the length of the longest edge in the world AABB and *number* refers to the number of objects in the scene.

For updating the acceleration structure every frame, we update the existing data structure rather than rebuilding the whole grid, even though UG rebuild is generally efficient [41]. However, in our use case, the number of dynamic objects is relatively low and there is a significant overhead in rebuilding the acceleration structure every frame. As a result, we opted to update individual altered objects instead. Updating consists of three processes: inserting, removing, and moving objects. Inserting an object is similar to adding a single object during the construction. Removing an object involves calculating the number of cells each object occupies and then deleting the object from the cells. Moving an object, as previously described, involves first removing it and then inserting it in a new location.

4.2.2 Traversal

Our implementation mainly based on the algorithm introduced by Amanatides and Woo [34]. The Bresenham algorithm requires an axis to be determined before traversal, whereas this algorithm does not. In addition to this, the algorithm proposes an idea for avoiding multiple intersections when the object exists in multiple cells at the same time. The algorithm assigns a unique ID 'rayID' to each ray and stores a 'rayID' field for each object (initialized to 0). When testing for intersection between a ray and an object, the algorithm first compares the 'rayID' fields of the ray and

the object. If they are the same, the test is skipped because it means that they have been compared before. If the 'rayID' fields are different, the test continues and, at the end of the test, the 'rayID' of the ray is stored in the object's 'rayID' field. In this step, we made some modifications to save memory and also to avoid multiple intersections. We use a bool vector of length equal to the number of objects to represent the test state of each object. The vector is reset to all false values before each ray cast is started. The test is conducted in a similar way to the one mentioned earlier. If the value of the vector for an object is true, the test is skipped. If the value is false, the test is continued and the value is set to true at the end of the test. The details are shown below as pseudocode:

Algorithm 1 UG::RayCast

```
1: function RayCast(ray_start, ray_end, out_hit_entities)
2:    initialize the entity_hit_status, start_index[3], end_index[3],
   directions[3] intersection[3] and delta_time[3];
3:
4:    for ; ; do
5:       id ← start_index[0] × cells_per_dimension × cells_per_dimension +
   start_index[1] × cells_per_dimension + start_index[2]
6:       break if it out of the grid;
7:       intersect ray with each object;
8:       if intersection[0] ≤ intersection[1] and intersection[0] ≤
   intersection[2] then
9:          if start_index[0] = end_index[0] then
10:            break
11:         end if
12:         intersection[0] ← intersection[0] + delta_time[0]
13:         start_index[0] ← start_index[0] + directions[0]
14:      else if intersection[1] ≤ intersection[0] and intersection[1] ≤
   intersection[2] then
15:         if start_index[1] = end_index[1] then
16:            break
17:         end if
18:         intersection[1] ← intersection[1] + delta_time[1]
19:         start_index[1] ← start_index[1] + directions[1]
20:      else if intersection[2] ≤ intersection[1] and intersection[2] ≤
   intersection[0] then
21:         if start_index[2] = end_index[2] then
22:            break
23:         end if
24:         intersection[2] ← intersection[2] + delta_time[2]
25:         start_index[2] ← start_index[2] + directions[2]
26:      end if
27:   end for
28: end function
```

4.3 Hierarchical Hash Grids

In order to optimize performance in uneven scenarios, which is a very common scenarios in computer games, we decided to use HG. Uneven scenes refers to situations in which the distribution of objects is not uniform and the size of the objects varies greatly. Theoretically, HG is well suited for this type of scenario, therefore it is also a popular spatial partitioning acceleration structure.

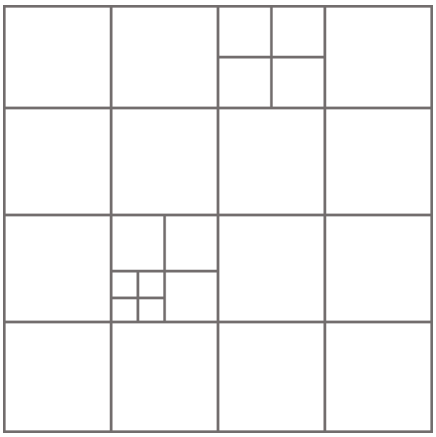

Figure 4.1: A hierarchical Hash Grid

4.3.1 Construction

Our implementation of the acceleration structure is based on the ideas of Jevans [38], with a few modifications. Their algorithm has three adjustable parameters: the density of the grid subdivision *density*, the maximum depth *max_depth*, and the maximum capacity *max_capacity* for one cell. If the number of objects in a cell exceeds the maximum capacity, the cell is subdivided until it is smaller than the maximum capacity or until the maximum depth is reached. One advantage of this algorithm is that it only stores cells that contain objects, which can save a significant amount of memory. However, our implementation includes an additional parameter, an amplification factor *amp*, which is used to calculate the density of the first layer of cells, as shown in Equation 4.2. Our implementation is a stack-based construction algorithm, in which a new stack is created to store cells that need to be subdivided, and the root nodes of all objects that have been added are placed on the stack. The stack is then processed until it is empty. During the processing, cells are subdivided and objects are stored in the appropriate new cells. If the number of objects in the new cells exceeds the maximum set capacity and the depth does not exceed the maximum depth, the new cells are added to the stack.

$$cells\_per\_dimension = (number \times amp)^{\frac{1}{3}}; \tag{4.2}$$

4.3.2 Traversal

The traversal algorithms for HG and UG are similar. When traversing cells, the traversal algorithm is called recursively if the cell being traversed contains sub-cells. This means that the AABB of the cell containing the sub-cell is treated as the new world AABB. The pseudo code for this process is shown in Algorithm 2.

4.4 Dynamic Bounding Volume Hierarchies

The DBVH algorithm is characterized by its ability to be incrementally constructed, with objects being added to an empty scene by inserting individual leaf nodes into the hierarchy. As such, the hierarchy can then be dynamically updated from frame to frame without the need for complete reconstruction. In this study, we followed the incremental construction approach presented by Catto [42], which is also used in the Box2D physics engine.

4.4.1 Insertion Algorithm

The insertion algorithm of the DBVH involves three distinct stages (1)find the best sibling, (2)create a new parent, (3)refit the bounding volumes.

1. Find the best sibling. In this stage, the algorithm considers the various nodes in the hierarchy and selects the one would be the best fit for the new object being inserted. The process of finding the best sibling is essentially a method for determining the location for inserting the object, which has a significant impact on the quality of the hierarchy. Thus, to maximize the quality of the tree, the SAH method described in Chapter 3.5.4 is commonly used. This involves minimizing the surface area of the bounding volume created by the inserted object and its sibling.

2. Create a new parent. In this stage, the selected sibling node is utilized to complete the insertion. This involves allocating a new parent node that enclose the bounding volumes of both the inserted object and the selected sibling. The new parent node takes the place of the selected sibling node within the hierarchy. Then the inserted object and its sibling become child nodes of the new parent node.

3. Refit bounding volumes. Once the object has been inserted, it is necessary to recompute the bounding volumes of the ancestor nodes of the inserted object. This process commences at the parent node of the inserted object and progresses upward through the hierarchy until reaching the root node. This step guarantees that, after the modification of the hierarchy, the bounding volume of each nodes can still accurately encompass the bounding volumes of its children.

