ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Frantál** Jméno: **Petr** Osobní číslo: **420056**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Metody pro vrhání paprsku s hierarchiemi obálek na CPU

Název diplomové práce anglicky:

Ray Tracing with Bounding Volume Hierarchies on a CPU

Pokyny pro vypracování:

Prostudujte existující metody pro stavbu hierarchie obálek (BVH) pomocí metod shora dolů, zdola nahoru a vkládáním a různé globální optimalizace zlepšující vlastnosti hierarchií. Vybraných šest algoritmů po dohodě s vedoucím práce implementujte na CPU v jazyce C++ a ověřte jejich správnost pomocí vhodných metrik, vizualizace datových struktur a metody vrhání paprsku. Pro urychlení algoritmu použijte po dohodě s vedoucím práce metodu paralelizace pomocí vláken nebo výpočtu SIMD, jako jsou instrukce SSE, AVX atd. Implementované algoritmy otestujte na sadě alespoň patnácti různých scén různé velikosti a charakteru a vlastnosti implementovaných algoritmů porovnejte (rychlost stavby datové struktury, časová a paměťová složitost, atd.) .
Testovací data dodá zadavatel práce.

Seznam doporučené literatury:

1) WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. ACM Transactions on Graphics 26, 1 (2007).
2) KENSLER A.: Tree Rotations for Improving Bounding Volume Hierarchies. In Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing (Aug 2008).
3) WALTER B., BALA K., KULKARNI M., PINGALI K.: Fast Agglomerative Clustering for Rendering. In IEEE Symposium on Interactive Ray Tracing (RT) (August 2008).
4) V. Havran, R. Herzog, H.-P. Seidel: 'On the Fast Construction of Spatial Hierarchies for Ray Tracing', at RT06 conference', September 2006, pages 71-80.
5) J. Bittner, M. Hapala, V. Havran: 'Fast Insertion Based Optimization of Bounding Volume Hierarchies', in journal Computer Graphics Forum, Volume 32, Issue 1, pages 85-100, February 2013, DOI: 10.1111/cgf.12000
6) M. Vinkler, J. Bittner, V. Havran: 'Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction', in conference High Performance Graphics 2017, Los Angeles, USA, July 28-30, 2017.
Další literaturu dodá vedoucí práce.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Vlastimil Havran, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **28.06.2018** Termín odevzdání diplomové práce: **08.01.2019**

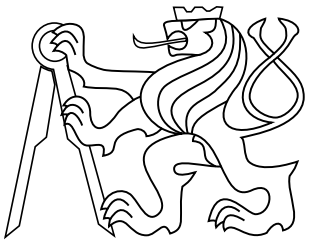
Platnost zadání diplomové práce: **30.09.2019**

doc. Ing. Vlastimil Havran, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

master's thesis

Ray Tracing with Bounding Volume Hierarchies on a CPU

Bc. Petr Frantál



January 2019

Supervisor: prof. Ing. Vlastimil Havran, Ph.D.

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

Acknowledgement

My sincere gratitude goes to prof. Ing. Vlastimil Havran, Ph.D., for introducing me and other students to the topic of acceleration structures for ray tracing. I would like to thank him for the supervision of my thesis, the inspiration and for his many valuable advices which helped me through the work on this thesis. I would also like to thank him for providing me some of the testing scenes.

I would like to thank Ing. Daniel Meister, Ph.D., for providing me the San Miguel scene.

My next thanks goes to my alma mater, Czech Technical University in Prague, and to Department of Computer Graphics and Interaction and to people that work there. They have introduced me to many interesting areas of computer science, for which I am very grateful.

I would also like to thank my parents and my brother for supporting me throughout the studies.

Last but not least, I would like to thank my girlfriend for all her support and love.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

Prague, January 8, 2019 .....

Abstract

V této diplomové práci se zaměřujeme na možnosti stavby jedné z akceleračních struktur pro metodu sledování paprsku, hierarchie obálek, na CPU. Práce nejprve přináší úvod do problematiky a shrnutí používaných metod. Přinášíme také kategorizaci metod vytvořenou na tomto základě. V další části práce jsme vybrali 7 metod dosahujících dobrých výsledků a napříč kategoriemi. Tyto metody jsme následně reimplementovali v jazyce C++ ve frameworku pro metodu sledování paprsku nanoGOLEM. Pro urychlení stavby jedné z metod jsme použili vlákna ze standardu C++11. Metody jsme otestovali na sadě 20 testovacích scén různé velikosti a typu, které se běžně používají v článcích zabývajících se akceleračními strukturami pro sledování paprsku. V další části práce přinášíme výsledky naší implementace a porovnání algoritmů. Část uvedených metrik je nezávislá na míře optimalizace, a proto může být použita pro porovnání s jinými metodami či pro budoucí verifikaci těchto metod v jiných implementacích. Dále také přinášíme návrhy o využití testovaných algoritmů.

Klíčová slova

Sledování paprsku, akcelerační struktury, hierarchie obálek

Abstract

In this diploma thesis we focus on the possibilities of one of the acceleration structures for ray tracing, the bounding volume hierarchy. First we provide an introduction to the topics and the review of the state of the art. We also provide a categorization of the methods based on this. In the next part of our work we chose 7 promising methods and reimplemented them in C++ into the nanoGOLEM ray tracing framework. For the acceleration of one of the methods we used the threads from the C++11 standard. We tested the implemented algorithms on a set of 20 testing scenes of different type and complexity, which are often used in the scientific papers related to acceleration structures for ray tracing. In the next part of our thesis we present the results of the implemented algorithms and their comparison. Some of the reported metrics are also independent on the degree of optimization and thus can be used for the comparison with other methods or for future verification in other implementations. Furthermore, we present suggestions about the usage of the implemented algorithms.

Keywords

Ray Tracing, Acceleration Structures, Bounding Volume Hierarchy

Contents

1 Introduction

When considering the most used rendering methods in contemporary computer graphics, apart from the photo-realistic rendering methods and others, the rasterization and the ray tracing come to mind. The rasterization still has the undisputable place in real-time rendering because of its speed. The speed is implied by the simplicity of the method, because in the very core the method does not work with the interactions of the objects in the scene (with the except of the visibility computation). Because of that it is also simple to implement in both software applications and computer hardware. However, the absence of the interactions is also the disadvantage of the method, as they (e.g. shadows) need to be computed separately in order to provide more realistic results.

Ray tracing, on the other hand, is a global illumination method and evaluates the interactions between the scene objects, namely the mentioned shadows, reflections and refractions and also implicitly evaluates the visibility. While the main principle of the ray tracing is also straight-forward, because of the interactions and required recursive computations the method is computationally expensive. Until recently, the hardware support for the ray tracing was not as high as for the rasterization. While ray tracing could be exploited in offline renderers, it could not be fully used in one of the most important computer graphics applications, the games. This seems to change with the recent introduction of the Nvidia RTX graphic processing unit series with Turing architecture that aims to bring the hardware support for the ray tracing to this part of the commercial sector [18].

It is beyond the scope of this thesis to discuss or predict whether or when such GPUs will become the standard component of everyone's personal computer. Without the extensive hardware support, however, the computational expensiveness of the method stands and affects both real-time and other applications. The method cannot achieve sufficiently low rendering times without preprocessing the scene to be rendered. This phase consists of building an acceleration structure that stores the scene primitives (e.g. triangles) and thus performs the spatial sorting of these. The acceleration structure then provides a more convenient data arrangement for subsequent rendering. As the rendering part of the application renders the resulting image, it traverses the more convenient acceleration structure instead of the former plain primitive list. This allows to save computations because of considering only an amount of the primitives really relevant to a certain part of the rendering (this amount differs based on the acceleration structure used) and also by determining the relevant primitives fast. As a result, the rendering times can be reduced significantly.

During approximately thirty past years numerous acceleration structures have been developed and examined. The most promising of them in terms of rendering times are grids, k-d trees and bounding volume hierarchies (often abbreviated as BVHs). Each of these structures have strengths and weaknesses of their own. The grids are very fast to construct and also provide sufficiently fast rendering times for scene with uniform primitive distribution. When this condition does not hold, however, the rendering times become rather high. This is not true for both k-d trees and bounding volume hierarchies, which adapt to these scenes well. Both of the structures yield sufficiently

low rendering times for both mentioned kinds of the primitive distribution.

The k-d trees have gone through an extensive research, in which fast construction algorithms, methods for animated scenes, GPU algorithms and other topics have been examined. The situation of bounding volume hierarchies is similar, but the BVHs have certain convenient attributes that the k-d trees do not.

In bounding volume hierarchies, a primitive can be stored in exactly one node. This is different from k-d trees, where a primitive straddling the splitting plane is placed into two child nodes defined by the plane. We assume the primitives to be stored in the leaves of the acceleration structures only. Having n primitives, the bounding volume hierarchy can have n leaves at most, while this does not hold for k-d trees (some of the BVHs relax on the rule of storing a primitive in exactly one leaf and these can have more leaves, such as the Split BVH proposed by Stich et al. [SFD09], but we will not consider them now). Having n leaves, a binary BVH has at most $n-1$ inner nodes and $2n-1$ nodes in total. The bounding volume hierarchies therefore have limited memory complexity. This potentially does not hold for k-d trees. This also implies that when knowing the exact number of primitives stored per leaf, we can preallocate the memory used for hierarchy even before construction.

There is also analogy between the bounding volume hierarchies and other principles and algorithms in the computer science. By storing the primitives the acceleration structures perform spatial sorting. Unlike k-d trees, which are built top-down, there are more approaches to construct the BVHs. Three of these approaches are top-down, bottom-up and by insertion. These are in fact analogy to sorting algorithms, top-down being analogy to quicksort, bottom-up to mergesort and the insertion principle to insertionsort. In terms of graph theory, the bounding volume hierarchy is a tree (binary or with higher arity), in which the bounding volume of an inner node fully contains the bounding volumes of it's children. The tree rotations can be applied to BVHs, restructuring the tree and updating the effected bounding volumes, yielding another correct hierarchy.

The hierarchies are also able to refit the stored bounding volumes. The refitting is an update technique which can be exploited when rendering dynamic scenes. When a part of the dynamic scene (e.g. a computer game) moves or changes size, it may be unnecessary to construct the whole hierarchy from scratch. Instead, the respective nodes of the hierarchy can be repositioned or their bounding boxes can be resized. This approach can be faster than the new construction.

As mentioned before, there are more construction approaches for the BVHs. By applying tree rotations or different techniques, the hierarchies can be also subjects to optimization, aiming to reconstruct the hierarchy into one with higher quality which could provide lower rendering times. Each of these approaches imply a single category of the bounding volume hierarchy methods. More algorithms with different properties have been proposed in each of these categories. These algorithms differ in the resulting hierarchy quality, construction and rendering times, memory complexity and other metrics. It makes sense to examine the differences between the BVH methods.

The algorithms were also proposed in different years, implemented in various frameworks and when reporting the results, not the same metrics were always used, which make the algorithms difficult or even unable to compare. Unfortunately, the construction and rendering times, which are reported in all the related works, are not entirely sufficient for the comparison of the algorithms, because they are dependent on several properties, such as the degree of optimization, parallelization, vectorization, computer hardware, operating system, time measurement approach and others.

It is of course beyond the scope of this thesis to compare each of the algorithms,

instead we aimed to choose the ones with interesting properties. We aimed to choose rather the algorithms proposed in recent years, which are reported to be state of the art methods, construct high-quality hierarchies, have low construction and rendering times and are able to run parallely on multi-core CPUs. We reimplemented all the chosen algorithms from scratch in a single ray tracing framework and evaluated each of them on the same scenes and using the same metrics. We believe that the comparison can bring new suggestions about the use of these algorithms.

1.1 Aims of the thesis

One of the aims of the thesis is to bring a review of the construction, optimization and related bounding volume hierarchy methods and some of the algorithms. The main aim of the thesis is to provide a thorough comparison of several chosen construction and optimization algorithms. The comparison is aimed to be repeatable and we therefore report metrics independent on optimization or computer hardware.

1.2 Thesis structure

The thesis is organized into six chapters. Chapter 2 states the basic principles in bounding volume hierarchy algorithms and provides the categorization of the construction algorithms and the related techniques as well as the review of the state of the art. Chapter 3 describes the algorithms chosen for the implementation and provides the design details. Chapter 4 brings the implementation details as well as the description of the verification, testing and comparison of the algorithms. Chapter 5 then brings the achieved results. The thesis concludes in chapter 6. There are also two appendices included in the thesis. In the appendix A we present rest of the measured results. In the appendix B we summarize the content of the provided DVD.

2 Theoretical background

This chapter starts by recalling the basic knowledge about the ray tracing method, the acceleration structures and the surface area heuristic used in the construction. It continues by focusing on the acceleration structures examined in this theses, the bounding volume hierarchies, and presents the categorization of the algorithms related to these. A review of the state of the art in bounding volume hierarchy methods is provided after that.

2.1 Ray tracing

Ray tracing is a fundamental rendering method in computer graphics. It is a global illumination method and as such it allows to take light transport interactions between scene primitives into account when computing scene illumination. The advanced form of the ray tracing was proposed by Whitted in 1980 [Whi80] and the method is very well known, so we will only briefly recall it and focus on the facts related to our thesis.

Ray tracing is based on casting rays, which are used to retrieve the information about the illumination in the scene. The method starts by casting the rays from the location of the camera center through the pixels in the image plane into the scene. In the basic form, a single ray is cast through each pixel, but more rays can be cast in order to apply anti-aliasing. A ray will subsequently return the information about the illumination (i.e. color) visible from the camera through the assumed pixel.

The ray cast from the camera either does not intersect any primitive in the scene, in which case predefined background color is used as a result or the color can be retrieved from the environment map, or it does intersect a scene primitive. In that case illumination contributions from all light sources visible from the location of intersection are calculated and summed. To determine whether a light source is visible from the intersection location, another ray is cast, this time from the location of intersection to the light (for simplicity we now assume only point lights). The light is visible if there are no occluding primitives between it and the intersection. To determine the visibility, it is sufficient to find only one occluder (and stop the search upon finding it).

After the calculation of the illumination caused by the visible light sources, the method recurses by casting another types of rays. These will eventually return the information about the illumination visible in the former intersection point from certain directions. These contributions will be incorporated in the result calculated so far (which will be then returned). The process to calculate these contributions is exactly the same as the one we just described (including recursion). The new rays are:

- A reflected ray, which is created by reflecting the former ray in the intersection point according to the surface of the intersected primitive. The reflected ray is calculated as the ideal mirror reflection of the incoming ray.
- A transmitted (or refracted) ray, which is created from the former ray according to the Snell's law.

The contributions from the reflected and transmitted rays are incorporated based on the coefficients describing the reflectivity and transitivity of the intersected sur-

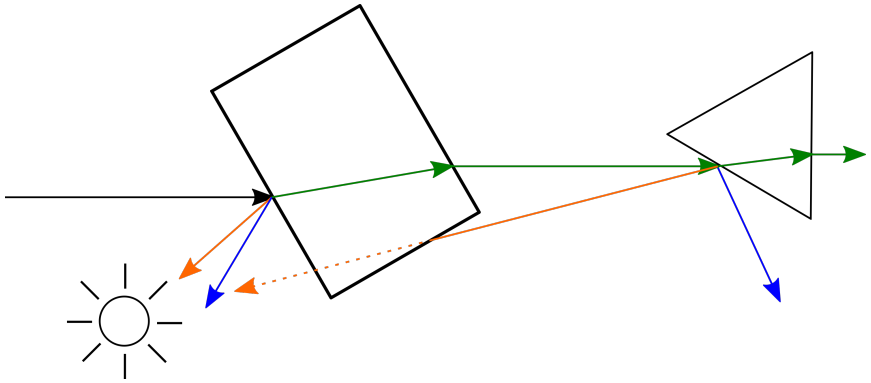


Figure 1 An illustration of the ray tracing principle. We can see various types of rays. The rays are illustrated using the arrows. The black ray is the primary ray, the blue rays are the reflected ones, the transmitted rays are drawn in green and the shadow rays are drawn in orange. The sun represents a point light source and the other shapes represent scene primitives. We can see that the intersection point on the triangle is occluded by the rectangle.

face. The principle described above repeats for the new rays. This could potentially repeat indefinitely, as the new rays would intersect objects, create new rays and so on. We can think of the ray shooting as of a recursive principle and therefore certain depth-of-recursion threshold is used to end the calculations eventually. The calculated illumination is then returned from the deepest level of the recursion and incorporated into the results of the previous level. This repeats, until we have the result for the first level.

The rays cast from the camera through the image plane are called primary rays, the rays used to determine the visibility of lights from the intersection points are called shadow rays and the other rays (from all the recursive calls except the other shadow rays) secondary rays. In each intersection point the same number of shadow rays as the number of light sources is created, joined by the two more secondary rays. All the types of the rays can be seen in figure 1 depicting a simple scene.

Assuming only the primary rays, the time complexity of finding the nearest primitive for each ray is:

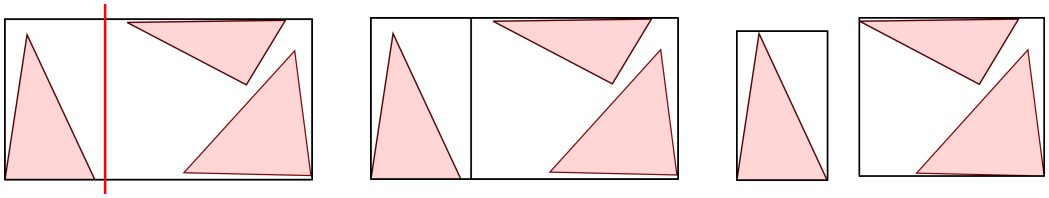
$$O(w \cdot h \cdot n) \tag{1}$$

where w is the number of pixels in the row of the image plane, h is the number of pixels in the column and n is the number of primitives in the scene.

In our thesis we focus on this complexity rather than on the illumination calculations themselves. The complexity is rather high, making the ray tracing computationally expensive. Analyzing the complexity, we are not able to reduce terms w and h describing the number of pixels (unless would trace more rays at once using the ray packet tracing). What we are able to reduce is term representing the number of primitives n that each ray must tests in order to find the closest one to the camera. We can achieve this by using auxiliary acceleration structures. We describe the fundamentals of these in the next section.

2.2 Acceleration structures

If we examine the complexity of determining the closest intersection for primary rays, $O(w \cdot h \cdot n)$, it is rather high. In order to lower it, the ray tracing applications exploit ef-



a) The former cell.

b) Spatial subdivision (k-d tree).

c) Object hierarchy (BVH).

Figure 2 The example of the partition of a cell. In the left image we can see the former cell and a splitting plane drawn in red. On the other two images we can see the new child cells. Their volume differs based on the type of the acceleration structure. The type is stated below each image.

ficient acceleration structures. These allow to decrease the third term in the complexity formula. There are also other ways to increase the efficiency of ray tracing, e.g. using faster ray-primitive intersection calculations or tracing packets of rays at once instead of a single ray. We will omit the more efficient intersection calculations in this thesis and mention some techniques of tracing packets of rays in the state of the art section, 2.4 and we will focus on the general description of the acceleration structures now.

The purpose of an acceleration structure is to spatially sort the primitives (or objects in general) in the scene. The sorted order, stored in the acceleration structure, is required to be more convenient for the ray tracing application than the original, unsorted sequence (e.g. the polygon soup stored in a list). At first, the acceleration structures can be categorized as regular and hierarchical. The regular structures (such as the uniform grid) partition the whole scene uniformly, while the hierarchical provide an adaptive partitioning. At the second level, the acceleration structures can be categorized as spatial subdivisions or object hierarchies. The cells of both types represent a volume (a part of the scene) and can refer the primitives associated with the volume. The difference between them is in the construction of the cells.

Assume we have a cell representing a certain volume and referring a set of primitives. Using the spatial subdivision, we partition the volume (e.g. by defining a splitting plane), which define volumes for new cells. The primitives referred in the former cell are then placed into the new cell (or cells) based on the volume (volumes) they belong to. The construction of cells is volume-driven. Using the object hierarchy, the set of primitives is partitioned in a convenient way and each subset implies a new cell. The volumes of the new cells are then determined from the contained objects. The construction is thus object-driven. The examples of the spatial subdivisions are k-d trees, octrees and grids, and the example of the object hierarchies is the bounding volume hierarchy, which is the topic of this thesis. The popular structures for ray tracing are the grids (in a limited way, will be described below) and k-d trees and BVHs (in a more extensive way). The example of partitioning a cell in spatial subdivision and object hierarchy can be seen o figure 2.

The grids can be constructed fast in $O(n)$, where n is the number of primitives in the scene. For tracing rays, they perform well when the primitive distribution in the scene is uniform. However, when this is not the case, their performance is rather inefficient. The k-d trees and BVHs are thus more popular. Both of them allow for high performance of tracing rays. In the next section, we focus on the bounding volume hierarchies and describe the differences between them and k-d trees.

2.2.1 Bounding Volume Hierarchies, differences to k-d trees

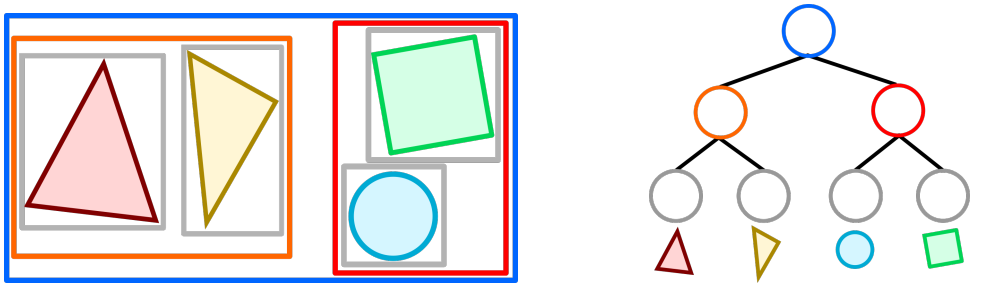
The BVH is a hierarchical acceleration structure, more specifically a tree. The arity (also called branching factor) of a BVH can be two or more. A node of a BVH stores the bounding volume of its domain. The parent node of two or more child nodes always tightly encloses the children's bounding volumes in its own. Though different bounding volumes can be exploited in the hierarchies, e.g. in the work of Weghorst et al.[WHG84], the axis-aligned bounding boxes (AABBs) are usually used. They enclose the contained primitives or nodes sufficiently tight and still provide for fast intersection calculations. Such bounding box stores the maximum and minimum coordinates in all three axes (these vectors can be also viewed as two corners of the box). When representing the bounding box in the computer memory using C++, the usage of the single precision floating point numbers leads to the memory complexity of 24B (six floats, each has 4B). Although it is usually sufficient to use the bounding boxes in the inner nodes only, the BVHs have still larger memory requirements per single node than the k-d trees. The inner BVH node typically has 32B and the k-d tree one can have 8B using several optimizations. On the other hand, the BVHs usually have a more limited number of nodes.