4.5 Linear Bounding Volume Hierarchies

In contrast to the DBVH, the LBVH employs a complete reconstruction of the structure at each frame in order to maintain a high-quality hierarchy. To achieve

Algorithm 2 HG::Traverse

```
1: function TRAVERSE(cell, ray_start, ray_end, out_hit_entities)
2:    initialize the entity_hit_status, start_index[3], end_index[3],
   directions[3] intersection[3] and delta_time[3];
3:
4:    for ; ; do
5:       id ← start_index[0] × cells_per_dimension × cells_per_dimension +
   start_index[1] × cells_per_dimension + start_index[2]
6:       break if it out of the grid;
7:       if cell A contains sub-cells then
8:          TRAVERSE(A, ray_start, ray_end, out_hit_entities)
9:       else if cell A do not contain sub-cells then
10:         intersect ray with each object;
11:      end if
12:      if intersection[0] ≤ intersection[1] and intersection[0] ≤
   intersection[2] then
13:         if start_index[0] = end_index[0] then
14:            break
15:         end if
16:         intersection[0] ← intersection[0] + delta_time[0]
17:         start_index[0] ← start_index[0] + directions[0]
18:      else if intersection[1] ≤ intersection[0] and intersection[1] ≤
   intersection[2] then
19:         if start_index[1] = end_index[1] then
20:            break
21:         end if
22:         intersection[1] ← intersection[1] + delta_time[1]
23:         start_index[1] ← start_index[1] + directions[1]
24:      else if intersection[2] ≤ intersection[1] and intersection[2] ≤
   intersection[0] then
25:         if start_index[2] = end_index[2] then
26:            break
27:         end if
28:         intersection[2] ← intersection[2] + delta_time[2]
29:         start_index[2] ← start_index[2] + directions[2]
30:      end if
31:   end for
32: end function
```

a construction speed suitable for real-time applications, the LBVH utilizes a parallelized construction method that becomes faster with an increase in the number of threads. This approach was initially designed for GPUs, but has also been demonstrated to be effective on modern multi-core CPUs. Since the development of LBVH by Lauterbach, Garland, Sengupta, *et al.* [39], numerous research projects have proposed improvements to it. This study adopts the fast LBVH construction method proposed by Karras [22], who claims that the construction performance grows linearly with the number of available cores. This construction procedure can be broken down into four main steps: (1) assignment of Morton codes to the input objects, (2) sorting of objects based on their Morton codes, (3) generation of a binary radix tree using the sorted list of Morton codes, and (4) calculation of bounding boxes for each inner node of the tree. With this in mind, we can now examine the full construction algorithm in more detail.

4.5.1 Morton Codes

In LBVH, Morton codes, also known as Z-order curves, are used to sort the objects in the scene. Morton codes are a way of encoding the spatial coordinates of an object into a single integer value, such that objects that are close to each other in 3D space will have similar Morton codes. To assign Morton codes to the objects in the scene, the coordinates of the objects' centroids (the center of their bounding boxes) are first normalized to the range [0, 1], and then each coordinate is split into a set of binary digits. The digits are then interleaved to create a single Morton code for each object. The process is shown on Figure 4.2. Once the Morton codes have been assigned to the objects, they can be sorted based on their Morton codes using a sorting algorithm such as radix sort. This has the effect of placing objects that are close to each other in 3D space close to each other in the sorted list, which helps to balance the tree and reduce its overall depth.

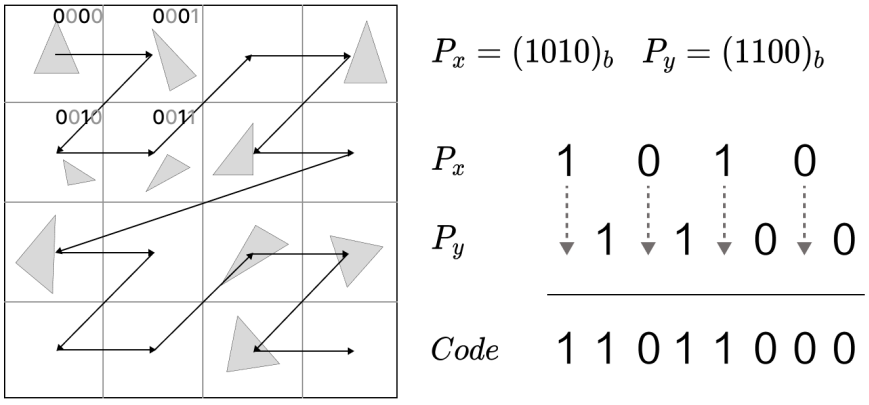

Figure 4.2: Illustrations of 4-bit Morton Codes and the encoding process for an 8-bit Morton code.

4.5.2 Binary Radix Trees

A radix tree is a tree data structure that is used to efficiently store a set of string keys. It works by organizing the keys based on their common prefixes and storing these prefixes in the tree's internal nodes, with the leaf nodes containing the keys themselves. In the context of LBVH, the keys are the binary representations of the Morton codes assigned to the objects in the scene. To construct the radix tree, the sorted list of Morton codes is recursively split into two groups at each level of the tree. The split point is determined by the highest bit that differs between the Morton codes on the left and right side of the split. This process is repeated until a leaf node is reached, resulting in a binary radix tree. Each inner node of the tree represents the common bit prefix shared by its children.

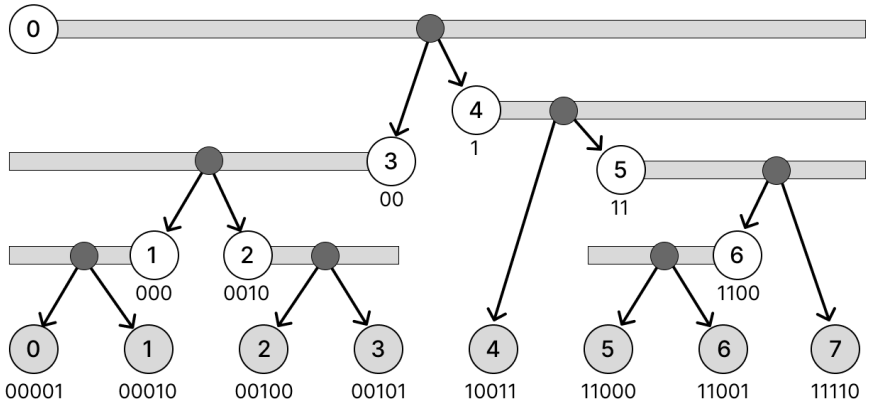

Figure 4.3: The node layout for an ordered binary radix tree

Karras and Aila [43] observed that the binary radix tree can be constructed in parallel as each node can be processed independently. As shown on Figure 4.3, an inner node corresponds to a range $[i, j]$ which covers the $i - th$ to $j - th$ leaves which have the same common bit prefix. The length of the common bit prefix of the interval $[i, j]$ can be represented as $\delta(i, j)$. Its two children covers the range of $[i, \gamma]$ and $[\gamma + 1, j]$, where γ is the split point to the interval. When building the tree, since the input list of Morton codes is ordered, for each intermediate node, two binary searches can be used to find the lower bound i, the upper bound j of its corresponding range. Then a separate binary search can be performed to find the split point γ, which determines the interval for its children nodes. Unlike the recursive method described above, this method only requires a lookup in the array of leaf nodes, and each intermediate node is processed independent of the other intermediate nodes, so the process can be parallelized for efficiency. The detail of this method is presented in Algorithm 3 [22].