Similar to the k-d trees, the BVHs can be constructed in the top-down manner. They can also be constructed using different approaches, which we examine in detail in the categorization section 2.3. In the top-down construction, the BVHs also use the splitting plane such as k-d trees do. However, they treat differently the objects that straddle the plane. K-d trees place such primitives into both new child nodes (assuming the arity of 2 for both BVHs and k-d trees). This leads to (primitive) reference duplication and possibly unlimited memory demands for k-d trees, therefore another termination criteria must be used in contrast to BVHs, such as the maximum k-d tree depth. BVHs, on the other hand, usually place the considered primitive into exactly one of the new child nodes, thus leading to predictable memory complexity. In particular, when knowing the exact or minimum number of primitives contained in each hierarchy leaf, the whole BVH can be allocated before the construction itself and only updated during it (by defining the links between the nodes).

The bounding volumes of the sibling nodes in the BVHs can overlap each other, whereas in the k-d trees do not. This impacts the design of the traversal algorithm. In k-d trees, we can traverse to the leaves nearest to the origin of a ray and after finding the closest intersection in the first leaf, we can terminate the traversal because we know that this intersection is the closest in the whole tree. This is not possible in BVHs. Similarly, we can traverse the tree structure to the nearest leaves, but we cannot terminate upon finding an intersection. We must always finish traversing the subtree (rooted in a certain inner node) which is intersected by the ray (we do not always need to traverse all the branches, because we always test, whether the ray intersects them). This is because the bounding volumes of the subtrees can overlap.

Both of the data structures can be also constructed in $O(n \log n)$ (an efficient construction scheme for k-d trees is presented in the work of Wald and Havran [WH06] and the BVH scheme in the work of Ganestam et al. [Gan+15]). The advantages of bounding volume hierarchies are the predictable memory complexity, more construction approaches and possible topology optimizations (these are implied by the analogy of the BVHs to different algorithms in computer science) and update techniques applicable for dynamic scenes (such as refitting).

An example of simple bounding volume hierarchy storing 4 primitives can be seen on figure 3.



a) The scene containing primitives and bounding volumes of the BVH.

b) The tree structure of BVH corresponding to the scene.

Figure 3 An illustration of a simple BVH. We can see the scene containing primitives and bounding volumes of the BVH (left). The colors of the volumes correspond to the inner nodes of the BVH (right).

2.2.2 Surface Area Heuristic

The surface area heuristic (abbreviated SAH) is used in many BVH and k-d tree algorithms. It allows to predict the ray tracing performance provided by the structures and has a significant impact on their quality. The ideas behind the SAH were first described in the work of Goldsmith and Salmon [GS87] and the SAH was later formalized by MacDonald and Booth [MB90]. The SAH makes few assumptions about the scene. These assumptions usually do not hold in practice, but the results of using the heuristic are still good. The assumptions made by the SAH are:

- The distribution of rays is uniform.
- A ray cast through the scene does not hit any primitive (object).

By assuming the previous, the heuristic can describe the probability of a ray hitting the acceleration structure node using surface area. When used in a top-down construction of the k-d trees and the BVHs, the SAH makes one more assumption: the examined node is to be split only once (i.e. the node is split, therefore defined as inner node, two leaves are created and then the top-down construction in both new branches is terminated), which is usually not the case, but the SAH-based construction leads to good results even in the top-down construction.

The formulas exploiting the surface area heuristic also exploit the implementation-specific constants. These are:

- c_T - Traversal constant. Describes the expected cost of a ray traversing the inner node of acceleration structure.
- c_I - Intersection constant. Describes the expected cost of a ray intersecting a primitive in a leaf.

During the subdivision in a top-down construction, we can describe convenience of a subdivision candidate by comparing the expected cost of the possible inner node N (after considered subdivision):

$$C(N) = c_T + \frac{SA(left(N)) \cdot C(left(N)) + SA(right(N)) \cdot C(right(N))}{SA(N)} \tag{2}$$

In the equation the $left(N)$ denotes the left child of the node N, $right(N)$ denotes the right child and $SA(N)$ denotes the surface area of the bounding box of node N. We can also describe the cost of not subdividing the node N, thus defining it as a leaf:

$$C(N) = c_I \cdot t_N \tag{3}$$

In this equation t_N is the number of primitives in the leaf N.

As mentioned, the SAH can be exploited in the top-down construction of BVHs and k-d trees this way. We can use it to find a best partition candidate or to determine whether the subdivision is even convenient (if the expected cost of a leaf is lower than the one of subdivided inner node, the construction in the respective branch is terminated is many algorithms).

Apart from the top-down construction, the SAH is exploited in the BVH compaction algorithm, described in section 3.4.4 and also implemented in this thesis, and to calculate the cost of the whole hierarchy using the formula, which is obtained by derivation the two formulas above:

$$C(T) = \frac{1}{SA(T)} \cdot \left(c_T \cdot \sum_{N \in innernodes} SA(N) + c_I \cdot \sum_{N \in leaves} SA(N) \cdot t_N \right) \tag{4}$$

In this formula, the $SA(T)$ is the surface area of the bounding box of the scene (also of the whole hierarchy). The three formulas in this section are also described in the work of Bittner et al. [BHH13].

More recently, Aila et al. [AKL13] discussed the discrepancy between the costs calculated using the SAH and the actual rendering times. In their work, the authors also proposed two new metrics to be used along the SAH to achieve better results. However, we have implemented only the original version of the SAH in this thesis.

2.3 Method categorization

When categorizing the bounding volume hierarchies and the related algorithms, there are more points of view to be considered. The first is implied by the character of the rendered scene. The scene can be static, remaining unchanged over time, or various types of changes can happen to the scene primitives. These can be moved, scaled, destroyed (they can disappear) or new primitives can be added to the scene. Such scenes are called dynamic (also animated). When considering the algorithms for this type of scenes, we could find a finer categorization, e.g. by defining the construction (which would require even finer categorization) and the update algorithms categories. In this thesis we focus solely on the algorithms for static scenes. Of course, some algorithms can be used for both types of scenes.

Another point of view would be to divide the algorithms into the CPU and GPU categories, based on the type of the processing unit they are designed for. Some of the algorithms could be implemented for both types without significant design changes, but the others are suited for one of the types only. In general, the GPU algorithms require a high degree of exploitable parallelism to maximally utilize the many-core architecture. The parallelism is desirable also in the CPU algorithms, but its degree is usually lower. In our thesis we review the algorithms from both categories but in the implementation we focus only on the pure CPU methods or those easily usable for both types of processing units.

When categorizing the algorithms for static scenes, we could define the first-level categorization by defining the construction, optimization and related algorithm categories (consisting of ray traversal algorithms and other methods). In this thesis we focus mainly on the construction and optimization algorithms. We will omit the ray traversal algorithms and leave the related category for other methods. We extend the construction category further by defining a finer subdivision, creating the top-down,

bottom-up (agglomerative clustering), incremental, linear BVH and hybrid construction categories. These five are added by the optimization and related categories, leading to seven categories in total. We define the BVH method categorization using these. In the next sections, we will provide a more detailed description of the categories and the algorithms belonging into them.

2.3.1 Top-down construction

The top-down construction algorithms are very popular. The main principle is a subdivision of a BVH node containing primitives into two or more child nodes, each containing a subset of primitives (in this section we will assume the number of child nodes to be 2). This approach is very similar to constructing k-d trees, but in the BVHs the considered set of primitives is split into disjoint subsets (usually; there are exceptions which we will not consider now). K-d trees, on the other hand, place the primitives straddling a splitting plane into both respective child nodes, which leads to primitive reference duplication.

At the start of a top-down algorithm, the initial set of primitives contains all in the scene. This set corresponds to the root of the bounding volume hierarchy. Construction starts by evaluating the set (i.e. the possible ways to partition it) and termination criteria and eventually partitions the set and the root if convenient. This principle repeats on the possible new BVH nodes. The algorithm thus proceeds from the root down and constructs the branches of the hierarchy, implying the top-down name.

As termination criteria the algorithms usually use the following two thresholds. Either of these can indicate to terminate the construction of a BVH branch. The thresholds are:

- Primitive threshold, also referred to as a maximum number of primitives in a leaf. It terminates the construction when the cardinality of the primitive set is sufficiently small. It is considered more convenient not to apply the subdivision principle to a branch as far as possible (i.e. until all the leaves contain one primitive each), but to create leaves containing more primitives (values 2 or even 6 or 8 are used in various works).
- Cost threshold, based on comparing the expected costs of a possible inner node and a leaf. If it is more convenient to define a leaf (i.e. its expected cost is lower than for the inner node), the construction is terminated. The excepted costs for both types of nodes can be calculated using the surface area heuristic described in the section 2.2.2.

Eventually, other termination criteria can be added, e.g. a threshold of a node bounding box in the work of Wald [Wal07] (more specifically, Wald terminates the construction when a bounding box of the centroids of the primitives in the node is sufficiently small). The construction continues until all branches in the BVH are finished. A construction in a branch is finished either if indicated by the termination criteria or if only one primitive remains in the branch.

There are more ways to choose the candidates for a subdivision of a primitive set. In total, there are $2^n - 2$ possible ways to partition a set containing n primitives, which is impossible to evaluate. Therefore a partition candidate is usually obtained by defining a splitting plane, similar to the k-d trees. Each primitive is then considered to be placed to the left or right child node based on the coordinates of its significant point and the plane. As the significant point, the centroid of the bounding box of the primitive is usually used. Various splitting planes can be considered. The easiest top-down methods are spatial median and object median. In the spatial median, we directly

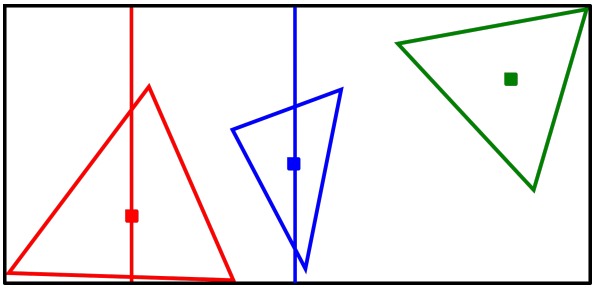


Figure 4 The example of meaningful splitting plane candidates for a node containing three triangles (in one axis). The first plane (red) implies placing only the red triangle into left child, while the second one (blue) implies both red and blue triangles placing there.

choose the splitting plane to divide the spatial region of a node into two halves. More particularly, we choose the coordinate axis with the largest spatial extent and calculate the center coordinate in this axis. Object median, on the other hand, partitions the set of primitives into two equal halves.

A more advanced approach evaluates the convenience of more splitting plane candidates. A cost of a candidate can be estimated using the surface area heuristic, which also allows to compare the candidates. We should be interested in the lowest possible number of candidates (to achieve high performance construction), which are truly meaningful to evaluate. Since the primitives are usually placed into child sets based on the centroids, it makes sense to define the splitting plane candidates as those where the numbers of primitives in the child sets change (by taking a primitive out of one of the sets and placing it into the other). In a single coordinate axis there are $O(n)$ meaningful planes for n primitives - the ones constructed through the primitive centroids. A usual approach is to sort the centroids in the respective axis, construct the planes and sweep from the left to the right to evaluate them. More particularly, the number of meaningful planes is $n - 1$, since we restrict any of child sets to be empty, splitting plane with index 1 places only the first object into the left child set, and the plane with index n implies empty right child set. We can see an example of such splitting plane candidates in the figure 4.

We can evaluate the splitting plane candidates in one or more coordinate axes. Assuming only one, we can flip the axes regularly in between the successor nodes or use a random-based selection, e.g. by using the axis with largest spatial extent with the probability of 0.7 and the regular flip strategy with the probability of 0.3. Alternatively, we can evaluate the candidates in all three axes. This can be implemented by iterating through the axes, sorting the centroids, evaluating candidates and choosing the best candidate through all the axes. This method is called Full Sweep (also Full Sweep SAH) and allows to construct high-quality hierarchies. It is therefore often used as a reference method. It main disadvantage is the higher number of evaluations (evaluating all three axes) and therefore higher construction times. The method is described in the work of Wald et al. [WBS07].

The top-down construction is the analogy to the quicksort algorithm, where the splitting plane is used as a pivot. The theoretical lower bound of the construction complexity also comes from the quicksort, more specifically from the complexity of sorting based on comparisons, which is $O(n \log n)$. To reach the lower bound, careful implementation is needed. The scheme achieving this complexity is described by Ganestam et al. [Gan+15] (the method sorts the primitives once in each of the coordinate axes and then maintains sorted sequences). The Full Sweep method as described above

(with sorting before evaluating each of the axes in each node), on the other hand, yield a complexity of $O(n \log^2 n)$ The complexity can also be reduced even more at the cost of approximating the construction, e.g. by not considering all the splitting planes and evaluating only their subset, as Wald suggested [Wal07].

The methods from the top-down category are:

- The spatial median algorithm, described in the work of Kay and Kajiya [KK86].
- The full sweep SAH construction method, described by Wald et al. [WBS07].
- The algorithm using the approximation of binning, proposed by Wald [Wal07].
- One of the proposed algorithms to construct a multi-BVH (i.e. hierarchy with higher branching factor than 2) by Wald et al. [WBB08].
- The Split BVH (also SBVH) structure, proposed by Stich et al. [SFD09].

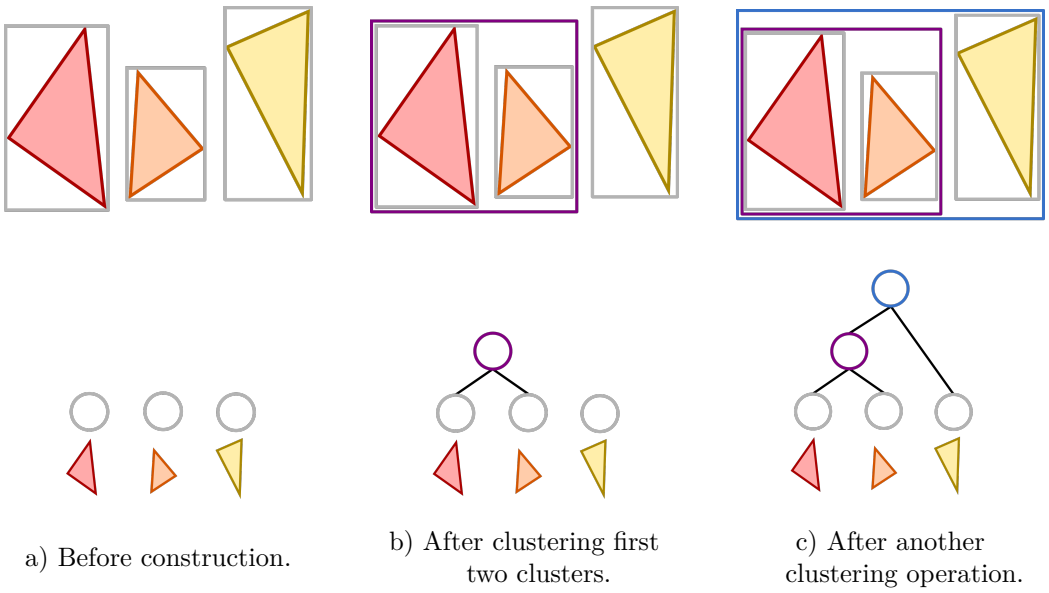
2.3.2 Bottom-up construction (agglomerative clustering)

The agglomerative clustering can be thought of as an opposite approach to the top-down construction. The clustering was adapted to bounding volume hierarchy construction from other scientific fields, but was long considered inefficient for the use here. The complexity of the naive solution is $O(n^3)$ and other algorithms proposed yielded $O(n^2)$ complexity, which was still rather high. First efficient algorithms were shown in the work of Walter et al. [Wal+08], which brought more interested into the field of agglomerative clustering construction. Though the construction times of the algorithms were still not sufficiently low, other works followed and improved them, making the times sufficient. The interest in the agglomerative clustering methods stands mainly because of the ability to produce hierarchies of high quality.

In the context of this construction scheme, by using the word clusters we mean hierarchy nodes (inner ones or leaves). The construction is based on merging clusters (which correspond to BVH subtrees) together and thus obtaining the larger ones (i.e. building the hierarchy from bottom to top). The construction starts by creating the initial set of clusters. These are of size 1 (i.e. it is a set of future hierarchy leaves, each containing exactly one primitive). The clustering phase is then repeatedly applied (clustering first the leaves and then also inner nodes into subtrees by creating more inner nodes), until only one cluster, corresponding to the root of the BVH, remains. The construction of the BVH using the bottom-up scheme can be seen in figure 5 on a simple example of scene containing 3 triangles.

Having the set of clusters and proceeding to merge some of them, we must be able to evaluate the convenience of merge of any two clusters. For this purpose the dissimilarity function (or also distance function) is used. It allows to describe the convenience by cost and thus to compare more possible merge candidates. There are more ways to define the function and it can be even user-defined. However, it must must be symmetric. A surface area of the bounding box containing both of the two considered clusters is often used. Another example would be the distance between the two centroids of the considered clusters. Having the distance function defined, the approach during clustering is to find the globally most-convenient pair of clusters and merge these into one. This principle repeats.

When searching for the most-convenient match for a single cluster, we can either search through the whole scene or limit the search to a neighborhood of the queried cluster. As the whole scene search is inefficient, this is crucial for the construction performance. The scene-wide search is also considered not necessary in order to achieve higher quality of the resulting hierarchy, because the search is more of a local operation, and the neighborhood of the queried cluster is usually sufficient to search in only. The



a) Before construction.

b) After clustering first two clusters.

c) After another clustering operation.

Figure 5 The example of the bottom-up construction on a simple set of objects. In the top we can see the primitives in the scene and the bounding volumes of the hierarchy that is being constructed. In the bottom we can see the topology of the hierarchy. In the left image we can see the scene before the construction, in the middle after clustering first two clusters. In the right image one more clustering operation is performed, resulting into the complete hierarchy. The color of the bounding volumes in the top part correspond to the respective inner nodes in the bottom part.

restriction of the search into the different types neighborhood is examined in various works and allows to propose algorithms with sufficiently low construction times, while still achieving high quality of the resulting hierarchy. By the restriction, the time complexity of the construction can be even lower than $O(n \log n)$, for example, Gu et al. report the complexity of $O(n)$ while carefully setting the parameters of the algorithm. The lower complexity than the theoretical lower bound implied by sorting using comparisons comes from the approximation.

The methods using the agglomerative clustering are described in the following works:

- The heap-based and locally-ordered clustering algorithms presented by Walter et al. [Wal+08].
- The Approximate Agglomerative Clustering algorithm, proposed by Gu et al. [Gu+13].
- The Parallel Locally-Ordered Clustering algorithm, proposed by Meister and Bittner [MB18].

2.3.3 Incremental construction

The category of algorithms, that construct the bounding volume hierarchy incrementally, is rather a small one. The main principle is to process the primitives in the scene sequentially and insert them one by one into a partially-built hierarchy (which is a valid hierarchy after each of the insertion operations). To be able to insert a primitive into the BVH, the algorithm must be able to determine the convenient place in the hierarchy to insert the primitive into. The construction begins by inserting the first primitive (i.e. defining the root as a leaf and containing the primitive). After that, for each processed primitive the globally most convenient place to insert it into is found and the primitive

is inserted there. The process is repeated until all the primitives are inserted, resulting into the complete hierarchy.

Goldsmith and Salmon [GS87] proposed one of the incremental algorithms, but it was later shown as inefficient by Havran [Hav00]. Seeming difficult to design, the incremental algorithms then did not receive much attention and other construction schemes were prefered. Recently, a more advanced algorithm was proposed by Bittner et al. [BHH15]. The authors proposed the strategy to find the globally most convenient place in the hierarchy to insert to and combined the insertion process with phases of hierarchy optimization to achieve better results. The authors also proposed two parallel versions of the algorithm, making it more suitable for the modern multi-core CPUs.

The algorithms of this category are:

- The algorithm proposed by Goldsmith and Salmon [GS87].
- The algorithm proposed by Bittner et al. [BHH15], which combines the insertion of primitives with periodical topology optimizations.

2.3.4 Linear BVHs

The Linear bounding volume hierarchy was proposed by Lauterbach et al. [Lau+09] and various works continued to improve it. The main principle of these algorithms is to reduce the hierarchy construction to sorting. More specifically, linear BVHs reduce it to a problem solveable by radix sort, which is convenient because of the low time complexity of radix sort ($O(d \cdot n)$, where n is the number of elements and d is the number of digits to describe an element).

In order to be able to use a variant of radix sort to sort primitives in the scene, we need to assign each primitive a number. This is done by calculating a Morton code for each primitive. The code is constructed based on a significant point of the primitive such as the centroid. The radix sort of the primitives then implies ordering of the primitives along the space-filling Morton curve, as can be seen on figure 6 (courtesy of Lauterbach et al. [Lau+09]). Though there exist also other space-filling curves, Lauterbach et al. proposed to use the Morton curve because the Morton codes can be computed easily using bit operations.

To calculate the Morton codes of primitives using k bits per dimension, the bounding box of the scene is divided into the lattice of $2^k \cdot 2^k \cdot 2^k$ cells, which is illustrated on figure 7 (courtesy of Vinkler et al. [VBH17]). Each cell can be indexed in all three coordinate axes and the binary representation of the indices is used in the subsequent Morton code calculation. To calculate the code for a single primitive, the centroid of the primitive is first projected into the lattice. It is projected into one of the cells and the corresponding binary indices are used to calculate the Morton code for the respective primitive. In other words, the purpose of the lattice is to quantize the coordinates of the centroids. To be more straight-forward, the projection is usually calculated using a predefined constants calculated from the scene bounding box and is done for each of centroid coordinates separately. Having calculated the three binary indices, each of these is shifted to be able to interleave with the others later. The shifting produces two zeros in between each of the former digits. The three modified indices are then interleaved using bit shifts and bit AND operations, resulting into the Morton code. The code containes of subsequent triples of indices-specific digits interleaved so that they regularly alternate, i.e. $x_1y_1z_1x_2y_2z_2x_3y_3z_3...$.

Having the Morton codes for all primitives calculated, we can then apply the radix sort to sort them along the Morton curve. When they are sorted, the hierarchy construction part itself begins. The algorithm works in top-down manner, and recursively

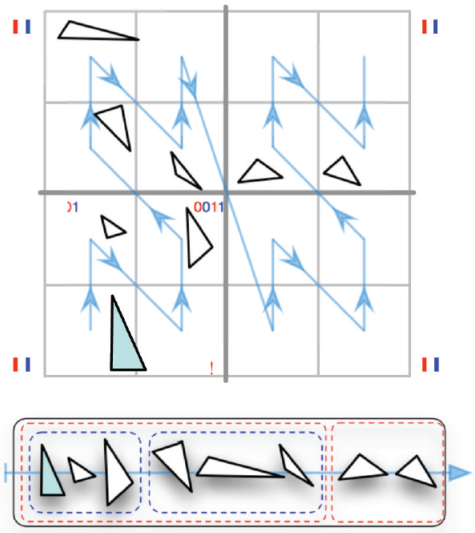


Figure 6 The ordering of the primitives in the scene along the Morton curve implied by their sorting according to the Morton codes. Courtesy of Lauterbach et al. [Lau+09].

subdividing the set of primitives similarly to the original top-down construction. The difference is that it always partitions the primitive set based on the Morton codes of primtives. More specifically, it examines the bits of the Morton codes to determine whether to place the primitives into left or right child sets. In the root level, it starts by examining the most-significant bit, placing the primitives with 0 bit values to the left child set and those with 1 bit values to the right child set. After subdividing the set, the algorithms continues by evaluating the child nodes and does this by examining the next (one after the most significant) bit in the Morton codes. The algorithm stops subdividing a hierarchy branch, when the bits of the Morton codes used in the branch are spent, or when we have only one primitive in the set.

The determination of the belonging of the primitives to child nodes can be done fast, since the primitives are sorted according to their Morton codes. Therefore the binary search algorithm can be used to find the partitioning primitive as the last the sequence having 0 value in the respective bit. All the primitives prior to the partitioning one in the sequence are placed into the left child set, and the others are placed into the right child set.

Since the complexity of constructing the hierarchy is connected to the convenient one of radix sort, the original linear BVH itself (as described above) achieve high performance construction. On the other hand, the quality of the hierarchy computed solely by using the such principle is lower compared to the other categories. This follows from the fact that the ordering of primitives along the Morton curve itself is not sufficiently convenient for use in ray tracing alone. Therefore the basic construction principle is combined with other, e.g. with the top-down construction using the surface area heuristic. Part of the hierarchy is then constructed based on the Morton codes while another part is constructed using SAH. The linear BVH principle can be implemented on both CPU and GPU, but is used mainly on a GPU, due to the various massively-parallel algorithms proposed.

In the context of this thesis, we refer to all the BVH algorithms exploiting the basic linear BVH principle proposed by Lauterbach et al. (and eventually combining them with other construction approaches) as linear BVHs (also abbreviated LBVHs) The

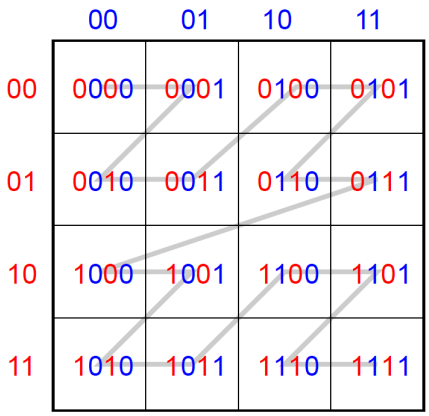


Figure 7 The dividion of the scene into a lattice implied by the use of the Morton codes. In this example, two bits per dimension are used in the code. Courtesy of Vinkler et al. [VBH17].

algorithms from this category are:

- The linear BVH algorithm and the hybrid LBVH-SAH algorithm proposed by Lauterbach et al. [Lau+09].
- The HLBVH structure extending the LBVH, proposed by Pantaleoni and Luebke [PL10].
- The optimized version of the HLBVH simplifying the construction using work queues [GPM11].

2.3.5 Hybrid construction

The hybrid category is defined for the algorithms either working differently from the previous construction approaches, or for those combining them. An example of such an algorithm is the Bonsai, proposed by Ganestam et al. [Gan+15]. The algorithm works in two phases. It first partitions the primitives into smaller groups using the fast spatial median top-down approach. Then it proceeds by creating a local hierarchy of each group (called mini tree), using arbitrary approach. A top-level hierarchy is then built using the local hierarchies as leaves, also by arbitrary approach. Any method from the categories above can be used for either of the phases of construction (Ganestam et al. use an optimized implementation of the full sweep SAH), therefore we consider it as a hybrid.

2.3.6 Optimization algorithms

The optimization algorithms use an already constructed hierarchy as an input. They perform topology optimizations of the tree structure of the hierarchy and aim to improve its quality beyond the former results of the construction algorithms. As non-construction ones, these algorithms stand beside the categories above, defining an independent field of study. The optimizations can be applied to an already constructed hierarchy when the application has enough time for it. As an example, when the construction algorithm for a dynamic scene is even faster than just matching the frame rate, the time saved can be used for optimizations to increase the hierarchy quality.

The optimizations can be terminated in various ways. One termination criterion comes from the example above - the time. The optimization then proceeds for a certain period of time. Alternatively, we can terminate the optimization when it cannot increase the quality of the hierarchy further. Such an approach can, however, lead to

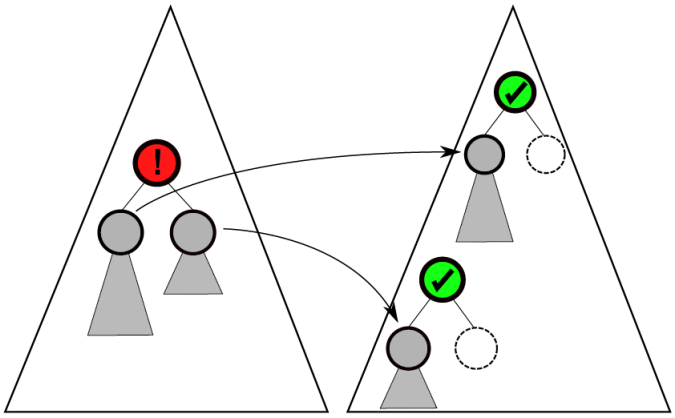


Figure 8 The illustration of optimization proposed by Bittner et al. An unconveniently placed node (red) is optimized by removing both its child nodes representing whole subtrees (grey) and reinserting them back at more appropriate places (represented by the green nodes). Courtesy of Bittner et al. [BHH13].

an optimization ending in the local maximum (of hierarchy quality) and not increasing the quality further. Some algorithms account for that fact, e.g. by using random sampling in the case of the Insertion-based optimization algorithm proposed by Bittner et al. [BHH13]. This algorithm is based on removing the unconveniently placed BVH subtrees and inserting them back into more appropriate placed in the hierarchy. It is illustrated in the figure 8 (courtesy of Bittner et al. [BHH13]).

The field of the optimization algorithms got the attention after the proposal of the algorithm by Kensler [Ken08], which applied tree rotations to improve a hierarchy. The algorithms of this category are:

- The algorithm based on tree rotations using the hill climbing and simulated annealing heuristics, proposed by Kensler [Ken08].
- The algorithm combining the tree rotations as proposed by Kensler and the refitting for the updates of the animated scenes, proposed by Kopta et al. [Kop+12].
- The algorithm based on removing the nodes of a BVH and reinserting them back to more appropriate locations, proposed by Bittner et al. [BHH13].
- The TRBVH algorithm based on restructuring treelets of nodes, proposed by Karras and Aila [KA13].
- The ATRBVH algorithm optimizing the TRBVH by employing the agglomerative clustering in the treelet restructuring, proposed by Domingues and Pedrini [DP15].

2.3.7 Related techniques

In this category we mention some other methods not described until now. These methods can be design details in the other methods and can be used in the construction or optimization categories mentioned above.

Though BVHs differ from the k-d trees in the construction, some of the published methods combine their construction principle, thus making a hybrid construction scheme. In particular, the construction approach from the k-d trees, referred to as to the spatial splitting, is adapted for the BVHs. The spatial splits, used in the top-down construction, can be more convenient at some points than the BVH partition approach. The spatial splits are adapted for the BVHs in the Split BVH (SBVH) structure proposed by Stich et al. [SFD09]. During the partitioning in the SBVH, a choice can be

made between a spatial split in the spirit of the k-d trees and the usual BVH partition candidate, selecting the more convenient one. The SBVH has been later parallelized on a CPU by Fuetterling et al. [Fue+16]. The spatial splits are also exploited in the work of Hendrich et al. [HMB17], who propose to incorporate them into the proposed construction algorithm refining an auxiliary BVH.

The BVHs can also have a branching factor more than 2. This is used e.g. in the work of Wald et al. [WBB08], in which the higher branching factor allows to test a traced ray against multiple BVH inner nodes. The authors proposed to use the SIMD instructions to achieve that. Such an approach is an alternative to the usage of the SIMD instruction in the tracing of packets of rays.

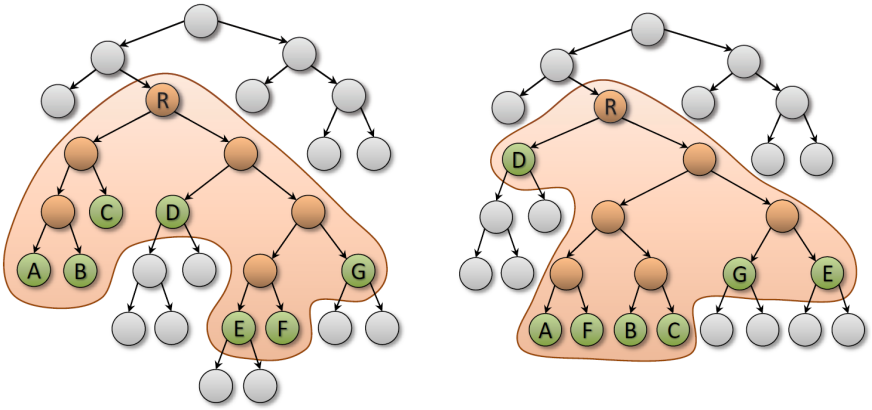
2.4 State of the art

In this section we present a review of methods we have not discussed further in the previous sections. We also omit the descriptions of the algorithms we chose to implement in our thesis. These are presented in the next section, 3.

The bounding volume hierarchies were discussed in one of the early works by Kay and Kajiya [KK86]. The authors first described the use of bounding volumes for ray tracing. The ray-volume intesrction test is cheaper than the object-volume one, which can thus be avoided in case on not intersecting the volume first. An efficient design of volume representation is presented along with the approach to construct bounding volumes for polyhedra, implicit surfaces and compound objects. The authors then extended the usage of bounding volumes into the hierarchies (BVHs), which allow for pruning the entire subtrees of volumes and objects. Two BVH construction approaches are presented. The first is based on clustering the objects as based on the order they are stored, aimed to be usable when the objects are stored in spatially coherent fashion. The second approach is the spatial median top-down method. The authors also describe the ray traversal algorithm.

Wald et al. [WBS07] presented methods based on the top-down construction scheme. Their motivation was to propose efficient methods for deformable scenes (i.e. the ones where only positions and sizes of primitives change). They were also motivated by not changing the BVH topology as can be done collision detection algorithms and proposed to only translate the bounding volumes. In their work, they described the full sweep top-down construction method by extending the surface area heuristic for use in the BVHs. They also compared this method to the spatial and object median methods. The second important contribution of the work is the ray-packet traversal method, which can be a key technique for higher rendering performance (mainly for the primary rays). Wald et al. proposed to implement this technique using the SIMD instructions on a CPU. Instead of traversing a single ray at a time, the packet of e.g. 16x16 rays can be traversed using techniques such as the early hit and the early miss tests. During the traversal, the rays in the packet are flagged as active or inactive. In the early hit test, the first active ray is tested for the intersection and the whole packet descends in the hierarchy if the test returns positive result. In the early miss test, which can quickly discard the tracing of the whole packet. These two techniques are combined in an effective scheme.

Wald et al. [WBB08] examined the use of SIMD instructions in an alternative technique to packet-based ray tracing. Though this technique is efficient for primary rays (or the coherent ones in general), the proposed methods aimed on the acceleration of tracing less coherent rays. The core of the work was the construction of the BVHs



a) Before restructuring. **b)** After restructuring.

Figure 9 An illustration of the treelet restructuring in the optimization algorithm proposed by Karras and Aila. The illustrated treelet is of size $n = 7$, having 7 treelet leaves. Courtesy of Karras and Aila [KA13].

with higher branching factor (more particularly 16) and testing a single ray against 16 BVH nodes or triangles in the same time using the SIMD instructions. The authors proposed two methods to construct these BVHs, one based on top-down splitting and the other on the collapsing of a binary BVH into the one with higher branching factor.

Walter et al. [Wal+08] examined another construction approach for the BVHs, the bottom-up scheme. This scheme was long considered inefficient, which changed after the work of Walter et al. The authors proposed two advanced bottom-up algorithms, the heap-based one and the agglomerative clustering. The second technique received various optimizations in the subsequent works. Walter et al. examined the exact version of the clustering (with no approaximations), in which the search for the globally most-convenient pair of clusters to merge was accelerated by an auxiliary k-d tree.

Pantaleoni and Luebke [PL10] followed the work of Lauterbach et al. [Lau+09] by extending the LBVH structure. They proposed the hierarchical LBVH structure (HLBVH), which divides the construction principle into two parts, each of which is performed according to idea of the LBVH. The scene is again processed according to the Morton codes. In the first part of the construction, this is done in coarser manner, while in the second part the division is done more finely. The authors also saw the disadvantage of the proposed method in the lower hierarchy quality which comes from the LBVH principle. They therefore combined the HLBVH with the full sweep SAH method by taking the cells of the upper coarser subdivision as input for the full sweep. The lower part of the SAH-optimized structure is still constructed using the LBVH principle.

The HLBVH structure was later optimized by Garanzha et al. [GPM11]. In their work they eliminated the somewhat more complicated computations and proposed a novel solution based on work queues, running on a GPU. Instead of the full sweep method originally used for constructing the top part of the BVH, the authors employed a parallel binning method in the spirit of the work of Wald [Wal07], but also run on a GPU.

The motivation of the work of Karras and Aila [KA13] was to propose a BVH method that would run similarly fast to the GPU algorithms based on the Morton codes while still reaching the hierarchy quality high-quality builders. The authors proposed a GPU-based massively parallel optimization algorithm (called TRBVH) based on restructuring

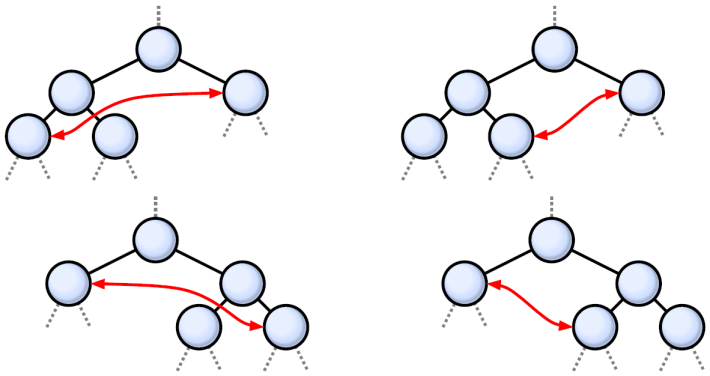


Figure 10 The possible rotations usable for the BVH optimization in the algorithm proposed by Kensler. Courtesy of Kensler [Ken08].

of treelets. A treelet of a BVH node is a set of its immediate descendants. A treelet also consists of inner nodes and leaves, which can in fact be BVH subtrees. The optimization is based on forming the treelets and then restructuring them into more convenient ones by finding the most convenient subtree for the treelet leaves, which is illustrated on figure 9 (courtesy of Karras and Aila [KA13]). This is believed to be an NP-hard problem and the authors therefore use a treelets of highly limited size and dynamic programming approach with optimizations to achieve faster solution. More specifically, they propose to construct a lower quality BVH first using the algorithm proposed by Karras [Kar12] and then perform the optimization phase on multiple treeles in parallel. Another contribution of the work of Karras and Aila is the improvement of the triangle splitting algorithm proposed by Ernst and Greiner [EG07] using various heuristics to estimate better the impacts of the splitting, which can be applied optionally before the construction of the initial BVH to improve the ray tracing performance.

The TRBVH structure was further improved in the work of Domingues and Pedrini [DP15]. They improved the treelet restructuring phase by not evaluating each new possible tree for the set of treelet leaves, but employing the bottom-up agglomerative clustering with these as input clusters. In general, this method can produce treelets of lower quality, but it is notably faster and larger treelets than in the work of Karras and Aila [KA13] can be considered. Because of that, Domingues and Pedrini report results similar to Karras and Aila while achieving them in in about 30% less time. This structure is abbreviated as the ATRBVH.

A method different from the others was presented by Kensler [Ken08]. The author investigated the optimization of an already constructed BVH in order to improve its quality. The main idea of the proposed method comes from the analogy of the binary search trees and BVHs and adapts the tree rotations for the BVH optimization. The situation is slightly different in the BVHs as the rotations boil down to swaps of a child of a node from the one side with a grandchild from the other side (the sides depend on the type of the rotation used, these are illustrated on figure 10, which is the courtesy of Kensler [Ken08]). The authors also propose two optimization schemes based on the rotations, using the hill climbing and the simulated annealing algorithms. The first method is greedy in nature and thus tends to end in the local minimum, which is accounted for by the second one. Kensler propose to combine these two methods, executing the simulated annealing followed by the hill climbing.

When using the BVHs for animating scenes, rather than rebuilding the entire structure per frame we can perform the refitting. However, due to the changes to the

primitives in the scene, the refitting itself can lead to periodical decrease of the BVH quality. The motivation of Kopta et al. [Kop+12] was to compensate the decrease and they proposed to achieve that by combining the refitting with the BVH tree rotations, previously employed as optimization procedure by Kensler [Ken08]. To account for the fact that triangles in the same BVH leaf can also get far from one another during the animation, the authors also proposed to perform leaf splitting which can be done trivially for leaves containing 2 triangles and using the binning top-down algorithm [Wal07] for those containing more of them.

3 Analysis and Design

In this section we present the description of the algorithms we chose to implement. We chose the methods based on the attributes they are known for (e.g. the speed in the case of the algorithm using the binning proposed by Wald [Wal07]), but we also wanted to compare the algorithms from different categories from our categorization presented in section 2.3. Our task was to implement 6 algorithms. We had added these by implementing the seventh work, the Extended Morton codes. For the 6 algorithms, we have chosen these:

- The top-down construction using the binning approximation proposed by Wald [Wal07]. This algorithm provides significantly low construction times on a CPU. On the other hand, it achieves those by sacrificing some of the resulting BVH quality.
- The Approximate Agglomerative Clustering algorithm proposed by Gu et al. [Gu+13]. As a bottom-up method, it is known to provide high-quality hierarchies while maintaining low construction times on a CPU.
- The Bonsai algorithm proposed by Ganestam et al. [Gan+15]. This algorithm is reported to construct high-quality hierarchies in low construction times. We placed this algorithm in the hybrid construction category.
- The Insertion-based optimization algorithm proposed by Bittner et al. [BHH13]. We have chosen this algorithm from the optimization category since it provides significant BVH improvements and is designed for a CPU. It is also reported to be faster than the previous CPU optimization algorithms.
- The incremental algorithm proposed by Bittner et al. [BHH15]. This is one of the few incremental algorithm. It is modern and reported to construct high quality hierarchies. It is designed for a CPU.
- The Parallel locally-ordered clustering algorithm proposed by Meister and Bittner [MB18]. It is a rather new bottom-up method. As such, we were interested in the comparison between it and the AAC. The PLOC algorithm is also reported to built hierarchies of high quality.

Another reason for choosing these algorithms was that they are parallelizable on a CPU (with the exception of the Insertion-based optimization method). In the end, we preferred the PLOC algorithm (a bottom-up method) to a method from the Linear BVH category, from which we did not choose any method. The reasons for that were that these algorithms generally do not provide as high-quality hierarchies and that they are often designed for a GPU rather than a CPU. The GPU algorithms are beyond the scope of our thesis.

3.1 BVH construction using the binning

Due to the high computational cost, the ray tracing was considered too expensive for real-time applications for a long time. As the computer hardware improved and started to not being a limitation, the situation started to change. The real-time applications such as games can be, however, very dynamic, demanding significant changes in the

acceleration structures. Full, high-quality rebuilds from scratch of the k-d trees and the BVHs were long considered to be too expensive and update methods were examined instead. Fast construction techniques based on approximations were also the subject of research and some of them were applied to the k-d trees at first. Since the BVHs have convenient attributes, Wald [Wal07] was motivated to adapt the fast builds to them as well. He also expected the negative impact of the approximations to be smaller than in the k-d trees. This is because the degree of approximation is not that high for the BVHs, since they work with the exact bounding boxes of triangle sets during the node partitioning (as opposed to dividing a volume corresponding to an inner node in a k-d tree) and perfect splits and split clipping, described by Havran [Hav00] and Havran and Bittner [HB02], need not to be considered. In his work, Wald adapts the binning approximation to the construction of bounding volume hierarchies [Wal07].

3.1.1 The use of the binning for BVHs

The binning can be understood as a simplification of the partitioning step performed by the full-sweep construction algorithm. During the search for the best partition, the full-sweep method works with a sequence of objects' centroids subsequently sorted in all coordinate axes. Though evaluating the SAH-based costs for every possibility and leading to high-quality hierarchies, this approach is time-consuming. When the binning is applied, we evaluate the costs of only certain (rather low) number of equidistantly spaced partitions. This leads to avoiding a significant amount of computations. The simplified pseudocode of the partitioning step can be seen in pseudocode 1.

Algorithm 1: divideNodeBinning(Node n)

```
1  if evalTermination(n) then
2  |  define n as leaf;
3  splitAxis = n.centroidBounds.largestAxis();  // the largest spatial extent
4  calculate leafCost;                                  // cost of not splitting
5  initializeBins(splitAxis);     // empties the bins and bins the primitives
6  [split, minCost] = findSplit();                       // sweeps over the bins
7  if minCost < leafCost then
8  |  sortIndices(split);           // in-place sorting of primitive indices
9  |  define n as inner node;
   |  // creates the centroid bounds and primitive sets
10 |  create n.left and n.right according to split;
11 |  divideNode(n.left);
12 |  divideNode(n.right);
13 else
14 |  define n as leaf;
```

When the partition of an BVH node is to be determined, the algorithm proposed by Wald [Wal07] starts by creating k bins of the same width. Because the centroids of objects' bounding boxes are used during the partitioning, Wald proposed to create bins by subdividing their bounding box (i.e. centroid bounds) into k equal regions (bins) instead of using the bounding box of the node. This can be seen on figure 11 on an example of a node containing three triangles.