4.5.3 Bounding Volume Computation

Since the hierarchy is generated totally in parallel, the computation of the bounding volumes for the inner nodes has to performed in a separate pass. Karras [22] suggested a parallel bottom-up approach to compute the bounding boxes, in which

Algorithm 3 Parallel binary radix tree construction

```
1: A_I ← Array of inner nodes of the hierarchy
2: A_L ← Array of leaves
3: N_I ← Number of inner nodes
4: for each inner node with index i ∈ [0, N_I] in parallel
5:     d ← sign(δ(i, i + 1), δ(i, i − 1))
6:     δ_min ← δ(i, i − 1)
7:     l_max ← 2
8:     while δ(i, i + l_max × d) > δ_min do        ▷ Find the upper bound of the range
9:         l_max ← l_max × 2
10:    end while
11:    l ← 0
12:    for t ← {l_max/2, l_max/4, ..., 1} do    ▷ Determine the other end of the range
13:        if δ(i, i + (l + t) × d) > δ_min then
14:            l ← l + t
15:        end if
16:    end for
17:    j ← i + l × d
18:    δ_node ← δ(i, j)
19:    s ← 0
20:    for t ← {l/2, l/4, ..., 1} do          ▷ Find the split point with binary search
21:        if δ(i, i + (s + t) × d)) > δ_node then
22:            s ← s + t
23:        end if
24:    end for
25:    γ ← i + s × d + min(d, 0)
26:    if min(i, j) = γ then
27:        A_I[i].left ← A_L[γ]
28:    else
29:        A_I[i].left ← A_I[γ]
30:    end if
31:    if max(i, j) = γ + 1 then
32:        A_I[i].right ← A_L[γ + 1]
33:    else
34:        A_I[i].right ← A_I[γ + 1]
35:    end if
36: end for
```

each thread starts from a leaf node and works its way up to the root of the tree. To calculate the bounding box of a given node, the thread looks up the bounding boxes of the node's children and computes their union, resulting in the smallest bounding box that encloses the bounding boxes of all the children. To avoid duplicate work, an atomic flag is used for each node to terminate the first thread that enters it, while allowing the second thread to pass and continue processing. This approach helps to reduce unnecessary computation by ensuring that each node is processed only once after its child nodes have been processed. The process is presented in Algorithm 4.

Algorithm 4 Bounding Volume Computation

```
1:  A_I ← Array of inner nodes of the hierarchy
2:  A_L ← Array of leaves
3:  A_F ← Array of atomic flags
4:  for each L ∈ A_L in parallel
5:      I_p ← L.parent
6:      while I_p ≠ INVALID do
7:          oldVal ← AtomicCompareAndSwap(&A_F[I_p], 0, 1)
8:          if oldVal = 0 then                ▷ check if the flag has been changed
9:              break;                            ▷ if not, terminate this thread
10:         end if                                  ▷ if changed, continue
11:         I_l, I_r ← the children of l
12:         I_p.bv ← Union(I_l.bv, I_r.bv)
13:         I_p ← I_p.parent
14:     end while
15: end for
```

4.6 BVH Traversal

In our implementations, we employed stacked-based depth-first search (DFS) as a method for traversing bounding volume hierarchies. This approach is commonly used in ray casting due to its efficiency and simplicity. Stacked-based DFS is a variation of traditional DFS that utilizes a stack to store the nodes that need to be visited rather than rely on recursive function calls. This modification serves to reduce the overhead associated with recursive function calls, particularly in large and complex hierarchies.

In a stacked-based DFS, the algorithm initiates by pushing the root node of the bounding volume hierarchy onto the stack and entering a loop. Within the loop, the top node is popped from the stack and its bounding volume is checked for intersection with the ray. If the node is a leaf, it is added to an array of hit objects. If it is not a leaf, its child nodes are pushed onto the stack. This process continues until the stack is empty.

Algorithm 5 Stack-based DFS

```
1: function RayCast(ray_start, ray_end, out_hit_entities)
2:     toVisit ← stack
3:     toVisit.push(rootNode)
4:     while toVisit.empty() ≠ true do
5:         curr ← toVisit.top()
6:         toVisit.pop()
7:         if curr.bv.intersects(ray_start, ray_end) then
8:             if curr.isLeaf() = true then
9:                 out_hit_entities.push_back(curr.uid)
10:            else
11:                toVisit.push(curr.left)
12:                toVisit.push(curr.right)
13:            end if
14:        end if
15:    end while
16: end function
```

5

Evaluation Methodology

To measure the performance of the previously mentioned acceleration structures, we introduced an analytical model and implemented a testing framework to gather the data needed for the analysis.

5.1 Performance Model

Evaluating the performance of acceleration structures can be a complex task. Many existing studies compare different structures using a benchmark of time consumption which typically depends on the algorithm implementation, hardware platform and the complexity of the scene data. To make the results more instructive, we not only benchmarked various acceleration structures, but also introduced a performance model that enables an in-depth analysis of the overhead for any specific structures. The details of the performance model will be discussed in the following sections.

5.1.1 Motivation

The performance of ray casting is often given the average time consumption per ray casting query or the number of rays that can be cast per second. However, the query speed can be affected by a number of factors. Firstly, the implementation method can have a significant impact on the performance of any algorithms. The execution speed is also depending on the computing power of the hardware platform. Additionally, the scale and construction of the test scene can also influence the query time. For example, a scene containing a large number of objects can result in a BVH with a high depth, which will require more time to traverse. Especially when the algorithm is designed to find all hit objects, rather than just the nearest one. Furthermore, the performance of ray casting in some specialized applications, such as ray tracing, is closely tied to the coherence of rays. The computation for coherent rays tends to be faster on account of the data coherence and the branch predictions in CPUs [16].

Hence, due to the abundance of complicating variables, results of a comparative experiment are typically given in the form of query time under specific experimental configurations and limitations. Even it is possible to determine which acceleration

structures run faster in certain conditions, it can not explain why they are faster and others are not. Therefore, it is of the utmost importance to develop a performance model that maps various acceleration structures to same parameters. In order that the overhead of an algorithm can be broken down into quantifiable components to identify what factors may affect the performance.

5.1.2 Cost of Acceleration Structures

The cost of a BVH can be divided into two components: ray-object intersection tests and the traversal of the hierarchy [17]. By accounting for the cost of moving data blocks from memory to registers and the cost of pre-processing computations, the overall cost can be further broken down as shown in the following equation [16]:

$$C = C'_{IT} \times N_{IT} + C'_T \times N_T + C'_R \times N'_R + C_{PREP} \tag{5.1}$$

Where C is the overall cost of the acceleration structure, C'_{IT} is the cost of ray-object intersection test, N_{IT} is the average number of intersection tests, C'_T is the expected cost of per traversal step, N_T is the average traversal steps per ray, C'_R is the cost of moving a data block from memory to registers, N'_R is the number of data block movement per ray, and C_{PREP} is the cost of pre-computation each ray. The pre-computation cost represents the overhead of initializing the traversal steps, such as locating the starting cell in a uniform grid.

The extended model can be used to analyze the performance of not only BVHs or uniform grids but also other acceleration structures. However, the cost parameters in this model are still significantly depending on the implementation methods and hardware platforms, which means a comparison regardless the implementation, computer architecture and scene data is unachievable. Nonetheless, as Havran [16] has argued, this model can abstract the algorithmic properties of published acceleration structures, to make the experimental results reproducible and verifiable. Moreover-the model also provides a new perspective to quantitatively analyze a given structure by examining the cost of each of its components, which can draw more insightful conclusions than "Algorithm A is faster than B in particular test scenes."

5.1.3 The False Positive Issue

The traversal cost term $C'_T \times N_T$ and intersection cost term $C'_{IT} \times N_{IT}$ in Equation 5.1 respectively represent the overhead of the broad phase and the narrow phase. The selection of acceleration structures does not only affect the broad phase. In some circumstances, the broad phase will produce false positive results that will increase the time consumption of the narrow phase. A false positive is an object that is considered to be hit by the ray in the broad phase but will be rejected in the narrow phase. Conversely, a true positive is an object that can pass both the broad and narrow phase. False positives may occur when objects' bounding volumes does not perfectly fit their collidable shapes (*bv-caused false positives*), or when an algorithm simply returns all the objects contained in traversal nodes that the ray hits (*traversal-caused false positive*). Traversal-caused false positives can be detected by a ray-bv

tset, which could be performed in either the broad or the narrow phase. bv-caused false positives need a thorough ray-object test, which is typically done in the narrow phase. Ideally, a broad-phase acceleration structure should only return true positives. However, the design concept of the acceleration structure and the complexity of the scene can be varied, false positives are practically unavoidable.

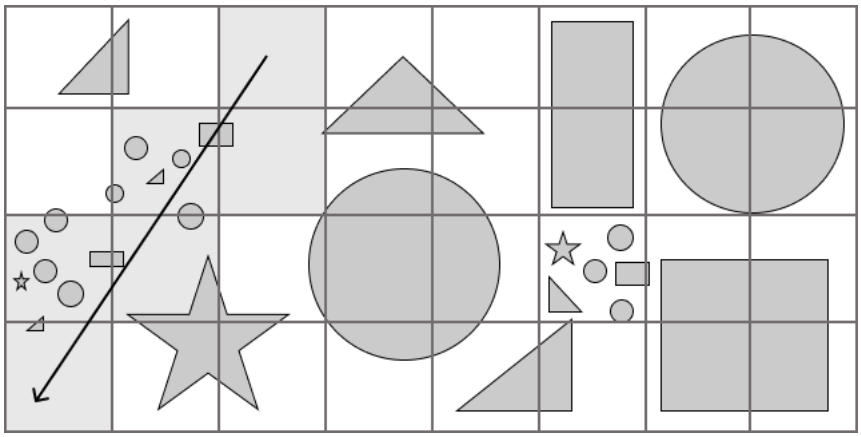

Figure 5.1: A uniform grid containing various-sized objects

Figure 5.1 demonstrates an example of the false positive issue in a uniform grid. Objects in this grid vary greatly in size. As the ray traverses the grid, cells containing a large number of small objects are hit. All these objects will be returned as the results of the broad phase, but the majority of them do not actually intersect with the ray. In this case, the narrow phase needs to spend extra time on testing the false positives that will be eventually rejected. If a finer grid is used, the traversal time will be increased but the number of false positives will be decreased, leading to a shorter overall time consumption.

5.1.4 A Practical Model

To take account of the false positive issue, the intersection test cost term $C'_{IT} \times N_{IT}$ in Equation 5.1 can be further extended as follows:

$$C_N = (C_{RO} + C_{RB}) \times (N_{TP} + N_{BFP}) + C_{RB} \times N_{TFP} \tag{5.2}$$

Where C_N is the total cost of intersection tests or the narrow phase, C_{RO} is the cost of ray-object intersection test, C_{RB} is the cost of ray-bv test, N_{TP} is the number of true positives, N_{BFP} is the number of *bv-caused false positives*, N_{TFP} is the number of *traversal-caused false positives*. True positives and *bv-caused false positives* are grouped as one term in Equation 5.2, since they have the same cost, and cannot be detected in the broad phase with a ray-bv test.

Then based on Havran [16]'s model in Equation 5.1, and with respect to the cost of narrow phase in Equation 5.2, we denote the overall cost of ray casting as Equation 5.3. Since the cost of moving data from memory to registers is highly depending on the CPU architecture and practically not measurable, this model dose not explicitly extract the $C'_R \times N'_R$ term from the intersection test cost term and the traversal

cost term.

$$C = (C_{RO} + C_{RB}) \times (N_{TP} + N_{BFP}) + C_{RB} \times N_{TFP} + C_T \times N_T + C_{PREP} \quad (5.3)$$

Given a ray casting query, the true positives should be the same regardless which acceleration structure is used. Consequently, the true positive term $(C_{RO} + C_{RB}) \times (N_{TP} + N_{BFP})$ in Equation 5.2 can be seen as constant while comparing two structures. As it is not possible to distinguish true positives and *bv-caused false positives* during the broad phase, both types of results can be treated as true positives. Consequently, the model can be further simplified as:

$$C = C_{RB} \times N_{TP} + C_{RB} \times N_{FP} + C_T \times N_T + C_{PREP} \quad (5.4)$$

For dynamic scenes, the acceleration structure should be updated as the objects changing their transforms. In that case, the cost of structure updates needs to be considered. In each frame, the structure only updates once, but the ray query may occur multiple times. Therefore, the total cost of the ray casting functionality of one frame can be represented by the cost of structure updates and the cost of N ray queries:

$$C_{total} = C_{update} + C \times N \quad (5.5)$$

This equation implies a trade-off between the update time and query time. If the overhead of building the acceleration structure is too high, the advantage of its fast query speed will be negated.

5.2 Testing Framework

In addition to implementing the acceleration structures, we have also developed a simulator to test and measure their performance. The simulator includes several commonly used acceleration structures and is based on ideas from the Broadmark system for measuring broad-phase collision detection algorithms [26]. To run the simulator, we have created a scene data generator using Unity. In the following sections, we will describe the details of the generator and simulator.

5.2.1 Data Generator

The data generator is designed to create precomputed scene data, which is used as input for the simulator. It is implemented as a Unity project, which means it includes test scenes and C# scripts that control the scene and export object and ray information frame by frame. The flowchart of the data generator is shown in Figure 5.2. Next we will discuss the implementation details.

- **Scene Modification**: In a typical game environment, there are many dynamic objects that can change over time. To simulate this, the generator modifies a certain percentage (default 1% per frame) of the objects in the scene, including adding, deleting, and moving them. This helps to test the performance of the acceleration structures under dynamic conditions.

- **Ray Cast**: This step simulates the ray casting that occurs while the game is running. To do this, a certain number of rays are generated from the camera and shot in random directions to simulate ray casting.

- **Output Result**: After completing the execution steps, the generator outputs the results every frame, including the BV and UID (unique identifier) of objects that have changed in the current frame, as well as information about the rays and the results of all ray casts.

5.2.2 Simulator

The Simulator is a critical tool that we use to measure the performance of acceleration structures. It is responsible for simulating the way that acceleration structures would operate in a real game engine, and it accurately measures the time that is spent on each execution step. This allows us to evaluate the efficiency of different acceleration structures and identify any bottlenecks or areas for improvement. The Simulator takes the test data generated by the data generator and a configuration file as input, and it produces a log file after the measurement. The log file produced by the Simulator contains detailed information about the performance of the acceleration structures during the simulation, including the time spent on each execution step and the measurable terms mentioned in the performance model discussed earlier. The flowchart of the Simulator is shown in Figure 5.2. In the following section, we will delve into the details of its implementation and discuss how it works.

- **Load Scene Data**: The simulator parses the precomputed scene file and loads the data. At the start (frame 0), it loads a large amount of object data and constructs the acceleration structure. Every frame thereafter, it only loads the data for objects that have changed and the data related to ray casting.

- **Refresh And Update**: The simulator synchronizes the scene data with the acceleration structure. This process involves inserting, removing, and moving objects. Some data structures, require the Update function to be called again to update them. In the case of LBVH, it needs to be reconstructed every frame.

- **Ray Cast Simulation**: In this step, the simulator performs the same ray cast as specified in the file and records the result. By default, it performs 100 ray casts and records the total time taken.

- **Evaluate Result**: In this step, the simulator compares the results of the ray cast previously executed with those in the file. It also measures the finer-grained terms mentioned in the performance model, such as N_TFP (number of *traversal-caused false positives*) and N_T (number of *traversals*).