He also proposed to evaluate the possible partitions in the axis with the largest spatial extent only as it leads to sufficient results. We use these two approaches. The k

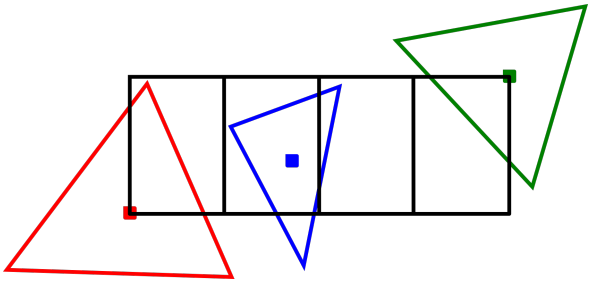


Figure 11 The bins constructed from the bounding box of the primitive centroids ($k = 4$). The primitives are placed to the bins according to their centroids.

bins imply $k-1$ splitting plane candidates, which will be evaluated in the same manner as in the full sweep method. Wald defined the cost of the candidate with neglecting the intersection and traversal constants and division by the surface area of the parent node of considered child nodes:

$$C_i = SA(left(n)) \cdot t_L + SA(right(n)) \cdot t_R \tag{5}$$

However, Wald did not define the corresponding cost of not dividing the node, yet he proposes to compare these two costs in order to determine whether the best chosen partition is to be performed at all. We therefore use the original SAH-based cost formulas for both inner node and leaf costs. To be able to calculate the costs, the bins store two auxiliary values:

- The number of primitives associated with the bin, n_i
- The bounding boxes of the primitives associated with the bin, bb_i (i.e. bin bounds)

The search for the best partition starts by first initializing the bins. The number of primitives n_i is set to 0 and the bin bounds bb_i are set to be an empty box (having all the minimum coordinates set to ∞ and the maximum coordinates to $-\infty$) for each bin i. Each primitive is then associated with the bin into which it's centroid spatially belongs. This can be done fast using the following formula for primitive o:

$$binID_o = \frac{k(1-\epsilon)(c_{o,l} - cb_{min,l})}{cb_{max,l} - cb_{min,l}} \tag{6}$$

where *epsilon* is a small constant (we use $1 * 10^{-5}$), $c_{o,l}$ is the coordinate of the centroid of the object o in the axis l, $cb_{min,l}$ is the coordinate of the minimum point of the centroid bouds in the axis l, and $cb_{max,l}$ of the minimum. The term $(1-\epsilon)$ shifts all the objects slightly to the left and therefore no extra handling of the objects on the right boundary of the centroid bounds is needed. After simple derivation and defining two constants $k_0 = \frac{k(1-\epsilon)}{cb_{max,l} - cb_{min,l}}$, $k_1 = cb_{min,l}$, the formula can be further simplified into a form allowing to bin the objects in a fast way:

$$binID_o = k_1(c_{o,l} - k_0) \tag{7}$$

When the primitive is associated to the bin i, the counter n_i is incremented and the bounding box of the primitive is included into the respective bin bounds bb_i. This procedure is done on every object of the considered set.

After all the objects are binned, we can evaluate the plane candidates to find the one that minimizes the cost. This is done by sweeping - two linear passes over the bins. The first pass is from the left to the right, and the numbers of objects to the left of the respective splitting plane as well as their bounding box are accumulated. The

Abbreviation	Parameter description
k	Number of bins used

Table 1 Parameters of the binning-based algorithm.

second pass is from the right to left, working analogically, but this time we can already evaluate the split candidates since we have all required values.

When the best partition is determined, the primitives are rearranged according to the it, implying two disjoint child sets. Wald suggests to perform this rearrangement in-place and we follow this idea. In the sequential version of the algorithm this can be implemented without additional copying. To determine to which child set the object belongs, the binning formula is applied again. Based on the resulting bin and index of the chosen split we determine the resulting child node.

Wald suggested to use two iterators to perform the sorting. One iterator starts from left and goes to the right, determining the child nodes for the primitives and stopping at the primitive belonging to the right one. Second iterator works analogically, but sweeps from the right to the left and stops at the primitive belonging to the left child. After both iterators find such primitives, these are swapped. The procedure continues until the two iterators meet. We have used a slightly different approach that leads to the same result. Our right iterator starts from the first index in the right half, which we determine from the number of objects accumulated on the left side of the chosen plane. This approach is slightly more friendly to caching.

Wald also discussed the number k of bins to use. As he recalls, the binning introduces smaller approximation when used for bounding volume hierarchies than it does for k-d trees. Therefore lower numbers of bins can be expected to achieve better results when used for the BVHs. During the experiments, he achieved sufficiently good results with 16 bins. Also, Gu et al. [Gu+13] used the same number of bins in their reimplementation of the method. We therefore follow these two works and also use 16 bins in each evaluation. Alternatively, an adaptive approach proposed by Wald can be used, setting the number of bins based on the number of primitives in the node:

$$k = 4 + 2 \lfloor \sqrt{n} \rfloor \tag{8}$$

where n is the number of objects in the considered node. However, the adaptive approach is stated to bring no appreciable improvement, except for the lower number of bins in the nodes near the BVH leaves and thus higher performance. On the other hand, one could expect lower performance in the higher levels of the hierarchy. We have not attempted this approach. We use 16 bins. Moreover, the parameters of the algorithm proposed by Wald can be seen in table 1.

To terminate the subdivision Wald proposes to use two standard termination criteria (primitive and cost thresholds) already mentioned in section 2.3.1. Wald suggests to use 2 or 4 as the primitive threshold and we follow the suggestion by using 2. He also proposed to terminate the construction if the centroids bounds become too small. He did not define this criterion further and we implemented it to hold if the extent of any of the dimensions of the centroid bounds falls below a constant, more specifically $1 \cdot 10^{-7}$.

The rest of the construction is not different from the full sweep method. We have implemented the method using iteration instead of recursion. We also use our defined stack for driving the construction which is located on the stack (in terms of computer memory) for higher performance.

3.1.2 Parallelization

Wald proposed three different parallelization techniques and combined them into two different parallelization schemes. These are the mixed horizontal/vertical sceheme and the grid-based binning. In our work, we have implemented the mixed horizontal/vertical scheme with slight improvements.

This scheme combines two parallel techniques, named vertical and horizontal. The horizontal technique is to construct the upper levels of the hierarchy, while the vertical constructs the lower. The horizontal technique is used when there are not enough tasks to run different threads independently on. It can be used until reaching certain depth or until the number of primitives in nodes becomes smaller than a defined threshold. The construction then switches to the vertical scheme. This scheme is somewhat simpler and means having distinct threads to construct disjoint BVH subtrees.

In the horizontal scheme, the threads co-operate on a single node subdivision. Briefly, the principle is the same as the sequential one described above, but the threads work on disjoint subsets of primitives, each treating in the same manner as in the sequential construction. The results of distinct threads are then merged and then a parallel version of the in-place sorting follows.

In the beginning of the subdivision of a node, each thread is assigned a subset of the primitives contained there. It then performs the binning of these primitives into its own bins. The binning results from all the threads are then merged. Then only one of the threads, e.g. thread 0, searches for the best partition candidate using the sweeping principle.

In our implementation (which is slightly different from the one described by Wald), after binning the primitives, each thread performs the two sweeps to accumulate the numbers of primitives on the respective sides of splitting plane candidates. These are later required during the parallel in-place sorting and we thus store the results into two arrays - one for the subsequent search for the best candidate performed by thread 0 and the other for the sorting. After the sweeps, we merge the results (accumulated numbers of objects per bins and the bin bounds). Wald uses the thread 0 to do this, but we perform the merge using parallel reduction algorithm, which is faster, especially when using higher thread counts. After the merge, we let the thread 0 to perform the two sweeps to determine the best partition candidate in the same manner as in the sequential construction.

After that, the parallel in-place sorting is performed. It starts by employing the parallel prefix scan to determine to which parts of the array the threads will write the primitive indices of their portion. For this, the mentioned second array of accumulated primitive counts is required. The threads are then again assigned a respective portion of primitive array and now have two values, $N^t_{L,i}$, $N^t_{R,i}$, which determine the beginning indices of the resulting portions of the primitive indices array to write to. These indices denote the start of the portions for left and right child node respectively.

By executing only like this, the threads would, however, write to the same parts of the array as they read the primitives from. Wald uses copying of the respective parts into an auxiliary array and then has the threads rewrite the primitives back at the correct indices, which is simpler to implement. We focused on the performance and therefore avoid the unnecessary copy step by using a primitive array of double length and maintaining pointers into the currently and previously used part of the array. This is, however, more difficult to code and debug.

Apart from using the parallel reduction for merging the binning results as described before, we also use it to calculate the centroid bounds and the scene bounding box, in

order to maximize the performance. After the primitive sorting, the thread 0 creates a new hierarchy inner node and recursively continues to process the child nodes. We add, that these parts of execution must be interleaved with thread barriers used to synchronize them. We used C++ 11 threads for the parallel construction and since the standard does not provide a barrier, we wrote our own version of it using two atomic variables and waiting implemented as while loop. We could alternatively use the Boost library version of barrier, but this way we managed to keep our code to use only the C++ standard library.

When a certain threshold of the number of primitives in the respective node set is reached (i.e. the number is lower than this threshold), the recursive principle stops in this hierarchy branch. We then pass the node as a task to the vertical algorithm. As mentioned, the vertical part executes threads to construct disjoint subtrees and therefore minimal synchronization is required. The process is very similar to sequential version, except each thread works with its own data structures. The threads are assigned with tasks dynamically. We drive the assignment using one atomic variable (an index to the task array), which is the only synchronization mechanism in this construction part. We also sort the tasks based on the numbers of objects in the subtrees in descending order, which allows to better utilize the threads, as Wald mentions.

Also, the BVH nodes used for the subtrees must be preallocated (otherwise we would need further synchronization when creating the nodes), which we can do by knowing the numbers of primitives in them. However, the construction can leave spaces in the node array between the respective subtree parts and we thus perform a final array compaction to get rid of them.

Wald also proposed a second parallelization scheme, called grid-based binning, which aims to avoid the performance bottleneck implied by the synchronization in the horizontal parallel part mentioned above. This scheme performs a higher number of splits a priori, resulting into a regular grid. The primitives are then binned into this grid and the subtrees for each of the grid cells are then constructed in parallel. We have not implemented this parallelization scheme.

3.1.3 Reported results

Wald evaluated the three versions of the proposed algorithm (the sequential one, the mixed horizontal/vertical and the grid based one) on a set of five scenes and compared them with the full sweep method and "BIH-style" (Bounding Interval Hierarchy). He presented the results of absolute build times, scalability and compared the results to the k-d trees. He also briefly discussed the quality of the hierarchies, but unfortunately did not present the costs based on the SAH or other metrics.

He reported the sequential version of the binning to be from 2 to 2.5 times slower in construction than the BIH but about an order of magnitude faster than the full sweep method. The sequential version allowed for 12 and 7 builds from scratch (Fairy Forest and Conference room scenes), while the grid-based parallel version running on 8 cores allowed for 38 and 48 builds. He also reported the algorithm to be from 4 to 13 times faster in construction when compared to the k-d trees. He also discussed the scalability of the algorithm, stating good results for 2 and appreciable for 4 threads, but not that good for 8 threads.

3.2 Approximate Agglomerative Clustering (AAC)

The motivation for examining the bottom-up (agglomerative clustering) bounding volume hierarchy construction algorithms was the quality that these methods are able to provide. Walter et al. [Wal+08] proposed two advanced agglomerative clustering algorithms (heap based and locally-ordered), but Gu et al. identified several downsides of the proposed methods. First, their parallelization was rather difficult. As the modern CPUs are multi-core, the parallelizable BVH construction is very desirable. One of the goals of Gu et al. was therefore to propose en easily parallelizable construction algorithm. Second, as Gu et al. [Gu+13] reimplemented and the algorithms proposed by Walter et al., they were unable to reproduce the results, even after non-trivial effort. Their motivation was therefore to propose a bottom-up algorithm competitive in construction speed with the top-down algorithms, such as the algorithm exploiting the binning proposed by Wald [Wal07].

3.2.1 The algorithm

As the expensive part of the locally-ordered algorithm Gu et al. [Gu+13] identified the nearest neighbor search. For speeding up the search Walter et al. used k-d tree as an auxiliary acceleration structure. The k-d tree was required to implement the search and also insertion operations, which made the implementation somewhat more difficult. Gu et al. therefore wanted to come up with a similar solution.

The authors based their algorithm on decreasing the nearest neighbor search cost, which is highest at the first stages of locally-ordered and heap-based clustering. They proposed to relax on the exact nearest neighbor search and to find only the approximate one by restricting the search. The approximation is based on scene presorting using the Morton codes and constrolling the clustering phase using these. By sorting the primitives and then subdividing the sorted sequence based on the bits in their Morton codes, the algorithm performs scene subdivision, namely using the spatial median splits. This subdivision therefore yields an implicit data structure, which Gu et al. name as the constraint tree. A leaf of the tree contains primitives belonging to the corresponding scene region (based on the centroids of their bounding boxes).

The key idea of the AAC algorithm is to begin the nearest neighbor search for each cluster using only the other primitives in the same leaf the cluster is located. After the clustering in the leaf, a new set of clusters emerges and the algorithm continues by traversing the constraint tree up. The new set of clusters is combined with the set emerging from the sibling subtree and another phase of clustering is performed on the combined set. This repeats until the root of the constraint tree is reached. The set of clusters corresponding to the root is then clustered until only a single cluster (the root of the resulting BVH) remains.

Following Gu et al., we begin by computing the Morton codes of the primitives. As proposed by the authors, we use $\lceil log_4 n \rceil$ bits per dimension (i.e. per single quantized coordinate), where n is the number of primitives in the scene. After that, we sort the primitives using the radix sort algorithm. Gu et al. used a parallel radix sort based based on the radix sort described by Shun et al. [Shu+12], but we have not attempted this method.

The algorithm then enters the main part, in which it first traverses down the implicit constraint tree. It starts with the initial set of primitives (containing all in the scene) and repeatedly partitions the set while traversing down. This is done based on the individual bits of Morton codes, starting from the most significant one. The primitives

Abbreviation	Parameter description
δ	Initial set size threshold
ϵ	Variable for reduction function

Setting	δ	ϵ
High Quality	20	0.1
High Performance	4	0.2

Table 2 Left: parameters of the AAC algorithm. Right: Settings of the algorithm [Gu+13].

having this bit set to 0 belong to the left set, the ones with 1 to the right. After taking a step down in the constraint tree, next bit of the Morton codes is used. This principle repeats. Since the Morton codes are sorted, to distinguish the child nodes means to find the divisor primitive which is the first one with the corresponding digit 1 in the Morton code. Similarly to Gu et al., we find it using the binary search.

The traversal of a single branch of the constraint tree ends, when the algorithm reaches a node containing a sufficiently small number of primitives (we understand it as a leaf). This number is implied by defining a threshold parameter δ. When the number of primitives is smaller than δ, the traversal stops. Next, the clustering part begins. The initial clusters are obtained trivially by creating BVH leaves, each containing one primitive. The nearest neighbor search is then performed only among the primitives in the leaf. δ should be defined small, Gu et al. use values of 4 or 20. Since these values are low, the authors propose to perform the approximate nearest neighbor search in brute-force manner with $O(|C|)$ complexity, where $|C|$ is the size of the set. As the distance function describing the cost of clustering two candidates Gu et al. propose to use the same function as in the work of Walter et al. [Wal+08], i.e. the surface area of the bounding box tightly enclosing these clusters.

When clustering, the algorithm reduces the size of the former cluster set. The authors propose to control the size using a reduction function. For an integer input, the size of the former set, the function returns the size of the new one. In the constraint tree leaves, δ is used as the input. The authors propose to use functions of form $f(x) = cx^{\alpha}$ (where $\alpha \in [0, 1]$ and c is a constant) as reduction functions. In particular, they use $c = \frac{\delta^{0.5+\epsilon}}{2}$. ϵ is another parameter of the algorithm and the authors set it as 0.1 or 0.2. We follow using this function.

The AAC algorithm performs post-order traversal of the constraint tree. After finishing the clustering in a leaf (being a left subtree of a node), it performs it in the sibling (right) subtree. Then steps up, unites the clusters resulting from both subtrees into a single one and performs the clustering using this set. This principle repeats until reaching the root of the constraint tree, after which the algorithm performs last stage of clustering there. Unlike Gu et al. we implemented our version of the algorithm using iteration instead of recursion, again with our defined stack data structure allocated on the stack memory.

The AAC algorithm has three parameters: the initial set size threshold δ, the term ϵ and the reduction function. We omit the reduction function parameter and describe the rest in table 2 (left). Apart from the reduction function mentioned above, Gu et al. propose two settings of the parameters δ and ϵ. The first setting is called High Quality (also abbreviated HQ) and is meant to build BVHs of higher qualities at the cost of longer construction. The second setting is called High Performance (abbreviated LQ or AAC-Fast in other works) which aims to optimize the construction time while decreasing the demands on the hierarchy quality. Both settings can be seen in table 2 (right).

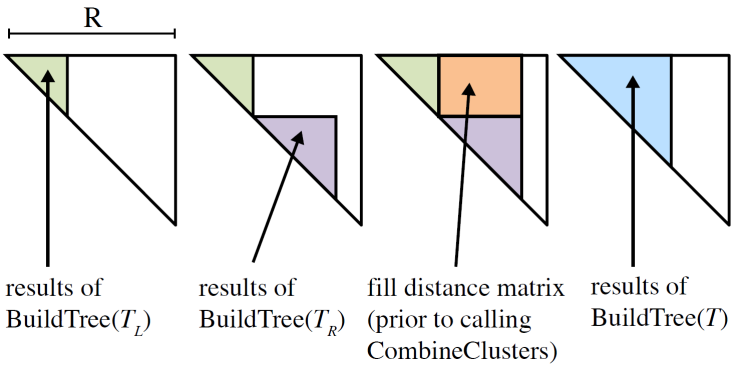


Figure 12 The results of the sibling subtrees in the constraint tree node are stored adjacent in the distance matrix (courtesy of Gu et al. [Gu+13]). In their work, the BuildTree(T) procedure constructs a BVH subtree from the set of primitives T using the AAC algorithm, T_L and T_R are the sets of primitives from the child nodes of node containing T in the constraint tree. CombineClusters procedure performs the clustering part.

3.2.2 Optimizations and parallelization

Gu et al. [Gu+13] also proposed several optimizations of the algorithm and we implemented some of those. To speed up the clustering and the nearest neighbor search parts, the authors proposed to cache the results from previous phases in a distance matrix. This matrix stores values of the distance function between any pair of active clusters. The matrix is symmetric and only one of its halves is used. We implemented the distance matrix optimization.

When two clusters are combined into new one, there is a need to compute the distances between in and the rest of the clusters in order to proceed with the clustering. Also, both of the former clusters are now invalid and have to be excluded from the computation. To reduce data movements in memory implied by these, the authors proposed to put the new cluster into the place of the first former cluster. Then, the last cluster in the set is moved to the place of the second former cluster. This way there is no more need to care for the invalid clusters and the size of the active part of the distance matrix decrements by one. After that, distances are calculated for the new cluster and for those that had one of the two former clusters as their nearest neighbors. We have also implemented this optimization.

After the clustering phase the part of the distance matrix used for the cluster set contains valid distances between each pair of clusters. It would be redundant to calculate them again in the next clustering phase corresponding to the parent node in the constraint tree. Gu et al. proposed to reuse the values. Because of the post-order nature of the AAC algorithm, this is simple. The clustering shrinks the part of the distance matrix used for a set of clusters corresponding to a node of the constraint tree. When putting a new set of clusters into the distance matrix, the values of the clustered left sibling subtree are already present and placed right before the respective set. After the clustering the set, the resulting part of the matrix is also adjacent to the left sibling part. When uniting the sets the only part left to calculate is the one containing the distances between pairs clusters, where each of which is of distinct former set. The appropriate moment to calculate this part is exactly during the union, which is what we do. These principles can be seen on figure 12 (courtesy of Gu et al. [Gu+13]).

Gu et al. also proposed to perform the hierarchy compaction (also called subtree flatting) according to the SAH in a way similar to that described by Bittner et al. [BHH13]. We also perform the compaction but in the spirit of Bittner et al., as a post

processing method. The compaction can be performed also during the construction, but leads to same results (because the construction of the tree depends solely on the distance function). When performed during the construction, it would be slightly faster, but the resulting code would be harder to read. In fact, Gu et al. also performed the compaction after the construction in their publicly available implementation.

The authors proposed to parallelize the AAC algorithm similarly to the techniques used in top-down algorithms, by having different threads process different disjoint subtrees of the BVH. In the view of the AAC algorithm, this means that different threads would process different branches of the constraint tree. Gu et al. note, that the optimizations using the distance matrix are not simply reusable for the parallel implementation of the AAC. In fact, the optimization can be used by having each thread maintain its own distance matrix. In the higher levels of the hierarchy, the results from distinct threads would need to be merged. Gu et al. did not proposed any parallelization usable for calculations withing the single cluster set as they identified the work to be sufficiently low. However, we have not implemented the parallel version of the algorithm.

3.2.3 Reported results

Gu et al. evaluated the proposed AAC algorithm on six test scenes, one of which was an individual object and five were architectural scenes. They evaluated single core implementation of AAC as well as parallel version run on 32 cores. They tested the two algorithm settings mentioned in table 2 (right), i.e. the High Quality and High Performance settings. The authors compared the proposed algorithm to three other algorithms, namely the full-sweep algorithm, the top-down SAH-based method using binning proposed by [Wal07] (in particular, the method tested by Gu et al. evaluated SAH costs using 16 bins along the longest axis while splitting) and locally-ordered clustering proposed by Walter et al. [Wal+08].

Gu et al. expressed the evaluation of the quality of hierarchies built by AAC algorithm using the costs of tracing rays through the scene. These values (for all algorithms) have been normalized to those of SAH-based top-down method using binning. Unfortunately, the authors did not report the costs of hierarchies calculated using surface area heuristic. This makes their results difficult to compare with our implementation. Therefore we use the results reported in the work of Meister and Bittner [MB18] for comparison. Meister and Bittner use the the traversal constant $c_T = 3.0$ and triangle intersection constant $c_I = 2.0$. Gu et al. report the AAC-HQ setting to produce hierarchies of similar quality to those produced by locally-ordered clustering algrotithm and tracing costs from 15% to 30% smaller than in the case of top-down binning method. They report lower tracing costs for 5 of 6 scenes for hierarchies built by the HQ setting than for those built by full-sweep top-down method (the only difference being the Happy Buddha scene). They also report comparable or even better results for AAC-Fast setting than for full-sweep BVH.

3.3 Bonsai algorithm

When constructing a bounding volume hierarchy to accelerate ray tracing, a speed of construction as well as the speed of subsequent rendering must be considered. The rendering speed is implied by the quality of the constructed BVH. The SAH-based full sweep algorithm constructs high-quality hierarchies and is therefore often used as a reference method. Ganestam et al. [Gan+15] therefore decided to base their proposed

algorithm called Bonsai on the full sweep method, described by Wald et al. [WBS07]. The Bonsai algorithm uses an optimized version of the full sweep algorithm applied in two stages. The proposed algorithm can be parallelized and vectorized.

3.3.1 Optimized full sweep algorithm

To find the best partition of a BVH node the full sweep algorithm goes through all the relevant possibilities in all three coordinate axes. The axes are evaluated one after another and a sort is usually exploited before evaluating each of them. This approach is rather costly and leads to a time complexity of $O(n \log^2 n)$, where n is a number of primitives in the scene. However, the lower bound of BVH construction complexity is $O(n \log n)$. This complexity can be achieved by a different, more careful implementation.

This is based on presorting the primitives, an approach described for the k-d tree construction by Wald and Havran [WH06] and reused here by Ganestam et al. [Gan+15]. Using this approach, the primitives are sorted only once in all three axes before the construction starts, and the sorted sequences are maintained during the whole construction in three separate arrays. When searching for the most convenient partition the sweep algorithm works with already sorted sequences and there is no more need to sort the primitives.

After choosing the partition in one of the axes, we know that the respective array is already sorted and there is no need to update it. We only need to update the other two arrays to keep them sorted. We implement this principle according to the work of Ganestam et al. When the partition is found, we flag the primitives based on which child node they belong to. Based on these flags, we reorder the primitives in the two respective arrays such that they keep the sorted order in the within child sets.