The simulator currently supports 5 acceleration structures: Brute Force, LBVH, DBVH, Uniform Grids, and Hierarchical Hash Grids. It is also relatively easy to

implement and measure the performance of a new acceleration structure on the simulator. The key steps are to inherit and implement the *IAccelerator* classes, specifically the Initialize, *RefreshScene*, *Update*, and *RayCast* functions. Then, the new structure must be registered in the Accelerators.cpp file. When the new structure is selected, the Simulation.cpp file will automatically run it.

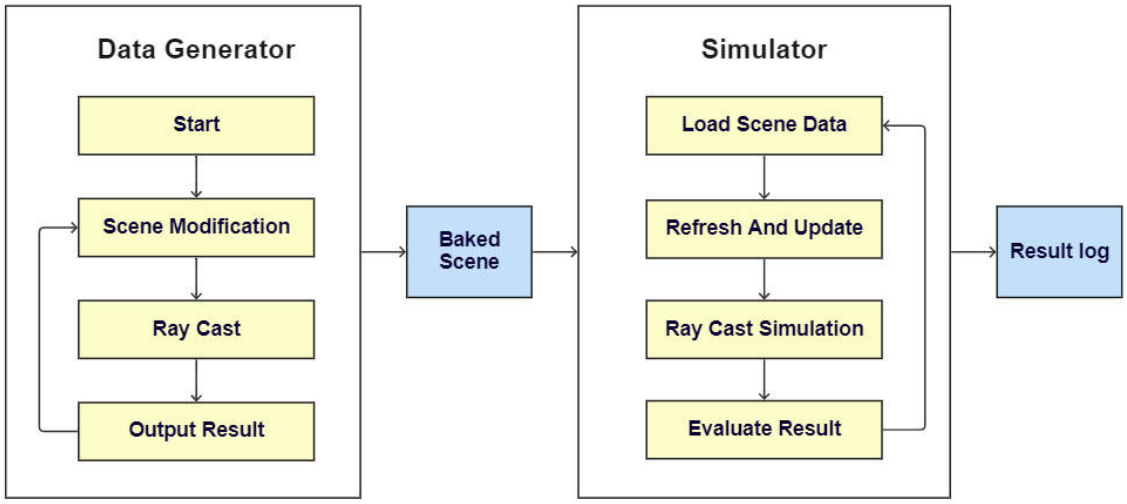

Figure 5.2: The flowchart of testing framework, including data generator and simulator. The yellow nodes mean data flow, and the blue nodes mean the input and output files.

5.2.3 Metric Measurement

Accuracy and efficiency in measurement are of utmost importance to the simulator. Therefore, it uses the `std::chrono::high_resolution_clock` from the C++ chrono library to measure time consumption. The simulator records the times at the start and end of functions and calculates the time interval between them as the time consumption. However, when the code being measured is small, i.e., the running time is too short, the measurement may be inaccurate. To reduce measurement error that arises from this reason, the simulator takes multiple executions and calculates the arithmetic mean.

6

Results

In this chapter, we first describe the experimental setup, including the hardware and scene configurations. Next, we compare the build times of the various acceleration structures. While this function is mainly called during scene loading and is not the main focus of our attention, we also compare the total simulation times for each acceleration structure in different scenes. We then selected the largest scene in each of the regular and irregular categories and evaluated their performance using the model mentioned earlier. Finally, we conducted a detailed analysis of the previous experimental results, examining the performance of each acceleration structure in both uniform and irregular scenes.

6.1 Experiment Setup

In order to evaluate the performance of the implemented acceleration structures, we conducted benchmarking on a laptop with a 3.2GHz AMD Ryzen 7 5800H CPU and 16GB of RAM running Windows 11. The structures were tested under two sets of scenes: three uniform scenes(uni-s, uni-m, uni-l) and three irregular scenes(irreg-s, irreg-m, irreg-l). The three scenes within each set contained a different number of objects: 99, 5 104, and 14 021, respectively. The uniform scenes were generated using the Unity3D engine. The objects in the uniform scene have similar sizes and are evenly distributed in space without overlapping. These scenes represent the ideal use cases. The irregular scenes were exported from the Snowdrop engine. The objects in these scenes have a skewed distribution, with varying sizes. The larger objects may occupy a significant portion of the scene while smaller objects are crowded in a small area and may be contained within the larger objects. These scenes represent the typical usage scenarios in video games.

The testing framework described in the previous chapter was utilized to test the structures. During the testing procedure, the framework first loads the baked scene file and constructs the specified acceleration structure using the recorded axis-aligned bounding boxes (AABBs) of the objects. This is referred to as *the Build Phase*. Once the structure has been constructed, the testing framework enters *the Execution Phase*, where the scene is updated with object insertion, removal, and movement. Ray casting simulations are then performed based on recorded ray information. At

each frame, a total of 1000 rays are cast, with the rays originating from the center of the scene and aiming at a random direction. This number of rays was chosen as it is close to the needs of the games developed with the Snowdrop engine, which make performance more informative in this specific scenario. Furthermore, the randomness allows to distribute the rays relatively evenly across the scene. The length of the rays is chosen to ensure complete coverage of the scene, as this more fairly reflects the effect of changes in the number of objects within the scene.

6.2 Build Phase Comparison

In the game engine, building acceleration structures is much less frequently than ray casting. Meanwhile, the build process is more tolerant of speed than ray casting, since it is often executed during the scene loading phase. The construction of the acceleration structure is only one of the many computationally intensive tasks in the game, in practice, not all CPU cores can be allocated to it. Therefore, during the build phase, all the acceleration structures were constructed with a single CPU core, including the LBVH.

	uni-s	uni-m	uni-l	irreg-s	irreg-m	irreg-l
BF	0.2614	0.2919	0.2864	0.3354	0.3351	0.2887
UG	0.3027	0.4795	1.4356	0.2713	0.6301	1.0281
HG	0.6313	4.4810	21.7787	4.5792	27.4098	31.1392
DBVH	0.3878	2.0872	6.534	0.5002	2.1168	6.1542
LBVH	1.6532	2.6962	5.0149	1.882	2.7742	4.3121

Table 6.1: Build time (in ms) of each acceleration structure in uniform and irregular scenes.

Table 6.1 compares the build time of these acceleration structures on regular and irregular scenes with different sizes. It is clear that HG takes a very long time to build on every scenes, especially as the scene size increases. This might be due to its high grid density and multi-layering. In addition, the memory footprint of HG is significantly higher than that of LBVH and DBVH, with a size of around 150M on the $irreg - l$ scene, while LBVH and DBVH only take up about 3M.

The results presented in Table 6.1 also indicate that the DBVH exhibits faster construction times for smaller scenes, whereas the LBVH performs better in large scenes. This can be attributed to the overhead incurred by DBVH's dynamic insertion strategy, which involves searching for appropriate locations within the tree for new objects. As the number of objects in the scene increases, the overhead associated with this process becomes more pronounced, leading to slower construction times in larger scenes.

	uni-s	uni-m	uni-l	irreg-s	irreg-m	irreg-l
BF	0.374	17.189	54.215	0.428	19.201	42.32
	(0.011)	(0.027)	(0.040)	(0.010)	(0.031)	(0.046)
UG	0.147	0.7	0.939	0.213	8.757	4.577
	(0.016)	(0.095)	(0.782)	(0.02)	(0.142)	(1.295)
HG	0.314	0.924	1.707	0.433	5.396	3.275
	(0.003)	(0.118)	(0.736)	(0.028)	(0.466)	(1.024)
LBVH	0.242	0.472	0.652	0.199	6.148	6.413
	(0.007)	(0.451)	(1.547)	(0.015)	(0.776)	(1.854)
DBVH	0.348	1.887	1.984	0.302	6.718	4.304
	(0.016)	(0.044)	(0.137)	(0.019)	(0.04)	(0.118)

Table 6.2: Average ray casting time (the first row) and update time (the second row) in ms.

6.3 Execution Phase Comparison

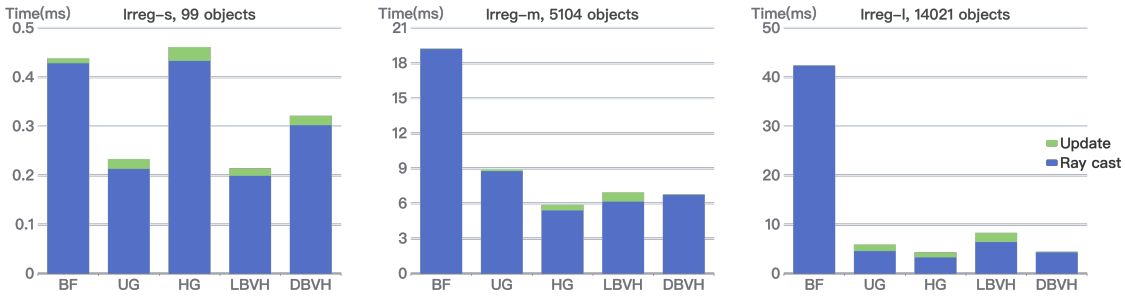

Figure 6.1: Average update and ray cast time per frame (irregular scenes).

Once constructed, the acceleration structures are ready to be used in the execution phase. During this phase, the acceleration structures were updated or rebuilt per frame as necessary, and then ray casting was performed. The performance of each acceleration structure during the execution phase is presented in Table 6.2. The execution time for ray cast that we measure does not include the narrow phase intersection, also known as the ray-object intersection. It can be seen that, in small scenes the acceleration provided by the acceleration structure may not be significant and may even have a slowing effect. As the scene size increases, the acceleration effect of the acceleration structure becomes more pronounced. In the uniform scene, the acceleration effect of UG is very pronounced and increases with the grid resolution. In irregular scenes, the performance of all types of acceleration structures is significantly decreased. However, in larger scenes, the Hierarchical Hash Grids (HG) structure still performs well. Figures 6.1 and 6.2 show the update and ray cast times, respectively, for different scenes. The graphs indicate that the LBVH structure performs well in certain scenes, but its long update times slightly decrease its overall performance.

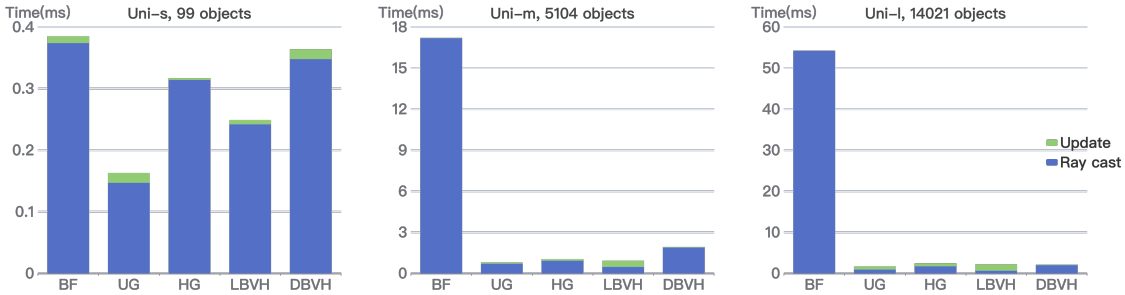

Figure 6.2: Average update and ray cast time per frame (uniform scenes).

6.4 Performance Analysis

In Section 6.3, the performance of various acceleration structures is evaluated in both uniform and irregular test scenes. In non-uniform scenes, the algorithms perform well due to the concentration of objects in small areas and the likelihood of rays being directed towards sparse areas. However, a significant increase in the number of hit objects is observed when rays are directed towards dense areas, leading to a degradation in the performance of the acceleration structure. This is a relevant consideration as games often include both open and densely populated areas. To further investigate this issue, we conducted an additional experiment in which rays were intentionally directed towards densely populated areas. Specifically, we shot rays from the center of the scene towards the object-concentrated region in the largest irregular scene ($irreg - l$), and analyzed the execution details of the different acceleration structures using the performance model previously described. This model provides a detailed analysis of the execution overhead by measuring the average cost of a single traversal step C_T and the number of traversal steps N_T, as well as the cost of ray-bounding volume (BV) tests C_{RB} and the average number of such tests N_{FP}. The measured values are presented in Table 6.3. As shown in the table, the critical factor distinguishing BVHs and grids is the cost of traversal. Traversing through a grid is faster than a BVH because the latter requires expensive ray-BV tests and checking more nodes.

	C_{total}	$C_{RB} \times N_{RB}$	$C_T \times N_T$	N_{TP}	N_{FP}	N_T	C_{RB}	C_T
BF	49288.7	49285.1	0	791	13231	0	3.5151	0
*UG	32555.3	32034.1	519.507	791	4179	92	6.4457	5.6468
HG	17133.4	16451.2	681	791	1078	98	8.8040	6.9430
DBVH	21312.8	5808.42	15502.3	791	374	3002	4.9875	5.1630
LBVH	16756.2	8722.15	8032.06	791	779	2052	5.5564	3.9136

Table 6.3: Measured parameters (in nanoseconds) of the performance model. *UG is simulated using a single-level HG.

6.4.1 Grid Resolution

The performance of the grid is determined by the grid resolution. As shown in the table, the overhead of low resolution grids is primarily concentrated in ray-bounding volume (BV) tests. Increasing the resolution of the grid can exclude more objects, thereby reducing this overhead. In order to more accurately represent this feature, we compared the performance of a *UG with that of a HG with the same grid resolution at the first level. However, the HG can continue to subdivide, resulting in more traversals. As shown in the table, while the number of traversals is higher for HG, the number of false positives N_{FP} is significantly lower. This demonstrates that, despite a small increase in traversal time, the overhead of the ray-bv tests is significantly reduced, leading to a considerable overall performance improvement.

Furthermore, we tested another resolution of the *UG in this scene. Unlike the HG, which only maintains high resolution in areas with dense objects, this *UG maintains high resolution throughout the entire scene, matching the maximum resolution of the HG. This experiment showed that, while it is possible to achieve similar or faster ray cast speeds with a high resolution *UG, the time required to update the grid is significantly higher and the memory usage considerably increases. As a result, in this scene, it would be more advantageous to use the HG due to its lower overhead and more efficient resource usage. It is worth noting that, due to the use of a hash table by HG and the direct reading of arrays by UG, there may be slight differences in their data access speeds. To minimize error due to the implementation, UG here is simulated using a single-level HG.

6.4.2 Tree Quality of BVHs

In bounding volume hierarchies, the ray casting query is accelerated by eliminating sub-trees with bounding volumes that do not intersect with the ray. BVH with higher quality should be able to eliminate more nodes in the process of traversal. According to Table 6.3, the number of visited node N_T for the DBVH is slightly higher than that for the LBVH, which indicates that the tree quality of the LVBH is also slightly better. To further explore the impact of tree quality on query speed, we replaced the incremental builder of DBVH with a full SAH builder to obtain a better tree. At this point, DBVH needs to visit fewer inner nodes than incremental construction, leading to a remarkable improvement (about 40%) in query speed (see Table 6.4).