3.3.2 Bonsai algorithm

The Bonsai algorithm proposed by Ganestam et al. [Gan+15] can be exploit any construction method, but in the spirit of Ganestam et al., we use the optimized full sweep method described in the previous section. The Bonsai algorithm itself consists of five phases:

1. Primitive centroids calculation.
2. Mini tree selection. In this phase the sets of primitives are found very fast using the spatial median algorithm based on the centroids of the primitives. Each of the sets will be used to construct a mini tree.
3. Mini tree construction. For each of the sets of primitives a local BVH is constructed.
4. Mini tree (also Bonsai) pruning. This is an optional step aimed to improve the quality of the resulting hierarchy by pruning the unconvenient mini trees.
5. Top tree construction. A second-level BVH is constructed using the mini trees as leaves.

The BVH construction can be seen on figure 13 (courtesy of Ganestam et al. [Gan+15]). We will now explain the phases in more detail.

First, the centroids are computed as in other methods from the bounding boxes of the primitives. They are later used in mini tree selection and construction phases. The selection phase aims to quickly preprocess the scene by creating sets of spatially coherent primitives. The speed is achieved using an approach based spatial median method. In our implementation, we simply partition the initial set of all primitives

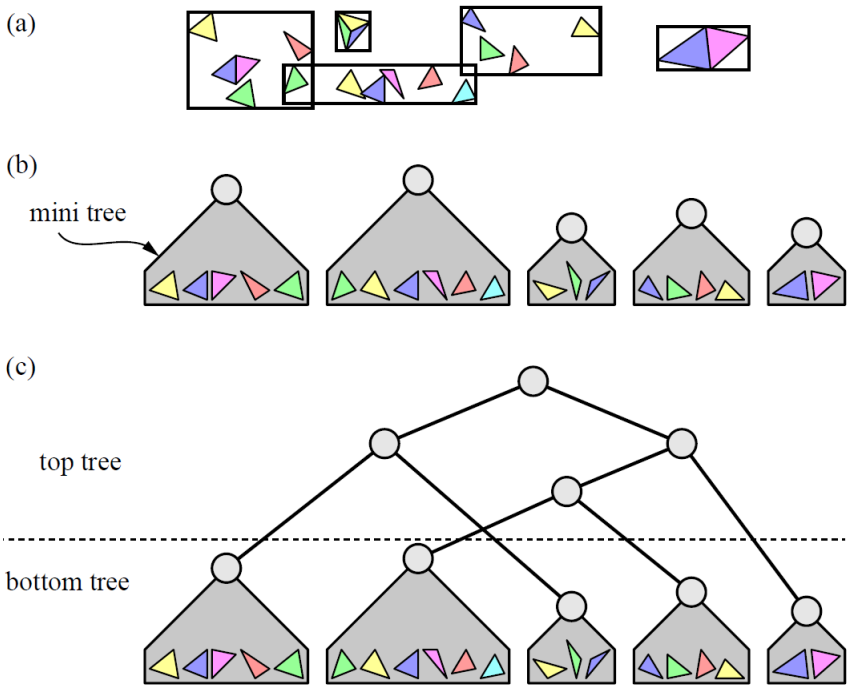


Figure 13 The construction phases of Bonsai algorithm [Gan+15]. (a) Mini tree selection. (b) Mini tree construction. (c) Top tree construction. Courtesy of Ganestam et al. [Gan+15]

while no BVH nodes are created. More specifically, we rearrange the primitive array in-place according to the method. This is applied until a threshold N of number of primitives is reached. Then the set of primitives is stored, which we do by storing begin and end indices into the rearranged primitive array. Ganestam et al. proposed to use 512 or 4096 as the possible thresholds, which we follow.

When the primitive sets are determined, the construction itself starts. It is performed in two stages, first of which is the mini tree construction and the second one is top tree construction. Though any construction algorithm can be used in both stages, we use the optimized full sweep method similarly to Ganestam et al., because it creates hierarchies of high quality. The method is applied on each of the primitive sets and constructs a small BVH, a mini tree. The output of our mini tree construction method is a set of mini tree roots. We will describe the next stage, mini tree pruning, in the next section and skip to the description of the top tree construction now.

The top tree construction phase uses the mini tree roots in the same manner in which the ordinary full sweep method uses the primitives, i.e. it constructs a BVH upon them. The mini tree roots are treated as leaves during the construction. To evaluate the SAH in this construction stage more correctly, we store the number of primitives in each mini tree with the respective root. These are used when searching for the best splitting plane. We also exploit the fact that we know the number of mini tree roots and we preallocate the top part of the BVH ($m - 1$ inner nodes for m roots), only setting the topology and variables in the node data structure during the construction. We do not use the preallocation in the mini tree construction, however.

3.3.3 Mini tree pruning

The Mini tree (or also Bonsai) pruning is a method designed to compensate the possible poor choice of the triangle sets used as an input to construct the mini trees. These are

chosen fast using the spatial median method, which does not account for the bounding boxes of primitives and the SAH.

The proposed pruning method traverses those mini trees that have the surface area larger than a certain threshold and splits each such mini tree in to two or more smaller mini trees. The threshold is based on the average surface area of the mini trees and a user-defined constant T. These two values are multiplied, which results in the desired threshold. After calculating the threshold, each mini is checked whether its surface area is larger than it and is eventually pruned. The authors, in fact, did not describe this part entirely clearly and two ideas come from their text. Either the threshold is obtained by the multiplication described above or T is used directly as the threshold. We use the first idea since it is more meaninful. Ganestam et al. suggest to use either $T = 0.1$ or $T = 0.01$ as the threshold.

On those mini trees having the surface area larger than the threshold, the pruning is applied. This algorithm traverses the mini tree and searches for the first node that is smaller than the previously used pruning threshold. After finding it, this node is defined as a root of a new mini tree. All the nodes on the path between it and the former root (including it) are deleted. This also means that the nodes highest in the hierarchy, not deleted and already traversed must become the new mini tree roots, too. The other nodes highest in the hierarchy (not deleted, but also not traversed) have also the potential to become new mini trees, but must be first checked using the pruning threshold.

We implemented the algorithm in the sweeping line fashion using iteration and three stacks allocated on stack memory. The first (traversal) stack, as usual, serves for traversing a mini tree. The second (roots) stack stores the possible mini tree roots from the already traversed part of the mini tree. More specifically, in a certain instant, it contains all the nodes that would eventually become new mini tree roots if the currently traversed node was found as the new mini tree as well. The third (passed) stack stores the path from the currently traversed node to the former mini tree root.

In the case a node is found as a new mini tree root, all the nodes from the roots stack are defined as roots of the new mini trees. Then all the nodes on the traversal stack are put into the list of nodes to be checked using the pruning threshold. Last, all the nodes on the passed stack are deleted. The deleting, however, leaves empty spaces in the BVH node array. Because of that, after the pruning is finished, we perform compaction of the array.

3.3.4 Parallelization and reported results

The authors implemented the proposed algorithm using multiple threads and vector instructions. The parallelization using threads is based on creating new threads during the subdivision in the optimized sweep method. In particular, when the node is subdivided, the former thread (that processed the node) continues by processing the right child, while a new thread is spawned to process the left child. This approach, however, leads to spawning a high number of threads. The authors added also other suggestions for the parallelization and vectorization, but we have not attempted these.

For the evaluation of the results, Ganestam et al. used a path tracer, in which the hierarchies were built on a multi-core CPU while the rendering was executed on both CPU and GPU (the traversal and intersections were executed on a GPU). The authors used several BVH construction algorithms for comparison. They used the optimized full sweep construction algorithm described above, the top-down construction that uses binning (proposed by Wald [Wal07]), for which they used Intel's Embree 2.2 implemen-

Abbreviation	Parameter description
N	Primitive set size threshold
T	Pruning threshold
prune	True if the pruning is used

Setting	N	T	prune
Bonsai	512	-	false
BonsaiP	512	0.1	true
BonsaiP*	4096	0.01	true

Table 3 Left: parameters of the Bonsai algorithm. Right: settings of the algorithm [Gan+15]

tation. They also used the Approximate Agglomerative Clustering algorithm proposed by Gu et al. [Gu+13], for which they used the publicly available implementation. This implementation is using a single thread and Ganestam et al. report the times generated by dividing the times of this single-core implementation by the number of CPU cores, which in their case was 4. They tested the two AAC settings proposed by Gu et al., the HQ and LQ settings. We discuss the AAC as well as the settings in section 3.2. The Bonsai algorithm was tested in three variants, the plain Bonsai without pruning and two pruning settings. The parameters of the algorithm are summed in table 3 (left). The settings of parameters for all the Bonsai variants are listed in table 3 (right).

The authors tested the algorithms on 14 scenes of different complexity, 9 of which were architectural scenes and 5 were individual models. The authors reported the absolute construction times and the times of tracing rays relative to the optimized sweep method. They also reported the absolute SAH-based costs of the hierarchies, and costs, build times and ray tracing performance relative to the optimized sweep method. For computing the costs of hierarchies, the authors used the traversal constant $c_T = 2.0$ and the intersection constant $c_I = 1.0$.

When compared the Bonsai algorithm variants with the optimized sweep algorithm, Ganestam et al. ray tracing performance from 75% to 95% with the variant without pruning, from 92% to 105% with an average of 98.5% for BonsaiP, and 97% to 108% with an average of 101.5% for BonsaiP*. They also report the increase in construction time because of the use of pruning on the range from very little to nearly double. The authors discuss that the cause is not the pruning itself, but the increased number of mini tree roots as an input for the top tree construction, which causes the time increase. As they also note, the build times are dependent on the primitive set threshold. They report the value 512 as reasonable based on the evaluation on the test scenes.

3.4 Insertion-Based Optimization

As the computational power of computer hardware increases, the ray tracing is becoming an alternative to rasterization. The efficient acceleration structures are mandatory in order to achieve sufficiently lower rendering times when using ray tracing. Therefore there has been (and still is) an extensive research in the context of bounding volume hierarchy construction algorithms. There are also other techniques to improve the hierarchy quality and rendering times, such as the optimization algorithms. These can improve the hierarchy quality to be even higher than when using the construction algorithm only. Any improvement in the hierarchy quality can be important to achieve faster rendering times when using ray tracing. This was the motivation of Bittner et al. [BHH13].

3.4.1 The algorithm

As an optimization technique, the algorithm proposed by Bittner et al. [BHH13] takes a built bounding volume hierarchy as an input, constructed by an arbitrary method. The

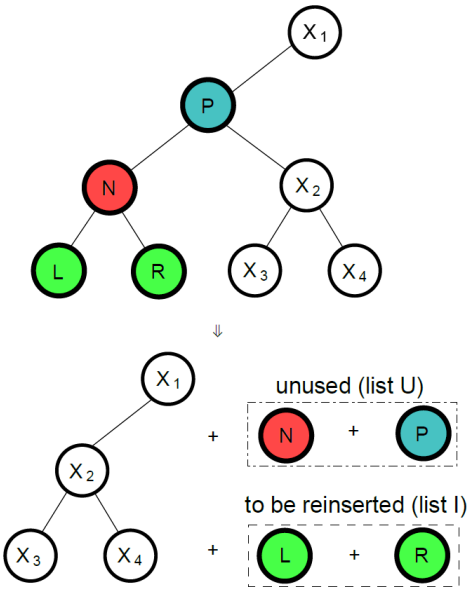


Figure 14 Illustration of the removal operation in node N. Node N, it's parent P and both children L and R are removed from the hierarchy. Courtesy of Bittner et al. [BHH13].

main principle of the algorithm is to improve the quality of the hierarchy by reinsertion of nodes. To describe it briefly, the algorithm consists of two main phases:

- The selection phase, when inner nodes convenient for optimization are chosen.
- The nodes update phase, which consists of removing the children of selected nodes from the hierarchy, finding a more appropriate position for them and inserting them there.

These two phases repeat, until the optimization is to be terminated. When examining the formula for calculating the cost of a bounding volume hierarchy 4, the main idea of the algorithm is to decrease the sum of the surface area of the inner nodes.

3.4.2 Node reinsertion

Since it is slightly more straight-forward, we will first describe the node reinsertion algorithm, similarly to Bittner et al. We will describe how to select the nodes for the optimization in the next section. Having the node to be optimized, the method starts by removing both its child nodes from the hierarchy. These are then processed sequentially. For both of them the globally most convenient position in the hierarchy is found and then the nodes are inserted there.

Assume we want to optimize the node N. As mentioned, we will remove both its children, L and R from the hierarchy. Moreover, we will remove the node N itself and its parent P. While the node L and R will be reinserted at more convenient positions, N and P will be used to link them with the rest of the hierarchy. After the removal of these four nodes, the hierarchy must be repaired on the affected place, i.e. the links in the hierarchy must be updated to be valid. This is illustrated on the figure 14, which is the courtesy of Bittner et al [BHH13]. As we can see, we must update the appropriate child index in the grandparent node of N to the index of the sibling of N.

Our inner node data structure (which is used by all our implemented BVHs) stores only the indices to both child nodes, which allows to keep its memory footprint low. On the other hand, in order to remove the four nodes, we must be able get them and in the beginning of the operation we are given only the index of N. In order to get

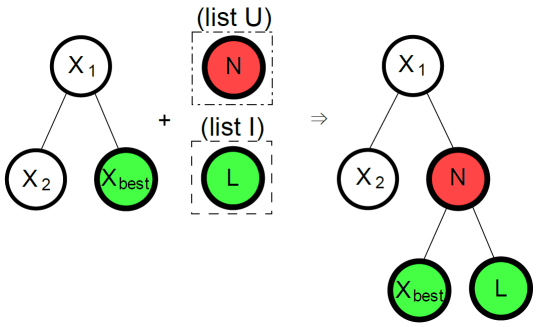


Figure 15 Illustration of reinsertion of node L to the most-convenient place in the hierarchy, represented by node X_{best}. The node N is used to link L to the rest of the hierarchy. Courtesy of Bittner et al. [BHH13].

the index of P we could traverse the tree from the root until finding it. However, this would result in low performance.

A simple way is to store the indices to the parent nodes. This is required for all the nodes. One way would be to modify the node data structure and add the index to the parent node to it. In C++, this would results to the increase of the size of the node data structures by 4 bytes and result to sizes, which would not be a power of 2 (which is the current size). However, this is convenient for the cache memory. Moreover, other implemented construction algorithms (with the exception of the incremental one) do not require the indices to the parents. We have thus chosen to store the indices in a separate array, which is allocated before the optimization and is deallocated upon its termination.

Being able to reach all four previously described nodes, we can remove them. We also update the array of up-indices accordingly. We also take care for the extra cases. If we were to remove a child N of the BVH root, we define the sibling of N as the new root. Next, the bounding boxes of all inner nodes on the path from grandparent of N to the root must be updated, since we have just removed a subtree from the BVH. Having the up-indices, this is rather simple, we simply traverse up, and for each traversed node we obtain the bounding boxes of both its children and create their union to obtain the new box for the node.

The reinsertion itself is similar to the removal and we thus describe it now. Assume we found the globally most-convenient place in the hierarchy to insert a node L into. This place is represented by another node X_{best} (can be inner or leaf). The algorithm starts the insertion by first inserting one of the linking nodes, N into the place of the X_{best}. Both L and X_{best} are then inserted into the BVH as children of N. We follow the work of Bittner et al. and define X_{best} as the left child and L as the right one.

In the subsequent ray traversal algorithm we can traverse the nearer BVH subtrees first and therefore this can have an impact on it. We compensate for that fact after the optimization by iterating through the nodes and setting the order of the child nodes. For a node, we do this by subtracting the centroids of the bounding boxes of its child nodes, choosing the largest-value coordinate and then comparing the centroids in the same axis. The one with the lower coordinate is set as the left child, the other as right.

The process of reinsertion is illustrated on the figure 15 (courtesy of Bittner et al. [BHH13]). Similarly to the removal of nodes, the links of the hierarchy must be updated (including the up-indices) and the bounding boxes of all inner nodes on the path from the linking node N to the root must be refitted, using the bottom-up traversal again.

The remaining part of the node reinsertion procedure is the search for the globally

most-convenient place in the hierarchy for a query node L. The return value of the search method is a hierarchy node X_{best} corresponding to the most-convenient place. For the search Bittner et al. proposed to use a branch and bound algorithm driven by a cost describing the convenience of insertion.

For this purpose the authors proposed a cost metric. The cost has two components defining it in combination, the direct cost $C_D(L, X)$ and the induced cost $C_I(L, X)$. Both of these components together represent the impacts (increases of the bounding boxes of the affected inner nodes) of the merge of the node L to be inserted with the examined (possibly the best convenient) node X. The direct cost is simply the surface area of the bounding box tightly enclosing both L and X nodes. The induced cost is the accumulated increase of the surface in all nodes on the path between the parent of X to the hierarchy root.

The branch and bound algorithm is implemented using priority queue. The queue item is a pair of values: node X from the hierarchy and the induced cost $C_I(L, X)$. The priority of the item is inversely proportional to the induced cost $C_I(L, X)$, i.e. the nodes with the lowest induced cost are taken first from the priority queue. The algorithm thus traverses the BVH branches with the lowest induces cost first. The algorithm is designed so that the branches having higher induced cost than the best solution X_{best} found so far will not be examined (they are pruned). Moreover, the whole search can be terminated when the branch on the top of the priority queue cannot bring any improvement against X_{best}.

For an item popped from the priority queue, the lower bound of induced cost in the respective branch is calculated. When the lower bound is higher than the best solution with the induced cost C_{best} corresponding to X_{best} found so far, the branch is not searched in. Otherwise the true induced costs for both children of the tested node X are evaluated. If any of these is lower than C_{best}, the child nodes are put into the priority queue. This implies that the algorithm searches in the most convenient branches first, ignores all the unconvenient branches as soon as possible and even ends the whole search is better solution than the one found so far cannot be found.

When discussing the algorithm with my supervisor and one of the authors, prof. Vlastimil Havran, Ph.D., we mentioned, that the branch and bound algorithm often generates nodes with induced cost equal to 0 in the levels near the root of the hierarchy. Pushing all such nodes into the ordinary priority queue would not be as efficient, so we implemented our own priority queue consisting of two internal data structures: an ordinary priority queue, into which the items with non-zero priority are pushed, and a linear array, which is used to store the ones with zero priorities. When popping this compound priority queue, it returns a node from the linear array, if there is any, or pops the inner priority queue eventually. This allows for a more efficient storing and obtaining of zero values without using the ordinary priority queue.

The branch and bound algorithm proposed by the authors is also used to drive the insertion of primitives in the incremental BVH construction, also proposed by the authors [BHH15].

3.4.3 Selecting nodes for optimization, optimization termination

The first phase of the optimization procedure is to select the nodes to be reinserted. This could be done using random node selection, but the authors propose to accelerate it based on various metrics. The purpose of these is to pick the nodes that cause the largest overhead in the BVH surface area and to optimize these first. The authors proposed three node inefficiency measures and eventually combined them into the fourth. The

three metrics account for various situations that cause the surface area overhead in the hierarchy.

The first measure, denoted $M_{SUM}(N)$, describes the relative increase of the surface area of node N with respect to the average surface area of it's children. The values of the measure are therefore high, when the children having a lot of space between them should be contained in N. This could happen for e.g. children with small bounding boxes which are, however, very distant. The second measure, $M_{MIN}(N)$, is similar, but the surface area of N is related to the surface area of that child of N, which has the surface area lowest. This measure is aimed to detect the situations when the children have significantly different surface area. The third measure, $M_{AREA}(N)$, is simply equal to the surface area of the node N. This implies selection of the nodes having largest surface area first. The three measures have the following definitions:

$$M_{SUM}(N) = \frac{SA(N)}{\frac{1}{|children(N)|} \cdot \sum_{X \in children(N)} SA(X)} \tag{9}$$

$$M_{MIN}(N) = \frac{SA(N)}{\min_{X \in children(N)} SA(X)} \tag{10}$$

$$M_{AREA}(N) = SA(N) \tag{11}$$

The authors then combined these three inefficiency measures into one by multiplying them, resulting in the combined inefficiency measure, $M_{COMB}(N)$:

$$M_{COMB}(N) = M_{SUM}(N) \cdot M_{MIN}(N) \cdot M_{AREA}(N) \tag{12}$$

This measure thus aims to detect all of the situations described above and Bittner et al. [BHH13] also report this combined measure to have the largest impact on the optimization, leading to the fastest optimization. We have implemented all four of these measures and decided to use the combined one, following the work of Bittner et al.

The optimization algorithm works is designed to work in iterations. In each iteration a fraction of nodes is optimized. The authors propose to optimize 1% of nodes in each iteration, which we follow. This correspond to k nodes, which should be optimized. We perform the selection using a linear pass over the array of nodes, allowing only the inner nodes to be processed, which is slightly faster than using a depth-first traversal, where we would need to maintain and use a stack or recursion. As described by the authors, we calculate the inefficiency for all inner nodes and then select the k most inefficient ones, which will be then optimized in the decreasing order based on their inefficiency (i.e. the most inefficient nodes will be optimized first).

First we implemented our own fixed-size priority queue for this task. Later we optimized this part by storing the inefficiency values with the respective nodes into an array. The values are then processed using the n-th element algorithm the C++ standard library, quickly reorganizing the array in $O(n)$ (where n is the number of inner nodes), followed by the library sort executing in $O(k \log k)$. Alternatively, the partial sort algorithm from the library can be used.

The algorithm achieves the largest improvements in the first iterations and then the cost of the hierarchy converges, oscillating around a certain value. During the optimization, the cost can be also increased for a certain period of time and then decreased again. The authors therefore proposed to base the termination criterion based on these facts. Instead of terminating the optimization immediately after any cost increase, the algorithm always performs a few iterations before deciding whether to terminate. This

Abbreviation	Parameter description
iboFrac	Fraction of nodes to be optimized
pT	# iterations for the inefficiency sampling
pR	# iterations for the random sampling

Table 4 Parameters of the Insertion-based optimization algorithm.

also speeds up the optimziation process, because the cost is not computed after each iteration, but after the number of these. This number is a parameter of the algorithm, denoted as p_T, and the authors propose to use $p_T = 10$, which we follow. After the algorithm would decide to terminate, the authors also propose to perform a last stage of iterations using the random sampling to select the nodes to be optimized. The termination in this stage happens in the same manner as in the previous one. The number of iterations in this stage, p_R is another parameter of the algorithm and the authors propose to set $p_R = 5$. Alternatively, the optimization could be terminated after a certain period of time, which is, however, not our case.

In the table 4 we sum the parameters of the algorithm.

3.4.4 The compaction algorithm

Bittner et al. [BHH13] also described a hierarchy compaction algorithm, which aims to decrease the hierarchy cost furthermore. When constructing a hierarchy, the branches can be built until single primitive remains in all leaves. It is, however, more convenient to construct hierarchies with more than one primitive per leaf. A hierarchy with only one primitive per leaf can be thus compacted using the algorithm proposed by Bittner et al. based on the surface area heuristic. In their work, the authors first let the construction algorithm to construct the branches as far as possible, then use the proposed optimization method and then the compaction algorithm.