	C_{total}	N_{TP}	N_{FP}	N_T	C_{RB}	C_T
DBVH (full-SAH)	12212	791	138	1867	3.9816	4.5596
DBVH (incremental)	21289	791	334	2836	4.9875	5.1630

Table 6.4: Comparison of two DBVH build methods

In the case of high object density, it is possible for a single ray to intersect a large number of objects, which can significantly reduce the efficiency of the BVH. Since our study concerns more on finding all hit objects in a ray cast, certain optimization

techniques based on ray travel distance, which are commonly used in closest-hit ray casting, cannot be employed. An increase in the number of leaf nodes intersected by the ray corresponds to an increase in the number of intermediate nodes that must be visited during traversal. In extreme cases, this traversal overhead can become significant. However, it is worth noting that such cases are generally rare and unlikely to occur frequently.

7

Discussion

This chapter will focus on the analysis of our experimental results in relation to the research questions. We will also discuss potential improvements to the testing framework, as well as other acceleration structures that may be worth considering.

7.1 Result Discussion

LBVH has often been thought to be less efficient, particularly in comparison to BVHs constructed with SAH-guided builders. However, our experimental results show that its query speed is even closer to that of a DBVH constructed using a full SAH builder. This can be attributed to the lower traversal overhead of LBVH, which is approximately 25% lower than that of DBVH. It is possible that the linear memory layout of LBVH contributes to this faster traversal, as it is more coherent than the unordered array used by DBVH. Additionally, our experiments only consider the broad phase, during which leaf nodes store objects rather than triangles. This results in a significantly smaller tree size compared to those constructed from millions of triangles, in which case traversal overhead may be more critical to overall performance.

It is also important to point out that because of the certain constrains of computing resources in real-world applications, we built LBVH using a single CPU core during our experimentation. This may not fully illustrate the potential benefits of LBVH's parallel building capability. In the meantime, we found that the time consumption for subsequent rebuilds is less than 50% of the initial build. We presume this is because that the first build requires the creation of corresponding leaf nodes for all objects and the computation of their Morton codes, while the update only requires to recompute Morton codes for the objects being moved.

In our implementation of LBVH, we attempted to avoid resorting the leaves at each frame by maintaining the ordered structure of the leaf array during the update of scene elements. However, our tests revealed that this approach did not result in a significant improvement in the rebuild speed of the LBVH structure. In fact, in scenarios where there was a high number of dynamic objects within the scene, this approach resulted in slower performance compared to resorting. This could be at-

tributed to the overhead incurred during the insertion of elements into an ordered array, which involves identifying the insertion position and shifting subsequent elements. It can be quite time-consuming when there is a high volume of changes in the scene. Consequently, we chose to resort the leaf array when rebuilding the LBVH.

When considering the trade-off between structure update speed and query speed, HG and DBVH may be the more suitable choices. Our target scenes frequently include irregularly distributed objects, where HG has been shown to have a superior performance. Previous experiments have also demonstrated that the quality of DBVH's hierarchical structure can be enhanced by replacing the incremental builder with an optimized SAH builder, resulting in both faster query and update speeds.

7.2 Future Work

The current implementation of the testing framework only supports axis-aligned bounding boxes (AABBs). While AABBs are intuitive to implement and fast to compute, they do have their limitations. For example, AABBs may not accurately represent the shape of an object, particularly when the object has non-uniform dimensions. In test scenes involving irregularly placed objects, using AABBs can result in the objects occupying significantly more space than their actual size, leading to reduced performance and accuracy in ray casting. An alternative is to use other forms of bounding volumes such as oriented bounding boxes (OBBs) or k-DoPs, which can fit the objects and represent the scene more accurately.

In our experiments, we evaluated the performance of LBVH and DBVH and obtained satisfactory results. However, there is still room for improvement in these algorithms, and there have been several studies proposing modifications to these methods. For example, the most time-consuming aspect of the insertion algorithm for DBVH is selecting the position to insert, which can be accelerated using a parallel search scheme [44]. As for the LBVH, optimization efforts have focused on improving both build speed and structure quality. Pantaleoni and Luebke [45] proposed a hierarchical LBVH (HLBVH) construction algorithm that significantly improves the speed of the original LBVH algorithm, achieving build times that are 2-3 times faster. Moreover, it also enables to employ SAH sweeping at the top levels of the tree can yield tree quality almost equivalent to that of a full SAH builder, while still maintaining the build speed at the same level with the original LBVH algorithm. Even though we did not implement these algorithms in our study due to certain time constraints, they represent promising areas for further investigation.

Our experiments have also shown that the performance of the HG and the UG is highly dependent on parameters such as the resolution of the grid. When the resolution is poor, both structures perform worse than the BF, but when the resolution is good, they significantly outperform other acceleration structures. As a result, there are significant benefits to using HG with different parameter configurations in different scenes. However, choosing the optimal grid resolution could also be challenging.

In addition to the resolution, HG also has parameters such as the maximum depth, the granularity of the division between different levels, and the maximum object capacity of the cell. One potential way to enhance the performance of the hierarchical grid (HG) acceleration structure is by incorporating machine learning algorithms. For example, we could use unsupervised learning to analyze the characteristics of the scene, including the number of objects within a certain range, and group the scenes into distinct categories based on these factors. By conducting regression analysis on the relationship between different parameters and execution time, we could identify the parameter values that result in the best performance for each group.

7.3 Ethical Consideration

An efficient acceleration structure can significantly improve the performance of ray casting, enabling more complex and interactive gameplay in games. This can make the game more enjoyable and potentially addictive for players. While playing games can be a source of pleasure, it is important to practice moderation and not overdo it. Prolonged video gaming can also have negative impacts on health, such as obesity and cervical spine issues due to prolonged sitting. It is important to be mindful of these risks and take appropriate precautions.

8

Conclusion

To address the research question of this project - "Which acceleration structure is the most efficient in terms of ray casting queries and updates in large physics scenes?" we begin by selecting and implementing several acceleration structures and evaluated their performance in various scenes, as shown in Table 6.2. In order to further understand the factors that increase the time taken by ray cast, we introduced a performance analysis model based on the one proposed by Havran. This model could measure the performance of acceleration structures that contain only the broad-phase, and captures the false positives N_{FP} caused by traversals. The results of this model's measurements are presented in Table 6.3. A closer analysis of the table reveals that both HG and the LBVH structure perform well when the ray is primarily directed towards areas with dense objects, but the LBVH is slower to update than the HG. Furthermore, the comparison of different construction methods for DBVH illustrates that it has the potential to attain a comparable query speed to that of LBVH, while offering a significantly faster update speed. Taking these factors into account, it can be concluded that, overall, HG and DBVH are promising choices when the irregular part of the scene becomes increasingly dominant.

Bibliography

[1] J. Clement. "Number of monthly active users on gaming platform steam worldwide from 2017 to 2021(in millions)." (2022), [Online]. Available: https://www.statista.com/statistics/733277/number-stream-dau-mau/.

[2] J. Clement. "Number of video game users worldwide from 2017 to 2027(in millions)." (2022), [Online]. Available: https://www.statista.com/statistics/748044/number-video-gamers-world/.

[3] E. Coumans, "Bullet physics simulation," in *ACM SIGGRAPH 2015 Courses*, 2015, p. 1.

[4] NVIDIA. "Nvidia physx system software." (2022), [Online]. Available: https://www.nvidia.com/en-us/drivers/physx/physx-9-19-0218-driver/.

[5] Havok. "Havok physics." (2022), [Online]. Available: https://www.havok.com/havok-physics/.

[6] M. Entertainment. "Massive entertainment." (2022), [Online]. Available: https://www.massive.se/our-studio/.