The compaction is a post-order traversal algorithm, and as such processes both child nodes before processing their parent. It thus first traverses to the leaves, counts the numbers of primitives stored there, and then returns upwards and tests the convenience of compacting the subtrees. In each such examined node, the cost of the actual fully built subtree is compared to the cost of the possible leaf in this position (which would contain all the primitives contained in the subtree). If the cost of the leaf is lower, the subtree is compacted. The compaction does happend on the way up and can also happen more times in a branch.

We have implemented the compaction algorithm as iterative one using three stacks. The first is used for the hierarchy traversal. The second stores the results of the subtree left to the one, which is currently traversed. The third stack stores the nodes on the way to the root. These are later used for traversing up. We store these whenever going to any right child (because we will eventually traverse back up soon). The hierarchy compaction is sometimes also referred to as to the subtree flatting and is used also in the Approximate Agglomerative Clustering (AAC) and Parallel Locally-Ordered Clustering (PLOC) algorithms, which we have also implemented in this thesis. We therefore implemented the algorithm in the common predecessor of these classes. In fact, we can thus compaction on any of our BVHs. The algorithm also requires the raversal and intersection constants, c_T, c_I, for which Bittner et al. use the values $c_T = 3.0$ and $c_I = 2.0$.

3.4.5 Reported results

The authors evaluated the proposed optimization algorithm on a set of 14 test scenes (5 individual objects and 9 architectural scenes). The distinguished between these two categories when reporting the results. They reported the results of optimizing the BVHs initially built with the full sweep and the spatial median method. They discussed the reduction of the cost of the hierarchies, which for the individual objects and full sweep method almost none (with the exception of the 5% decrease in cost on the Hairball scene), since the method constructs high-quality hierarchies. The BVHs built using the spatial median are reported to be optimized to the level of the those constructed using the full sweep method. For the spatial median, the time needed for the optimization is reported to be up to 25 times faster than when using the tree rotations driven by the simulated annealing proposed by Kensler [Ken08]. On the architectural scenes, the proposed algorithm reduces the cost from 4% to 24% for the hierarchies built using the full sweep. The improvement in the cost is also reported to be larger than when using the method proposed by Kensler.

3.5 Incremental algorithm

The vast majority of bounding volume hierarchy construction methods requires the whole scene to be known before the construction. However, there are applications where this is not possible. These could use an incremental construction algorithm, which would not require the whole scene to be known in advance. The incremental algorithms were, however, considered to create BVHs of lower quality, providing higher rendering times. Because of that, the incremental construction was not much used. The motivation of work of Bittner et al. [BHH15] was to tackle these topics and propose an efficient incremental BVH construction algorithm leading to high-quality hierarchies. In their work, Bittner et al. also presented two parallelization approaches for the proposed algorithm and a example application of ray tracing a scene streamed over the network.

3.5.1 Insertion-based construction with global hierarchy updates

The algorithm proposed by Bittner et al. [BHH15] follows the previous work of the authors [BHH13], which proposed an insertion-based optimization algorithm for improving the quality of already built BVHs beyond the results of the construction algorithms. The proposed incremental algorithm processes the primitives by creating a leaf for each of them first and then inserts them into the partially-build BVH.

In our implementation, we preallocate both the leaves and the inner nodes of the hierarchy before we start construction. We are able to do this, because we know the number of primitives in the scene and the desired number of these in leaf (one) in advance. Though this can be in a slight contradiction with the motivation of the authors, our application is different and we chose to optimize our implementation this way. Alternatively, the node allocation could be done in a lazy fashion.

After the allocation, each leaf containing a primitive is inserted into the hierarchy sequentially, one by one. The selection of the place in the hierarchy, where the leaf is to be inserted, is crucial for the resulting hierarchy quality. The authors proposed to exploit the cost-driven branch and bound algorithm formerly designed for the hierarchy optimization (proposed in their previous work [BHH13]) for the search. Because we intended to implement this optimization algorithm as well, we designed the search algorithm such that both method can exploit it. The description of the algorithm can

Abbreviation	Parameter description
batch	Size of the primitive insertion batch
oFrac	Fraction of nodes to be optimized
coh	Coherence threshold

Table 5 Parameters of the incremental algorithm.

be found in section 3.4.2. For linking the inserted leaves with the rest of the hierarchy, we use the preallocated inner nodes.

As Bittner et al. noted, though the search algorithm would be able to find the globally most convenient place to insert each of the leaves into, this position would reflect only the actual state of the hierarchy. Therefore it does not account for the primitives inserted after each considered one. The authors proposed to resolve this fact by incorporating series of global optimizations in the spirit of the Insertion-based optimization method. More specifically, the insertion algorithm alternates insertion of primitives with global updates: after inserting a batch of primitives, the optimization phase is executed (a single iteration in spirit of description provided in section 3.4). We use the combined inefficiency measure for the optimization phase.

The optimization phase uses a cache of nodes for selecting the nodes to be optimized (node update cache). While inserting the primitives into the hierarchy, only some branches are often modified and others do not. The authors propose to optimize only the inner nodes from the affected branches. The nodes from these are stored in the node update cache. We have implemented the cache as a linear array of flags of the size equal to the number of inner nodes ($n - 1$ in the case of n primitives). We then consider only the nodes having the appropriate flags for the optimization.

Bittner et al. have also proposed two optimization for their new incremental algorithm. One of the optimizations is a post-processing optimization phase exploiting the Insertion-based optimization method. Any of our BVHs can exploit this method and we evaluate the use on the incremental algorithm in our results. The second optimization proposed by the authors is to cluster the subsequent two primitives if convenient and eventually insert them into the hierarchy together. This optimization accounts for the fact that the primitives can be provided in spatially coherent way. For a pair of subsequent primitives, a coherence test is applied and eventually these are clustered using a single inner node. Such a small subtree is then inserted into the BVH (i.e. the inserted node is the inner one). The coherence test is defined as a comparison test of a predefined coherence threshold against the ratio of surface area of union of two leaves against the sum of the surface areas of both leaves, i.e.:

$$R_{coh}(x, y) = \frac{SA(x \cup y)}{SA(x) + SA(y)} \tag{13}$$

The x and y are the leaf nodes to be clustered eventually. If the result of this formula is lower than the threshold R_{max}, the primitives in the leaves are considered spatially coherent and the leaves are clustered before the insertion. The authors used $R_{max} = 1.5$, which we follow. This optimization is reported to have up to 30% speedup on some of the scenes by Bittner et al. We have implemented this optimization in our thesis. In the table 5 we sum the parameters of the algorithm.

3.5.2 Parallelization schemes and reported results

Bittner et al. [BHH15] have also proposed two parallelization approaches for their incremental algorithm. The incremental construction is different from the top-down or bottom-up ones and therefore requires different kind of parallelization. Both of the approaches exploit the parallelism in batch primitive processing.

The first approach, called parallel search, executes the search for the most convenient position to insert a leaf into for several leaves in parallel using multiple threads. A batch of leaves is processed, and the search is performed for each of the leaves in the batch. To prevent conflicts upon eventually inserting in the same place, the insertion is performed sequentially. Due to this sequential part, the algorithm does not provide a linear speedup. The authors therefore proposed a second method, in which the primitives are divided into larger batches than in the case of the first method and a local BVH is constructed for the primitives in each batch in parallel. These local BVHs are given to another thread when ready and it does insert them into the resulting BVH. For both of these phases the authors propose to use their incremental algorithm. However, we have not attempted these parallel methods.

3.5.3 Reported results

The authors evaluated the proposed algorithm and the parallel versions on a set of 9 test scenes, 4 of which were individual objects and the other 5 architectural scenes. They reported the costs of the hierarchies in charts, which also describe the progress of the insertion. They compared their results to the full sweep method and spatial median method. They evaluated the versions of the incremental algorithm with and without the Insertion-based optimization and the two parallel versions.

They reported the sequential incremental method to be significantly faster in construction than the full sweep algorithm. They also compared the proposed methods, reporting the parallel search version to be from 15 to 50% faster than the sequential version using the optimization algorithm. The block parallel method is reported to be up to 5 times faster than the respective sequential version. The authors also report the cost of the sequential method with updates to be usually about 10% lower than using the full sweep method. On the other hand, for some scenes with more regular structure such as the Happy Buddha or Armadillo scene, the algorithm is reported to yield slightly higher costs when compared to the full sweep.

3.6 Parallel Locally-Ordered Clustering (PLOC)

The bottom-up BVH construction algorithms have the potential to build hierarchies of high quality. The methods described by Walter et al. [Wal+08] and Gu et al. [Gu+13] use the exact and approximate agglomerative clustering respectively. The algorithm of Walter et al. [Wal+08] is difficult to parallelize and the AAC is suitable for multi-core parallelization on a CPU. The motivation of the work of Meister and Bittner [MB18] was to apply the agglomerative clustering on a many-core GPU, which has different properties and requires a different approach than the multi-core CPU algorithms. They proposed a novel parallel agglomerative clustering algorithm exploiting a higher degree of parallelism than the AAC and more suitable for a GPU.

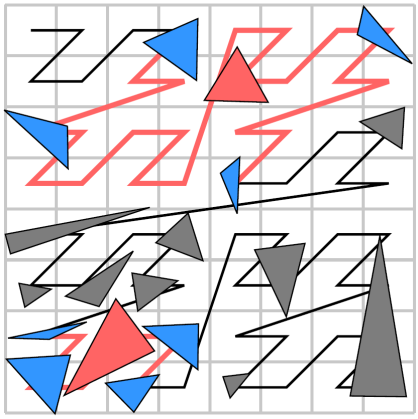


Figure 16 The approximate nearest neighbor search with radius 2. Courtesy of Meister and Bittner [MB18].

3.6.1 The algorithm

The proposed algorithm is based on two core principles. First, the locally-ordered clustering is applied on high number of clusters in parallel. This means that the set of clusters is divided in chunks and each chunk is processed by a single thread on a GPU. Second, the search for the closest cluster is approximate, which decreases the $O(n^2)$ complexity of the exact search and allows it to be faster.

The approximate search proposed by Meister and Bittner exploits the sorting of primitives using the Morton codes. Since the primitives are sorted along the Morton curve, the algorithm searches for the closest cluster for the query one in its neighborhood on the curve. More specifically, the search is done in a neighborhood defined by certain radius, which is a parameter of the algorithm. This principle can be seen on figure 16, taken from [MB18].

For each query cluster an approximate nearest neighbor is found by minimizing the distance function. We recall that various distance functions can be used (if they obey the non-decreasing property), but Meister and Bittner [MB18] use the surface area of the union of the two clusters' bounding boxes, which we follow.

After the approximate nearest neighbor search, the proposed algorithm merges the clusters that are found as mutual nearest neighbors. A question rises, whether such a pair of clusters always (i.e. after each approximate nearest neighbor search phase) exists. For any cluster x the nearest neighbor procedure finds the closest cluster y with respect to the distance function. Three situations may occur after that, which all are implied by the non-decreasing distance function. First, cluster x will be found as the closest cluster for y, in which case these will be clustered. Second, a different cluster z will be found as closest to y, in which case these situations will relate to y and z (and it means that x is further than z with respect to the distance function). Or third, cluster z will be found as the nearest neighbor of y and the distance between them will be the same as between x and y. In this (rather rare) situation, the authors propose to cluster the two clusters with smaller indices in the cluster array to solve this conflict. This and the definition of the distance function imply that there will always be at least one pair of mutual nearest neighbors in the iteration. This is also more described in the work of Meister and Bittner [MB18].

When the mutually corresponding clusters are clustered, the new cluster is inserted into the cluster buffer on the place of one of the former lower index. As the authors describe, the place for the new cluster could be also found by searching for its position

Abbreviation	Parameter description
r	Nearest neighbor search radius

Table 6 Parameters of the PLOC algorithm.

in the sorted cluster sequence (according to the Morton code). They propose not to do it, as this search could be expensive and inserting the new cluster on the place of the first index provides sufficient results.

The PLOC algorithm starts by computing the Morton codes of the primitives using the centroids of their bounding boxes. The radix sort algorithm is run after that. The subsequent core of the algorithm consists of three main phases that are executed iteratively. The first one is the approximate nearest neighbor search initiated for every currently existing cluster. The merging phase as described above comes after that. The merging conflicts (i.e. situations, where two different threads are to cluster the same pair of clusters) are avoided by having the thread processing the cluster with smaller index in buffer execute the merge. The new cluster is put into the position of the first former one and the second former one is marked as invalid.

Described as such, the algorithm would leave the cluster buffer with invalid clusters, which would make the next nearest neighbor search impossible or at least difficult to deal with. That is why the third (and last) phase of the algorithm performs the cluster buffer compaction. It moves the clusters into the new places in the cluster buffer such that there will not be empty spaces (i.e. invalid clusters). The algorithm calculates an exclusive prefix scan on all currently existing clusters to determine their new positions. The subsequent write to the new positions would not be possible without making a temporary copy of the cluster buffer, which would be inefficient. Therefore the authors use two cluster buffers, one for the clusters that existed before each iteration and one for the new set of clusters (that are result of the iteration). The buffers are swapped after each iteration. We follow this solution and store the two arrays of clusters in the one array of double length.

The algorithm runs in the iterations as described above until only one cluster (the root of the BVH) is left. The three phases (approximate nearest neighbor search, merging and compaction) are executed in parallel and are separated by barriers. After the execution of the core of the algorithm, the authors propose to execute the compaction of the BVH. The compaction has been described by Bittner et al. [BHH13] and Meister and Bittner [MB18] describe their GPU implementation of the compaction. We describe the compaction proposed by Bittner et al. [BHH13] in section 3.4.

We can see that the principles described in the work of Meister and Bittner [MB18] are suitable for both GPU and CPU. Though the authors implemented the proposed algorithm for the GPU using CUDA, the multi-core CPU implementation is also possible. We have implemented the PLOC algorithm for a CPU.

Similarly to other described methods, we sum the parameters of the algorithm in table 6.

3.6.2 Reported results

Meister and Bittner [MB18] implemented the PLOC algorithm in CUDA 7.5. They evaluated three settings of the algorithm with regards to the radius r used for the approximate nearest neighbor search, namely $r = 10$, $r = 25$ and $r = 100$. They also used the compaction algorithm after the core of the PLOC algorithm has been executed. As reference methods, they used LBVH proposed by Karras [Kar12], HLBVH proposed

by Garanzha et al. [GPM11], ATRBVH propsed by Domingues and Pedrini [DP15] and AAC. They used 30-bit Morton codes for both LBVH and HLBVH methods.

For the computation of the costs of BVHs, they used the traversal constant $c_T = 3.0$ and the ray-triangle intersection constant $c_I = 2.0$. They evaluated the algorithm as well as the referenced algorithms on nine scenes, three of which were individual objects and six were architectural models.

Meister and Bittner [MB18] report lower BVH costs and higher trace speeds for six of the nine scenes compared to the ATRBVH method., which itself has lower BVH costs than the other reference methods for all nine scenes. In particular, the PLOC algorithm achieved the lower costs mainly for the architectural scenes with large numbers of triangles, i.e. above 4305k triangles (including; Manuscript scene).

The construction times of the PLOC algorithm were faster than ATRBVH times in seven of nine scenes for the $r = 10$ and $r = 25$ configurations. The authors also report that the BVH costs based on the radius stabilize very quickly even for small values of r.

The authors have also compared the proposed PLOC algorithm to the AAC method. They used the publicly available implementation of AAC, which is sequential. The build times they reported were originally produced by the sequential method and then divided by the number of the cores of the computer that was used to measure on (the number of cores was 4). The times therefore represent the hypothetical AAC implementation with linear speedup. The authors report very similar BVH costs, but the proposed PLOC algorithm was significantly faster in the hierarchies construction, the only exception being the AAC-Fast setting for Conference scene.

3.7 Extended Morton Codes

The use of the Morton codes is a part of many state of the art BVH construction algorithms. Recall that the Morton codes induce a spatial sorting along the space-filling Morton curve and that such sorting of scene primitives is done by sorting the Morton codes generated from every primitive centroid. The use of the Morton codes thus provides an fast approximate to sort the primitives. This technique is used in the two of the algorithms we chose to implement, namely the Aproximate Agglomerative Clustering (AAC, Gu et al. [Gu+13]) and Parallel Locally-Ordered Clustering (PLOC, Meister and Bittner [MB18]), but also in e.g. LBVH (Lauterbach et al. [Lau+09]) and HLBVH (Garanzha et al. [GPM11]).

The sorting based on Morton codes has a direct impact on the quality of the resulting bounding volume hierarchy constructed by the respective algorithms. In the case of poor coherency of the primitives in the sorted sequence the BVH quality is lower. The improvement of the Morton codes such that the sorting based on them would yield primitive sequences with higher coherency of the subsequent elements and thus lead to higher BVH quality was the motivation of Vinkler et al. [VBH17].

3.7.1 Extending the Morton codes

They identified several situations when the coherency induced by the Morton codes could be increased and proposed appropriate improvements that are used in the generation of Extended Morton codes. The undeniable advantage of their technique is that it aims to increase the BVH quality by modifying the Morton code based sorting part of the algorithms only, leaving the construction part unchanged. Namely, the modification is to compute a slightly different codes, which also does not increase the

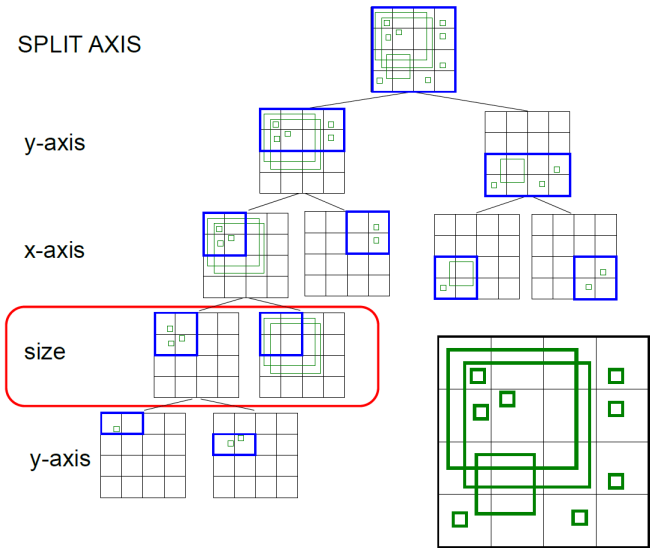


Figure 17 The splitting using coordinate axes and primitive sizes. Courtesy of Vinkler et al. [VBH17].

computation time significantly and is easy to implement and incorporate to the existing BVH construction algorithms.

Vinkler et al. [VBH17] identified four situations in which the Morton codes could be improved and proposed these improvements in their Extended Morton codes generation algorithm. The first improvement is not to encode only the 3D coordinates of the centroids but the size of the respective objects as well. This follows the situation in which two objects with spatially coherent centroids are clustered, but their bounding boxes are of highly different size. In this case, it would be more convenient to cluster the smaller object with another spatially coherent one with a bounding boxes of similar size as is the objects'. In other words, by clustering objects based also on the sizes, the bounding boxes of the respective BVH nodes can be also smaller. The authors proposed to use the length of the diagonal of the primitive's bounding box when encoding it's size. They proposed to normalize the length by the length of the diagonal of the bounding box of the scene. The simplest form of extending the Morton codes is to interleave 4 quantized coordinates (x, y, z, s), where s encodes the primitive size, instead of 3, i.e. (x, y, z). In this way, the first three splits of the code sorting are based on the coordinates and the fourth is based on the primitive sizes. The principle can be seen on figure 17, taken from [VBH17]. As can be seen, the third split is based on the size, inserting the smaller primitives to the left child and the larger to the right one.

The second improvement proposed by the authors is not to stay with the regular alternation of the split axes (x, y, z, s), but to split using the size axis s fewer times than using the coordinate axes x, y and z. This would allow to count for the subdivision based on the size while prefering the subdivision based on the spatial coherency a bit more. The authors propose to inject the size bits only every seventh bit instead of every fourth.

The third improvement proposed by Vinkler et al. [VBH17] is to adapt the axis order to the actual dimensions of the scene, i.e. not to interleave the quantized coordinates in the original (x, y, z, s) order but sort the axes based on the scene dimensions in the descending manner. This means that the primitives are subdivided using the axes with the largest spatial extent first during the hierarchy construction.

In their fourth proposed improvement the authors further generalize the axes order

as they propose to use a variable number of bits for each axis. They also propose to interleave the resulting code in such order that every bit leads to a subdivision in the currently largest axis. As a result, the axes do not need to alternate in the code regularly. The authors also note that the resulting code using this fourth improvement is no longer a Morton code as the axes do not alternate regularly. They also propose to combine this approach with injecting the size bit every fourth or seventh bit.

Choosing the axis for every bit in the code according to the largest extent can be more convenient for scenes that extent mainly in two dimensions such as terrain scenes or even in one dimension, imagine e.g. a scene with a single long street. The decision of choosing the largest axis for the bits other than size bits comes from maintaining a auxiliary bounding box and computing the axis of largest extent of this box. The box is initialized as a copy of the scene bounding box and for every bit (not including the size ones) the axis of largest extent is computed, written as the axis for the code bit, and then the box is halved in this axis. Alternatively, one can use a three dimensional vector encoding the scene dimensions computed from the scene bounding box and maintain this vector by just using the coordinate with the largest value as the desired axis and then halving this coordinate.

The authors also provided two pseudocodes, each describing a different 64-bit extended Morton code. The first code, also named EMC-64-sort, uses the improvements of encoding primitive size every fourth bit (first improvement) as well as to adapt axis order to the scene extent (third improvement). The second code, named EMC-64-var, encodes the primitive size every seventh bit (second improvement) and use the adaptive axis order and the variable bit count as well (fourth improvement). We have implemented both codes although we used a slightly different and more straight-forward approach for implementing the EMC-64-var code than in the pseudocode. In our implementation, the final bit shifts of respective parts of the code are not necessary, because the shifting for the quantized coordinates is already incorporated in their generation.

3.7.2 Reported results

Vinkler et al. [VBH17] tested the two variants of extended Morton codes described above (EMC-64-sort and EMC-64-var) and compared them to the standard 64-bit Morton code. They evaluated the use of extended Morton codes in several algorithms that use Morton codes, namely LBVH, HLBVH, ATRBVH and AAC. They tested the algorithms on nine test scenes, eight of which were architectural and one was an individual object. They evaluated construction and rendering times as well as the SAH costs of the constructed BVHs. They also reported the actual layout of the EMC-64-var code for the tested scenes, so the actual axis order is observable. The authors also evaluated the generation times of the extended Morton codes on the CPU (single core), which for both codes described above were sufficiently small. For computing the costs of BVHs, they used the triangle intersection constant $c_I = 1.0$ and node traversal cost $c_T = 1.2$.

For LBVH improved with extended Morton codes the costs improved from 0% to 52% with an average of 20%. For ATRBVH, the improvements in the costs were from -2% to 26% with an average of 7%. For AAC the costs improvements are is 11% on average and for HLBVH 16% on average. The authors also discuss that the improvements described in their work and also above are not orthogonal and therefore do not yield the same quality improvements on all scenes, but these are scene dependent.

4 Implementation and testing

In this chapter we will describe the implementation details of our thesis and then the testing strategy.

4.1 Implementation

We implemented the thesis in C++. We also tried to focus on the modern traits of C++ as much as possible given to our C++ knowledge. The thesis thus requires the C++17 standard and exploits also traits from the C++14 and C++11. The most modern trait we use are the structured bindings. The work thus requires at least the following versions of compilers:

- GCC, version 7.
- Clang, version 4.

We have also tested the implementation with Microsoft C++ compiler of version 19.16.27023.1. We have tested the successful compilation and application run with the Microsoft C++ compiler (in MS Visual Studio 2017) on MS Windows and GCC (version 7.3.0) and Clang (version 6.0.0) compilers on Linux (Ubuntu 18.04 LTS). The source code is thus portable between these two operating systems. We developed the thesis under MS Windows, but measured the results under Linux. We profiled the implementation on MS Windows using the in-built profiler in the Visual Studio.

The most of the thesis is implemented on single core and we implemented also the parallel version of the BVH using the binning approximation, for which we used C++11 threads. We implemented the hierarchies in the nanoGOLEM framework, a ray tracing framework initiated by prof. Ing. Vlastimil Havran, Ph.D., developed mostly by him and also by his former students. nanoGOLEM is a private software and thus it is not a part of our thesis submission.

4.1.1 Other minor contributions

nanoGOLEM also uses ASSIMP library for the .obj files loading. However, the ASSIMP tends to be slow when loading the scenes. It applies various kinds of post-processing procedures on them and found out that some of the scenes unfortunately require these. We therefore implemented the binary storing and loading of the post-processed objects, which is significantly faster. This approach is used instead of the ASSIMP whenever the appropriate file is present. We provide these files on the DVD only.

We also wrote a script for the export of the camera from the Blender modeling software in the work required for the nanoGOLEM. We provide the script in our submission.

We also wrote a small number of measurement Bash scripts, which we used to measure our results on Linux. These run the implemented BVHs in all the configurations and also generate Latex tables from the measured results. An arbitrary BVH implemented in our thesis can be given to the main script as an argument and also more BVHs and those of various types (basic, using the Insertion-based optimization or the parallel binning) can be given as parameters, which allows to measure more easily.

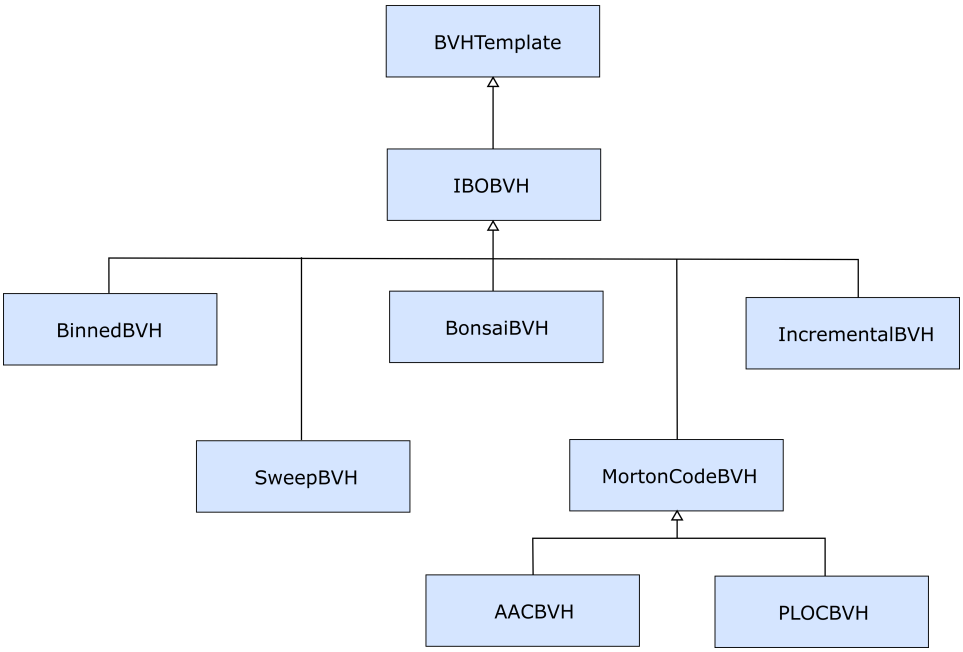


Figure 18 The architecture of our implementation.

The total number of measurements needed to be done was almost exactly 5000 (when measuring all only once), and the measurement scripts were therefore mandatory.

4.1.2 Architecture

The nanoGOLEM framework is designed to be easily extended by various types of acceleration structures. New structure is thus expected to inherit from the CASDS_BB class (abstract acceleration structure predecessor). The architecture of our implementation can be seen on figure 18.

The implementation is designed to reuse as much of code as possible and not to repeat it. The purpose of the classes is as follows:

- The BVHTemplate class is the base class of our BVHs. It contains the code needed by all our hierarchies and as such defines the basics of BVH data structures, allows to measure construction and rendering times and other metrics and provides a number of common methods such as the SAH calculation or hierarchy compaction.
- The IBOBVH class defines an optimizable hierarchy using the Insertion-Based Optimization algorithm [BHH13]. It defines the methods for this algorithm as well as for the similar Incremental construction algorithm [BHH15]. It also allows to measure the optimization time and metrics. This design also provides the ability of executing the optimization algorithm to all our hierarchies.
- The MortonCodeBVH class defines the methods related to usage of Morton codes, such as their calculation and radix sort. It also allows to calculate Extended Morton Codes [VBH17].
- Other classes implement the specific BVH construction algorithms. BinnedBVH implements both sequential and parallel versions of the binning-based construction proposed by Wald [Wal07], AACBVH implements the AAC algorithm proposed by Gu et al. [Gu+13], BonsaiBVH the Bonsai algorithm proposed by Ganestam et al. [Gan+15], IncrementalBVH the incremental construction proposed by Bittner et al. [BHH15], PLOCBVH the Parallel-locally ordered clustering algorithm

Scene name	#triangles	Scene name	#triangles
Serapis	88 040	Power Plant sec. 9	121 862
Armadillo	345 944	Fairy Forest	174 117
Angel	474 048	Sponza Large	262 267
Dragon (Chinese)	871 414	Conference Small	282 755
Happy Buddha	1 087 716	Conference Large	331 179
Turbine Blade	1 765 388	Power Plant sec. 16	365 970
Hairball	2 880 000	Soda Hall	2 169 132
Asian Dragon	7 219 045	Pompeii Ten	5 642 728
Sponza Small	76 102	San Miguel	7 838 406
Sibenik Cathedral	80 479	Power Plant	12 759 246

Table 7 Numbers of triangles in the scenes. The first 8 scenes (from Serapis to Asian Dragon) are individual objects, the other (from Sponza Small to Power Plant) are architectural scenes.

proposed by Meister and Bittner [MB18] (however, we have implemented only the sequential version of it) and SweepBVH implements the reference Full Sweep construction method as described by Wald et al. [WBS07].

4.2 Test scenes

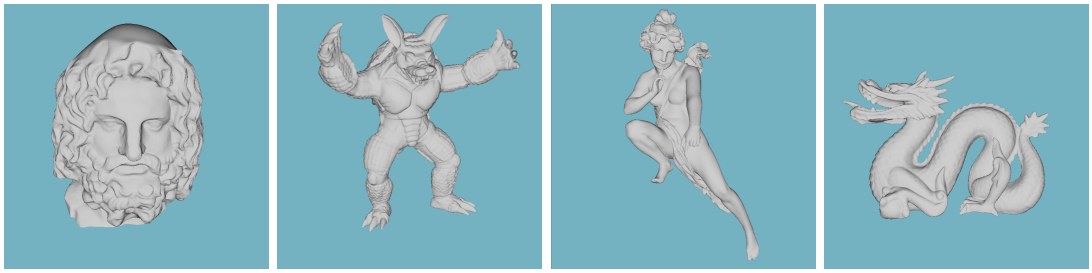
One of the goals of this thesis is to compare the chosen hierarchies among themselves. For this purpose, we have used 20 test scenes of different complexity (i.e. numbers of triangles). 8 scene contain individual objects, while the rest of them (12) are of architectural type. The numbers of triangles in the scenes can be seen in table 7. The first 8 scenes are individual models, the other 12 are architectural scenes. The snapshots of scenes, rendered using primary rays only, are in figure 19.

We have tried to find as many scenes used for evaluating of the chosen methods as possible in order to be able to compare our results with the respective works. We have added several other scenes. Most of the scenes are therefore often used for evaluation of data structures for ray tracing and they are publicly available. In particular, we have used scenes from the following archives: The Stanford 3D Scanning Repository [Uni], Morgan McGuire's Computer Graphics archive [McG17] and Georgia Institute of Technology [TM].

4.3 Verification of the methods

We have verified our implemented hierarchies in two steps. First, we examined the images of the rendered scene using our hierarchies. This verification step alone is, however, not sufficient, as it tells only that each of the rays finds the correct closest primitive, but it does not tell anything else about the topology of a BVH. In other words, the hierarchy can be built differently from the original method described in the author's work. Therefore a second verification step must be done, which allows to verify the topology.

This can be done using the costs of the hierarchies based on the SAH. We calculate the cost using the formula 4. A more thorough verification would be using the sums of surface areas of inner nodes and leaves, averages of traversal steps and intersection tests done while traversing a ray and other metrics, but unfortunately, the works usually do

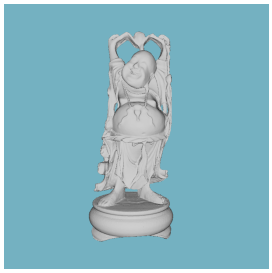


a) Serapis
88 040

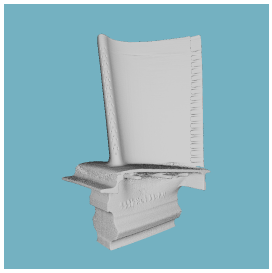
b) Armadillo
345 944

c) Angel
474 048

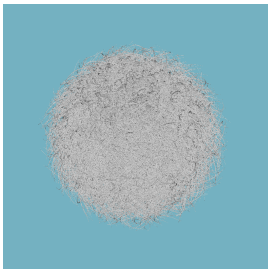
d) Dragon
871 414



e) Happy Buddha
1 087 716

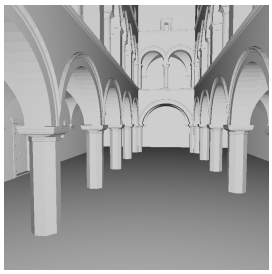


f) Turbine Blade
1 765 388

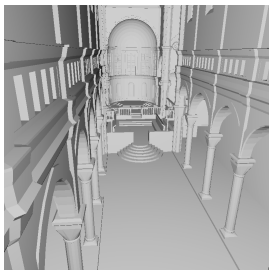


g) Hairball
2 880 000

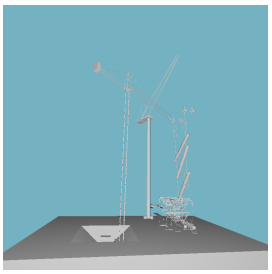
h) Asian Dragon
7 219 045



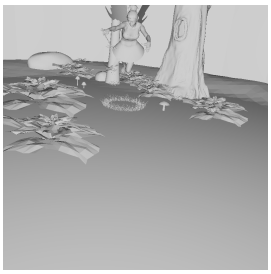
i) Sponza Small
76 102



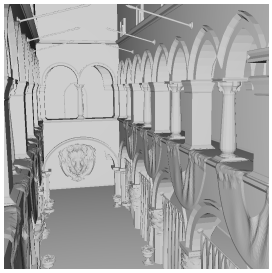
j) Sibenik Cathedral
80 479



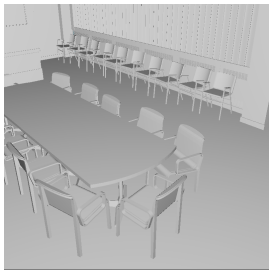
k) Power Plant sec. 9
121 862



l) Fairy Forest
174 117

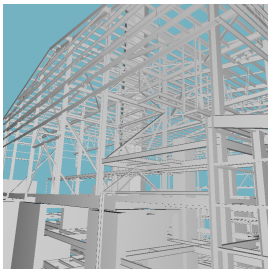


m) Sponza Large
262 267

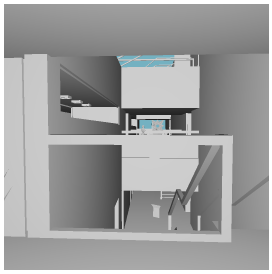


n) Conference Small
282 755

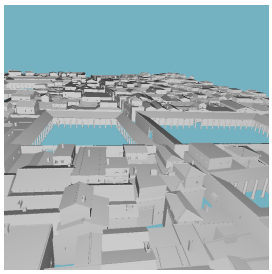
o) Conference Large
331 179



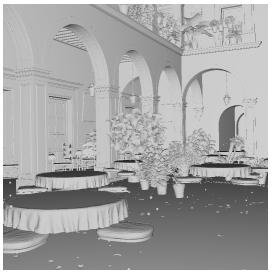
p) Power Plant sec. 16
365 970



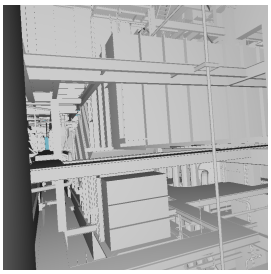
q) Soda Hall
2 169 132



r) Pompeii
5 642 728



s) San Miguel
7 838 406



t) Power Plant
12 759 246

Figure 19 The rendered images of the scenes (using the primary rays only). Under each of the images, the name of the scene is on the first line and the number of triangles in the scene is on the second line. The images were rendered using the hierarchy built with the PLOC algorithm with radius 10.

Algorithm variant	Abbreviation	Parameters
Full Sweep	FSW	-
Full Sweep + IBO	FSWO	-
BVH using binning	BIN	k=16
AAC-HQ	AHQ	δ=20, ϵ=0.1
AAC-LQ	ALQ	δ=4, ϵ=0.2
Bonsai	BP0	N=512, T=0.1, prune=false
BonsaiP	BP1	N=512, T=0.1, prune=true
BonsaiP*	BP2	N=4096, T=0.01, prune=true
Incremental	INC	batch=8000, oFrac=0.01, coh=1.5
$PLOC_{10}$	PL1	radius=10
$PLOC_{25}$	PL2	radius=25
$PLOC_{100}$	PL3	radius=100

Table 8 The algorithm variants with their abbreviations and parameters. These are used in all phases of measurement results. The FSWO abbreviation is used only in the verification results to distinguish between the method with and without applying the Insertion-based optimization (IBO). In other results (optimization, parallel and Extended Morton codes), the abbreviations describe algorithms having the appropriate attribute (e.g. using the optimization algorithm or EMCs). Moreover, the description of parameters can be seen in tables 1 (binning), 2 (AAC), 3 (Bonsai), 5 (Incremental) and 6 (PLOC).

not report these. We report these in our results section 5, so that they can be used for this purpose in the future.

Moreover, some of the papers proposing the methods we implemented do not even report the costs of the hierarchies. They report only the construction and rendering times, which, however, also do not claim anything specific about the hierarchy topology. This makes the results in the works incomparable and unfollowable. We therefore compared our results with other works, which implemented the considered methods for the comparison or as a reference. More particularly, this problem is the case of the binning-based algorithm and the AAC. We have therefore compared the results with the results in the work of Ganestam et al. [Gan+15] (the binning) and Meister and Bittner [MB18] (the AAC). Moreover, we compared the full sweep method with the results in the work of Bittner et al. [BHH13]. The other algorithms were compared with the results in the works that proposed them.

Throughout the following verification results and our measurement results in the next sections, we use abbreviations of the methods in tables. These are described in table 8. The results of our implementation can be seen in table 4.3. The costs were calculated using the constants $c_T = 3.0$ and $c_I = 2.0$ for the full sweep method, its version optimized by the insertion-based algorithm, the AAC and the PLOC. We calculated the costs for the binning and the Bonsai algorithm using the constants $c_T = 2.0$ and $c_I = 1.0$.

If we look into the related works, we can see that our costs corresponds almost entirely with the original results of the full sweep method, its optimized version, the incremental algorithm and the PLOC.

The AAC algorithm also corresponds almost entirely with an exception of the Pompeii scene and the AAC-LQ algorithm variant. Our implementation construct a BVH with higher cost. On the other hand, when using the EMC64VAR extended Morton code for the method (which is the part of our measurement), the increase in the cost disappears and the variant is close in cost to the AAC-HQ algorithm. Also, Meister and

Scene	FSW	FSWO	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Armadillo	85.0	81.9	-	-	-	-	-	-	96.9	-	-	-
Dragon	145.5	136.7	-	-	-	-	-	-	-	-	-	-
H. Buddha	165.5	155.1	-	181.4	181.8	-	-	-	233.7	175.7	174.9	177.5
Blade	190.5	179.6	-	-	-	-	-	-	-	-	-	-
Hairball	1416.4	985.6	645.1	1113.0	1137.2	644.2	614.5	617.7	1460.4	1094.5	1089.0	1084.2
Sibenik	82.4	62.1	-	-	-	-	-	-	73.3	-	-	-
P. Plant s. 9	59.2	32.6	-	-	-	-	-	-	-	-	-	-
Fairy Forest	-	-	60.6	-	-	51.9	51.1	50.3	-	-	-	-
Sponza L.	-	-	135.9	-	-	139.7	119.6	124.7	-	-	-	-
Conference S.	130.9	83.9	-	-	-	-	-	-	-	-	-	-
Conference L.	-	-	84.7	84.3	85.8	68.6	61.5	59.3	-	87.3	85.5	85.2
P. Plant s. 16	93.7	60.6	-	-	-	-	-	-	-	-	-	-
Soda Hall	216.4	140.4	-	176.5	178.1	-	-	-	180.0	175.5	173.1	175.7
Pompeii	256.2	158.8	-	189.7	240.5	-	-	-	228.7	186.0	181.9	176.2
San Miguel	-	-	119.5	139.1	142.2	115.6	108.5	105.1	157.0	147.1	141.4	137.9
Power Plant	115.7	68.4	-	77.6	79.0	-	-	-	103.4	80.6	78.0	77.4

Table 9 Costs of the hierarchies on the respective reference scenes used in the works.

Bittner suggested a bug in the AAC algorithm source code when evaluating their results (they used the publicly available implementation). The bug caused the cost result of the Power Plant scene to be higher, but our cost is sufficiently close to other algorithms (for example the PLOC). It is unusual that such a high result for the Pompeii scene as ours would be correct, but we spent massive effort to find a bug in our implementation, comparing it with the publicly available one and did not found any. On the other hand, for the other scenes the algorithm seems to work correctly.

When comparing the Bonsai algorithm, the costs of our implementation without the pruning correspond. The costs of the first pruning variant (BP1, BonsaiP in the work of Ganestam et al.) correspond on 3 of the 5 scenes, on the 2 other they are slightly higher. The costs of the second pruning variant (BP2, BonsaiP*) correspond on the Conference Large and the Fairy Forest scene. Again, we spent non-trivial effort to find a bug in our implementation, but did not find any. The Bonsai pruning algorithm is also described briefly in the work of Ganestam et al., but in fact more complex algorithm must be designed and used (which we did) to achieve the principles stated in the work. Also, when describing the pruning threshold, the authors state both that the threshold itself is used for the pruning as well as that it is multiplied by the average surface area of the mini trees, which cannot hold at the same time. We present the results with the second idea described, which we think is the correct way. We believe that the difference in the costs can be caused by these two reasons.

We did not use the Asian Dragon and the Sibenik Cathedral scene in the verification, because the results were significantly different (higher in the case of the Asian Dragon and lower in the case of the Sibenik Cathedral) not only for the Bonsai algorithm, but the full sweep and the AAC as well. This is probably due to the fact, that the scenes are not the same (they could be retessellated or edited) - they also have different numbers of triangles. We also did not use the rest of the scenes used in the work of Ganestam et al., because we were unable to find them.

The costs of the binning-based algorithm do not correspond with the costs reported by Ganestam et al., who, however, used the implementation from the Embree 2.2 ray tracing framework. This implementation use a modified binning algorithm, which constructs hierarchies with branching factor 4 and also combining with the spatial splits. In our implementation, we also made change from the original method by using the original SAH formulation when calculating the costs of split planes and leaves (instead of neglecting the implementation-specific constants), and we also stop the construction upon reaching a set of 2 triangles (i.e. we define a leaf for those). These can be the

reasons for the difference in the costs. The verification of the binning method proved to be difficult because surprisingly we found no other work which would report the costs of the hierarchies built using the method (including the original work of Wald [Wal07]), only several works reporting the construction and rendering times. Even here, we spent non-trivial effort debugging and testing the method, but did not find a mistake.

To sum up the verification, the full sweep algorithm, the insertion-based optimization, the PLOC and the incremental algorithm seem to be implemented the same as the original methods. The Bonsai and binning algorithms may have slight different design details. The AAC works correctly with the except of the Pompeii scene and using the AAC-LQ variant, which is, however, compensated by using the EMC64VAR extended Morton code.

4.4 Measurement strategy

For the comparison of the implemented BVH algorithms, we have done a four-phase measurement. The phases were:

1. Construction algorithms (single core implementations). We measured all parameter configurations described in the works proposing them.
2. Insertion-based optimization algorithm executed on several chosen hierarchies (built with specific construction algorithms). We have chosen the full sweep, binning-based and incremental algorithms.
3. Parallel construction algorithms. In this section we measured the construction times of the parallel binning-based algorithm.
4. Impact of the Extended Morton codes on the BVH construction (the EMCs relate to AAC and PLOC construction algorithms only).

The first phase was measured on three different types of rays: only primary rays (ray casting), primary and shadow rays, random rays. We measured construction and rendering times (and for the Insertion-based optimization algorithm also the optimization times) as well as the other metrics. We measured real and user times and report both of them. All the times reported in the next section are obtained by averaging 5 time measurements. We will now outline all the metrics considered for the measurement, but not all of these are reported for all of the measurement phases. The metrics for construction algorithms are:

- Cost of the hierarchy based on surface area heuristic, calculated according to the formula 4.
- Number of references to triangles per single leaf averaged to all the leaves.
- Number of traversal steps performed by a ray in average to all the cast rays.
- Number of intersection tests performed by a ray in average to all the cast rays.
- Sum of the surface area of inner nodes divided by the surface area of the whole scene.
- Sum of the surface area of leaves multiplied by the numbers of triangles they contain (we calculate surface area of a leaf and multiply it by the number of contained objects) divided by the surface area of the whole scene.

In the first measurement (single core construction) phase we report all the metrics as well as the construction and rendering times. In the third phase (parallel construction) we report only the construction times. In the fourth measurement (impact of the Extended Morton codes) we report all the metrics as well as the construction and rendering times. The second phase evaluates the Insertion-based optimization and here we report the numbers of: traversal steps and intersection tests as well as the

rendering times (same as for other phases) plus the following metrics measured after the optimization: hierarchy costs, numbers of references per leaf, sums of surface areas (both for inner nodes and leaves) and the time needed to execute the optimization. Some of the results are presented in the appendix A and we refer to it in the appropriate parts of the text.