[7] M. Entertainment. "The snowdrop engine." (2022), [Online]. Available: https://www.massive.se/project/snowdrop-engine/.

[8] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*, New Orleans, Louisiana: ACM Press, 2009, p. 7, ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572771. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1572769.1572771 (visited on 08/09/2022).

[9] A. Fujimoto, T. Tanaka, and K. Iwata, "Arts: Accelerated ray-tracing system," *IEEE Computer Graphics and Applications*, vol. 6, no. 4, pp. 16–26, 1986.

[10] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.

[11] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *ACM SIGGRAPH 2006 Papers*, 2006, pp. 485–493.

[12] J. G. Cleary and G. Wyvill, "Analysis of an algorithm for fast ray tracing using uniform space subdivision," *The Visual Computer*, vol. 4, no. 2, pp. 65–83, Mar. 1988, ISSN: 0178-2789, 1432-8726. DOI: 10.1007/BF01905559. [Online]. Available: http://link.springer.com/10.1007/BF01905559 (visited on 11/01/2022).

[13] H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by a priori tree structures," in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 1980, pp. 124–133.

[14] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[15] M. Hapala and V. Havran, "Kd-tree traversal algorithms for ray tracing," in *Computer Graphics Forum*, Wiley Online Library, vol. 30, 2011, pp. 199–213.

[16] V. Havran, "About the relation between spatial subdivisions and object hierarchies used in ray tracing," in *Proceedings of the 23rd Spring Conference on Computer Graphics - SCCG '07*, Budmerice, Slovakia: ACM Press, 2007, pp. 43–48, ISBN: 978-1-60558-956-5. DOI: 10.1145/2614348.2614355. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2614348.2614355 (visited on 10/11/2022).

[17] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," *ACM SIGGRAPH computer graphics*, vol. 20, no. 4, pp. 269–278, 1986.

[18] B. Smits, "Efficiency issues for ray tracing," in *ACM SIGGRAPH 2005 Courses*, 2005, 6–es.

[19] J. Goldsmith and J. Salmon, "Automatic creation of object hierarchies for ray tracing," *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, May 1987, Conference Name: IEEE Computer Graphics and Applications, ISSN: 1558-1756. DOI: 10.1109/MCG.1987.276983.

[20] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, no. 3, pp. 153–166, May 1990, ISSN: 0178-2789, 1432-8726. DOI: 10.1007/BF01911006. [Online]. Available: http://link.springer.com/10.1007/BF01911006 (visited on 11/15/2022).

[21] C. Apetrei, "Fast and simple agglomerative LBVH construction," *Computer Graphics and Visual Computing (CGVC)*, 4 pages, 2014, Artwork Size: 4 pages ISBN: 9783905674705 Publisher: The Eurographics Association. DOI: 10.2312/CGVC.20141206. [Online]. Available: http://diglib.eg.org/handle/10.2312/cgvc.20141206.041-044 (visited on 11/15/2022).

[22] T. 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*, 2012, pp. 33–37.

[23] J. Bittner, M. Hapala, and V. Havran, "Incremental bvh construction for ray tracing," *Computers & Graphics*, vol. 47, pp. 135–144, 2015, ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2014.12.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849314001435.

[24] L. Szirmay-Kalos, V. Havran, B. Balázs, and L. Szécsi, "On the efficiency of ray-shooting acceleration schemes," in *Proceedings of the 18th spring conference on Computer graphics - SCCG '02*, Budmerice, Slovakia: ACM Press, 2002, p. 97, ISBN: 978-1-58113-608-1. DOI: 10.1145/584458.584475. [Online]. Available: http://portal.acm.org/citation.cfm?doid=584458.584475 (visited on 11/04/2022).

[25] V. Havran, "Heuristic ray shooting algorithms," Ph.D. dissertation, 2000.

[26] Y. R. Serpa and M. A. F. Rodrigues, "Broadmark: A testing framework for broadphase collision detection algorithms," *Computer Graphics Forum*, vol. 39, no. 1, pp. 436–449, Feb. 2020, ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.13884. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/cgf.13884 (visited on 08/09/2022).

[27] J. Gregory, *Game engine architecture*. AK Peters/CRC Press, 2018.

[28] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '90, New York, NY, USA: Association for Computing Machinery, 1990, pp. 322–331, ISBN: 978-0-89791-365-2. DOI: 10.1145/93597.98741. [Online]. Available: https://doi.org/10.1145/93597.98741 (visited on 11/02/2022).

[29] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: A hierarchical structure for rapid interference detection," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 171–180.

[30] M. Held, "Erita collection of efficient and reliable intersection tests," *Journal of Graphics Tools*, vol. 2, no. 4, pp. 25–44, 1997.

[31] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha, "Spherical shell: A higher order bounding volume for fast proximity queries," in *Robotics: The Algorithmic Perspective: 1998 Workshop on the Algorithmic Foundations of Robotics*, 1998, pp. 177–190.

[32] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and R. Rowe, "Collision detection: A survey," in *2007 IEEE International Conference on Systems, Man and Cybernetics*, Montreal, QC, Canada: IEEE, Oct. 2007, pp. 4046–4051, ISBN: 978-1-4244-0990-7 978-1-4244-0991-4. DOI: 10.1109/ICSMC.2007.4414258. [Online]. Available: http://ieeexplore.ieee.org/document/4414258/ (visited on 09/12/2022).

[33] C. Ericson, *Real-time collision detection*. Crc Press, 2004.

[34] J. Amanatides, A. Woo, *et al.*, "A fast voxel traversal algorithm for ray tracing.," in *Eurographics*, vol. 87, 1987, pp. 3–10.

[35] F. Schornbaum, "Hierarchical hash grids for coarse collision detection," *Student Thesis, University of Erlangen-Nuremberg*, 2009.

[36] F. Cazals, G. Drettakis, and C. Puech, "Filtering, clustering and hierarchy construction: A new solution for ray-tracing complex scenes," in *Computer graphics forum*, Wiley Online Library, vol. 14, 1995, pp. 371–382.

[37] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley, "Interactive ray tracing for volume visualization," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05, New York, NY, USA: Association for Computing Machinery, 2005, 15–es, ISBN: 978-1-4503-7833-8. DOI: 10.1145/1198555.1198754. [Online]. Available: https://doi.org/10.1145/1198555.1198754 (visited on 11/15/2022).

[38] D. A. Jevans, *Adaptive voxel subdivision for ray tracing*. University of Calgary, 1990.

[39] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," in *Computer Graphics Forum*, Wiley Online Library, vol. 28, 2009, pp. 375–384.

[40] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," *Computer Graphics Forum*, vol. 40, no. 2, pp. 683–712, May 2021, ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.142662. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/cgf.142662 (visited on 11/29/2022).

[41] I. Wald, W. R. Mark, J. Günther, *et al.*, "State of the art in ray tracing animated scenes," *Computer Graphics Forum*, vol. 28, no. 6, pp. 1691–1722, Sep. 2009, ISSN: 01677055, 14678659. DOI: 10.1111/j.1467-8659.2008.01313.x. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2008.01313.x (visited on 11/14/2022).

[42] E. Catto, *Math for game developers: Dynamic bounding volume hierarchies*. [Online]. Available: https://www.gdcvault.com/play/1026269/Math-for-Game-Developers-Dynamic.

[43] T. Karras and T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 89–99.

[44] J. Bittner, M. Hapala, and V. Havran, "Incremental bvh construction for ray tracing," *Computers & Graphics*, vol. 47, pp. 135–144, 2015.

[45] J. Pantaleoni and D. Luebke, "Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry," in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 87–95.