For the calculation of all hierarchy costs reported in this thesis as well as all the parts using the SAH, we used the traversal constant $c_T = 3.0$ and the intersection constant $c_I = 2.0$.

For the shadow rays, we always construct one light source in the scene above the camera.

We have constructed the random rays using the algorithm proposed in the work of Havran et al. [HPP00] using a slightly modified version of the authors' source code. The algorithm generates points uniformly distributed on a bounding sphere enclosing the scene and creates the rays using pairs of these points. The center of the sphere is the center of the bounding box of the scene and the radius is set such that all centroids of the bounding boxes of the triangles in the scene are contained in the sphere. We used 10 bands partitioning the bounding sphere.

The results were measured by shooting 800x800 primary (or random) rays (and eventual other shadow rays). We have measured the results on a computer with Intel core i7-4500U CPU (running at base frequency of 1.8GHz with 2.4GHz turbo boost) with two physical cores and hyper-threading. The RAM capacity was 16GB. The measurement was done in Ubuntu 18.04LTS operating system. The application version used for measurement was compiled using GCC compiler of version and using a -O2 flag.

5 Results

We will now present the results measured according to the strategy presented in the previous chapter. The results are divided into four respective sections - the basic results, the results of applying the Insertion-based optimization method, the results of the parallel binning method and the results of applying the Extended Morton codes in the BVH construction.

5.1 First phase - basic measurement

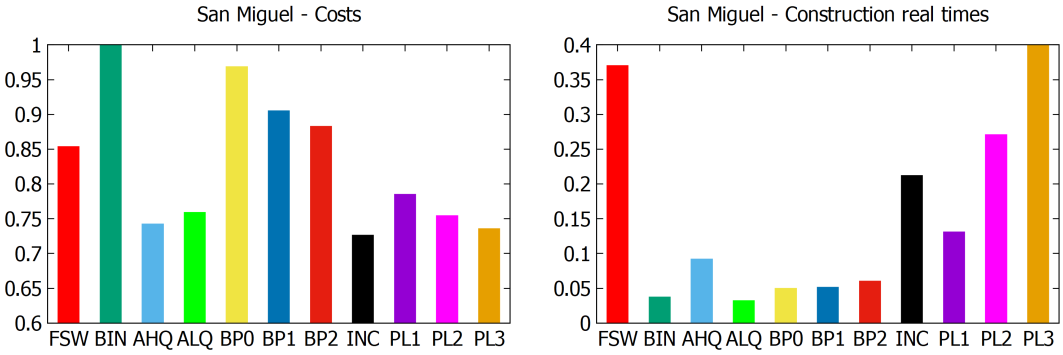
In the first phase, we measured the basic metrics for all construction algorithms using three types of rays (primary, primary and shadow, random). We used the parameter settings of the algorithms as proposed in the respective works, which leads to eleven different algorithm-variants in total. More specifically, the settings of the respective algorithms can be seen in table 8.

The top-down algorithms (full sweep, binning-based) and Bonsai can terminate the construction in an arbitrary branch also based on the surface area heuristic. The bottom-up methods (AAC, PLOC) use the compaction algorithm also based on the SAH after the construction. The resulting BVHs thus do not contain exactly one triangle per leaf. We have therefore used the compaction algorithm also on the Incremental algorithm proposed by Bittner et al., which would otherwise store only one triangle per leaf. The purpose of this is to have a similar construction strategy for all the methods.

The costs based on the SAH of all the BVHs can be seen in table 10. In the tables 11 and 12 we report the construction times (real and user). The following results are obtained by using the primary rays only. In the tables 13 and 14 we can see the real and user rendering times, in table 15 we can see the average number of traversal steps per ray, in table 16 the average number of intersection tests per ray. Next results are presented table 17 (average number of references per leaf), 18 (the sum of the surface area of inner nodes divided by the surface area of the whole hierarchy) and in table 19 (the sum of multiplying the surface area and the number of primitives in each leaf, also divided by the surface area of the whole hierarchy). We also present the bar charts of the costs, construction and rendering real times on San Miguel and Happy Buddha scenes. The results in charts are relative to the respective worst result in the measurement. We also measured the respective metrics using the primary+shadow rays and the random rays. The results of these can be seen in the appendix A.

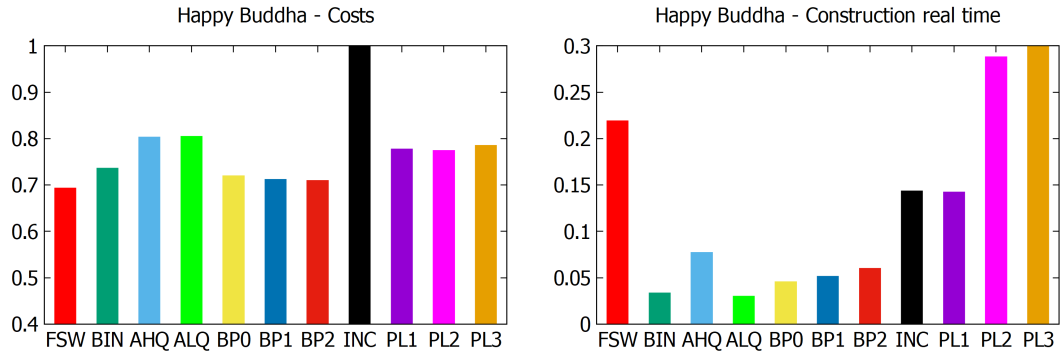
5.1.1 Individual objects

In general, the top-down methods and the Bonsai algorithm yield lower costs for the scenes containing individual objects than the bottom-up methods. However, the top-down methods higher lower costs on architectural scenes. The Incremental algorithm yields costs similar or higher than the bottom-up methods on the individual objects. More specifically, the results are similar for 4 scenes (Serapis, Armadillo, Angel and Hairball) but significantly higher for the other 4 scenes (Dragon, Happy Buddha, Turbine Blade and Asian Dragon).



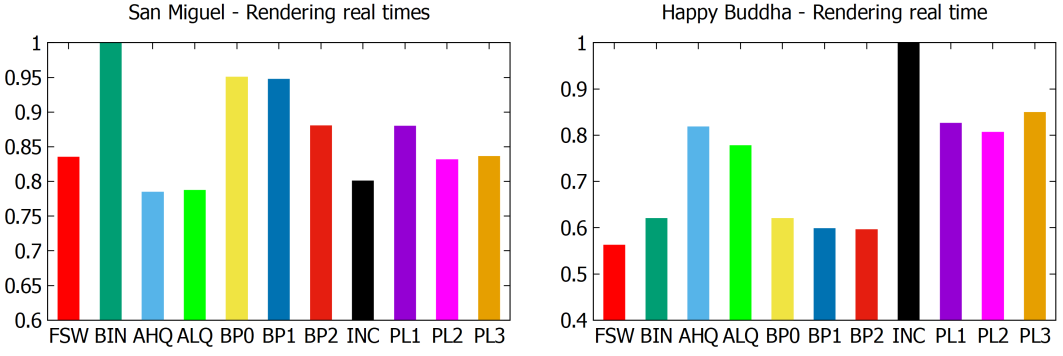
a) BVH costs. **b)** Construction real times.

Figure 20 Relative results on the San Miguel scene.



a) BVH costs. **b)** Construction real times.

Figure 21 Relative results on the Happy Buddha scene.



a) Rendering real times. **b)** Rendering real times.

Figure 22 Relative results of the rendering real times on chosen scenes.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	129.5	135.5	147.9	148.3	133.7	133.0	132.1	151.0	143.7	145.3	146.4
Armadillo	81.3	85.0	94.3	92.1	84.1	83.4	83.6	93.4	91.2	92.2	92.8
Angel	78.9	83.1	87.9	87.9	81.3	80.8	80.4	86.8	86.0	87.0	88.0
Dragon	138.2	144.8	160.0	158.4	143.7	142.1	141.4	197.5	154.0	155.8	156.1
H. Buddha	156.7	166.3	181.4	181.8	162.7	160.9	160.4	226.0	175.7	174.9	177.5
Blade	178.9	199.1	225.0	212.4	188.7	185.9	185.1	420.5	205.9	211.2	218.9
Hairball	1057.4	1088.2	1113.0	1137.2	1086.7	1041.9	1042.1	1092.8	1094.5	1089.0	1084.2
A. Dragon	90.5	96.2	104.9	102.4	93.2	92.7	92.6	148.1	101.3	101.5	103.6
Sponza S.	188.1	218.0	183.4	184.0	210.7	191.3	192.6	174.1	182.8	181.0	181.4
Sibenik	71.8	85.0	66.4	69.4	81.8	83.0	85.3	64.6	67.0	66.9	66.6
P. Plant s. 9	40.3	43.5	33.5	33.9	42.6	37.9	37.3	33.9	33.9	34.2	34.1
Fairy Forest	80.4	101.4	86.0	83.4	83.1	81.9	80.6	92.2	84.1	85.8	85.8
Sponza L.	198.5	218.9	167.8	172.2	222.6	195.0	199.9	170.7	174.9	168.7	166.8
Conference S.	111.2	144.1	89.8	90.4	114.4	107.8	108.3	88.1	92.2	92.2	91.3
Conference L.	92.6	136.1	84.3	85.8	109.1	95.7	94.8	84.1	87.3	85.5	85.2
P. Plant s. 16	74.6	97.8	69.8	72.4	84.2	73.8	73.5	66.0	71.4	70.3	69.8
Soda Hall	189.1	222.8	176.5	178.1	214.8	192.3	192.3	154.7	175.5	173.1	175.7
Pompeii	191.3	212.3	189.7	240.5	208.2	193.6	192.6	169.7	186.0	181.9	176.2
San Miguel	160.0	187.4	139.1	142.2	181.5	169.6	165.4	136.1	147.1	141.4	137.9
Power Plant	85.9	120.6	77.6	79.0	94.7	85.7	85.1	75.3	80.6	78.0	77.4

Table 10 Costs of the BVHs based on the surface area heuristic. The abbreviations of the methods can be seen in table 8.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.467	0.090	0.191	0.074	0.133	0.144	0.172	0.347	0.326	0.659	2.336
Armadillo	2.367	0.363	0.755	0.294	0.509	0.549	0.670	2.156	1.163	2.176	7.204
Angel	3.856	0.580	1.046	0.410	0.757	0.807	1.002	3.182	1.835	3.560	12.442
Dragon	5.807	0.889	1.966	0.747	1.211	1.358	1.569	3.851	3.595	7.178	24.998
H. Buddha	7.161	1.091	2.520	0.979	1.496	1.684	1.963	4.691	4.652	9.424	32.719
Blade	10.774	1.421	3.865	1.411	2.234	2.486	3.026	6.544	5.123	10.196	34.868
Hairball	22.801	2.432	7.021	2.649	3.332	3.893	4.993	13.717	9.670	18.093	61.253
A. Dragon	63.153	7.575	16.254	6.235	9.916	11.122	13.432	38.653	21.451	41.700	140.895
Sponza S.	0.264	0.060	0.173	0.059	0.092	0.098	0.115	0.160	0.251	0.513	1.808
Sibenik	0.323	0.064	0.188	0.062	0.099	0.103	0.122	0.227	0.279	0.596	2.527
P. Plant s. 9	0.499	0.083	0.284	0.090	0.142	0.143	0.170	0.328	0.402	0.841	2.904
Fairy Forest	0.962	0.167	0.389	0.136	0.243	0.254	0.294	0.518	0.603	1.300	4.817
Sponza L.	1.540	0.246	0.619	0.220	0.362	0.384	0.440	0.776	0.875	1.771	6.215
Conference S.	1.359	0.252	0.660	0.222	0.357	0.368	0.444	1.288	0.835	1.769	6.724
Conference L.	1.720	0.272	0.730	0.250	0.421	0.430	0.523	1.619	0.899	1.917	7.133
P. Plant s. 16	1.520	0.195	0.859	0.273	0.399	0.411	0.494	0.906	1.125	2.327	8.140
Soda Hall	14.140	1.812	5.455	1.880	2.770	2.895	3.462	7.397	7.104	14.857	54.273
Pompeii	39.693	5.297	13.967	4.841	7.567	7.777	9.171	24.531	19.772	41.357	149.326
San Miguel	79.849	8.107	19.809	6.958	10.783	11.065	13.003	45.742	28.275	58.376	215.727
Power Plant	114.059	11.363	32.087	10.365	16.148	16.662	19.374	80.893	33.299	69.240	242.036

Table 11 Construction, sequential, real time [s]. The abbreviations of the methods can be seen in table 8.

For the individual objects, the best results are constantly achieved by the full sweep construction with the exception of the Hairball scene. However, we are interested mainly in the other algorithms and, as expected, the full sweep method achieves these results in significantly higher construction times compared to the other methods. Apart from the full sweep, the best costs on these scenes are continuously achieved by the two variants of Bonsai algorithm with pruning. More specifically, the BP1 variant achieves the best costs on two scenes (Armadillo and Hairball), while the BP2 variant achieves the best costs on the other 6 scenes. The results of the Bonsai variants with pruning are interesting even more, because the construction times are lower than the times of all the PLOC variants and the AAC-HQ while achieving the lower costs. When compared to the AAC-LQ, the construction times of the Bonsai variants with pruning are higher

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.465	0.087	0.184	0.071	0.132	0.141	0.168	0.343	0.322	0.655	2.331
Armadillo	2.357	0.358	0.721	0.276	0.498	0.537	0.658	2.141	1.142	2.158	7.189
Angel	3.845	0.565	1.010	0.387	0.745	0.787	0.988	3.155	1.806	3.542	12.415
Dragon	5.790	0.864	1.894	0.700	1.188	1.330	1.542	3.810	3.551	7.143	24.957
H. Buddha	7.126	1.060	2.431	0.920	1.461	1.642	1.923	4.642	4.587	9.361	32.652
Blade	10.732	1.373	3.714	1.321	2.173	2.425	2.957	6.453	5.035	10.102	34.757
Hairball	22.711	2.367	6.730	2.442	3.239	3.804	4.899	13.582	9.451	17.895	61.045
A. Dragon	62.933	7.359	15.515	5.702	9.627	10.785	13.116	38.304	20.919	41.155	140.348
Sponza S.	0.262	0.056	0.167	0.058	0.086	0.098	0.113	0.157	0.246	0.509	1.802
Sibenik	0.321	0.063	0.183	0.058	0.095	0.098	0.119	0.221	0.272	0.593	2.520
P. Plant s. 9	0.497	0.079	0.272	0.083	0.134	0.138	0.166	0.326	0.394	0.840	2.900
Fairy Forest	0.959	0.162	0.368	0.131	0.235	0.249	0.289	0.507	0.592	1.294	4.807
Sponza L.	1.532	0.238	0.597	0.205	0.350	0.370	0.430	0.764	0.864	1.757	6.192
Conference S.	1.356	0.247	0.631	0.209	0.349	0.360	0.434	1.271	0.817	1.752	6.710
Conference L.	1.709	0.264	0.700	0.234	0.410	0.418	0.515	1.601	0.879	1.899	7.120
P. Plant s. 16	1.514	0.185	0.830	0.254	0.379	0.400	0.479	0.891	1.106	2.303	8.119
Soda Hall	14.057	1.767	5.228	1.716	2.695	2.825	3.394	7.286	6.942	14.697	54.104
Pompeii	39.498	5.149	13.362	4.404	7.355	7.538	8.959	24.234	19.326	40.923	148.882
San Miguel	79.594	7.896	19.027	6.392	10.499	10.735	12.680	45.352	27.705	57.791	215.174
Power Plant	113.471	11.069	30.733	9.340	15.622	16.088	18.803	80.227	32.253	68.228	240.980

Table 12 Construction, sequential, user time [s]. The abbreviations of the methods can be seen in table 8.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.385	0.415	0.526	0.492	0.416	0.401	0.397	0.518	0.490	0.512	0.506
Armadillo	0.288	0.307	0.403	0.372	0.311	0.299	0.298	0.400	0.389	0.431	0.385
Angel	0.273	0.285	0.353	0.351	0.292	0.277	0.277	0.372	0.327	0.378	0.348
Dragon	0.300	0.327	0.475	0.426	0.328	0.316	0.318	0.537	0.392	0.442	0.422
H. Buddha	0.235	0.259	0.342	0.325	0.259	0.250	0.249	0.418	0.345	0.337	0.355
Blade	0.290	0.335	0.583	0.446	0.328	0.314	0.315	0.703	0.439	0.440	0.493
Hairball	1.365	1.534	2.155	2.379	1.575	1.425	1.455	2.052	1.694	1.768	1.858
A. Dragon	0.476	0.557	0.695	0.629	0.515	0.524	0.519	0.938	0.657	0.686	0.739
Sponza S.	1.173	1.460	0.881	1.075	1.057	1.088	1.187	0.972	0.868	0.932	0.880
Sibenik	0.878	1.184	0.958	0.991	1.343	1.363	1.182	0.926	0.931	0.925	0.942
P. Plant s. 9	0.243	0.391	0.227	0.222	0.300	0.253	0.242	0.216	0.235	0.231	0.228
Fairy Forest	0.585	1.331	0.627	0.604	0.651	0.634	0.584	0.792	0.638	0.615	0.640
Sponza L.	1.721	1.831	1.597	1.685	1.853	1.603	1.589	1.551	1.859	1.538	1.649
Conference S.	0.737	1.184	0.684	0.717	0.799	0.734	0.710	0.690	0.705	0.682	0.682
Conference L.	0.604	0.946	0.586	0.567	0.728	0.651	0.630	0.598	0.575	0.628	0.585
P. Plant s. 16	1.329	2.359	1.575	1.551	1.753	1.337	1.361	1.455	1.496	1.489	1.561
Soda Hall	1.237	1.357	1.588	1.571	1.369	1.248	1.295	1.127	1.328	1.266	1.478
Pompeii	1.345	1.580	1.581	2.131	1.512	1.516	1.421	1.474	1.746	1.782	1.606
San Miguel	2.083	2.495	1.957	1.964	2.371	2.363	2.197	1.997	2.194	2.074	2.085
Power Plant	1.947	3.719	2.886	2.584	2.460	2.124	2.049	1.920	2.894	2.106	2.717

Table 13 Rendering, primary rays, real times [s]. The abbreviations of the methods can be seen in table 8.

on 7 out of 8 scenes (the exception being the Angel scene).

When comparing the results of the binning-based algorithm and the Bonsai variant with better results (in terms of cost; taking Bonsai variant as the reference method) throughout the individual objects, we can see that the costs of the binning-based method are greater up to 3.9% for 6 scenes, but also greater even more for the Turbine Blade and Hairball scenes. The advantage of the binning-based method over all three Bonsai variants is the construction speed. Another interesting result of the binning-based method is that the costs are constantly lower 7 of 8 individual objects (with the exception of the Hairball scene) than when using the bottom-up methods, the AAC and the PLOC, while achieving lower construction times than all the PLOC variants and the AAC-HQ. However, the binning is slower in construction than the AAC-LQ on 7

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.384	0.413	0.526	0.492	0.415	0.400	0.397	0.517	0.491	0.511	0.505
Armadillo	0.288	0.306	0.403	0.371	0.308	0.299	0.297	0.400	0.389	0.431	0.385
Angel	0.273	0.285	0.352	0.350	0.291	0.277	0.277	0.372	0.326	0.377	0.348
Dragon	0.300	0.327	0.474	0.426	0.326	0.315	0.317	0.534	0.391	0.441	0.421
H. Buddha	0.235	0.259	0.341	0.324	0.259	0.250	0.249	0.417	0.345	0.336	0.355
Blade	0.290	0.336	0.582	0.445	0.328	0.313	0.314	0.702	0.438	0.440	0.492
Hairball	1.363	1.533	2.154	2.379	1.574	1.424	1.455	2.051	1.694	1.766	1.858
A. Dragon	0.475	0.557	0.695	0.629	0.512	0.524	0.518	0.938	0.657	0.685	0.739
Sponza S.	1.172	1.459	0.881	1.074	1.056	1.088	1.186	0.971	0.868	0.932	0.879
Sibenik	0.877	1.182	0.957	0.991	1.343	1.362	1.182	0.926	0.931	0.925	0.942
P. Plant s. 9	0.241	0.390	0.226	0.222	0.299	0.254	0.242	0.216	0.233	0.231	0.228
Fairy Forest	0.585	1.327	0.627	0.603	0.651	0.633	0.584	0.791	0.638	0.615	0.639
Sponza L.	1.721	1.829	1.596	1.685	1.853	1.603	1.589	1.550	1.858	1.537	1.649
Conference S.	0.737	1.183	0.684	0.717	0.799	0.733	0.709	0.690	0.704	0.681	0.681
Conference L.	0.604	0.946	0.586	0.567	0.727	0.650	0.630	0.598	0.574	0.628	0.585
P. Plant s. 16	1.328	2.357	1.574	1.551	1.753	1.336	1.361	1.454	1.496	1.488	1.561
Soda Hall	1.237	1.357	1.588	1.570	1.369	1.248	1.294	1.127	1.328	1.265	1.478
Pompeii	1.345	1.579	1.580	2.130	1.512	1.515	1.419	1.474	1.745	1.782	1.605
San Miguel	2.082	2.492	1.956	1.963	2.369	2.361	2.196	1.997	2.194	2.072	2.085
Power Plant	1.947	3.717	2.884	2.584	2.459	2.124	2.048	1.919	2.894	2.106	2.716

Table 14 Rendering, primary rays, user times [s]. The abbreviations of the methods can be seen in table 8.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	23.2	24.1	33.6	30.9	24.4	24.0	23.7	29.1	30.2	30.7	31.5
Armadillo	17.2	17.9	24.7	22.9	18.6	17.9	18.0	20.8	23.0	24.4	23.1
Angel	14.7	15.2	19.3	19.0	15.4	15.1	15.0	16.8	17.0	18.8	18.9
Dragon	17.1	18.0	27.5	25.1	18.4	17.9	17.8	27.0	21.6	22.9	23.4
H. Buddha	12.0	12.5	17.6	16.8	12.7	12.5	12.4	19.5	16.5	15.3	17.3
Blade	16.4	16.6	31.6	24.7	17.9	17.4	17.2	38.0	22.7	23.2	26.2
Hairball	58.9	61.4	86.2	90.2	64.5	59.0	59.5	78.9	66.0	69.8	73.8
A. Dragon	22.6	23.6	31.3	28.8	23.5	23.3	23.3	37.8	27.6	28.5	30.8
Sponza S.	79.5	98.8	69.1	84.4	87.7	87.8	80.1	72.5	69.3	73.4	74.1
Sibenik	76.4	84.9	84.0	81.0	115.7	115.2	103.1	78.2	74.5	74.7	75.7
P. Plant s. 9	14.8	17.9	12.2	12.3	18.8	13.9	13.6	11.2	12.6	12.5	11.9
Fairy Forest	45.9	46.0	50.6	46.7	50.8	48.3	45.8	61.3	49.6	45.8	49.2
Sponza L.	131.8	145.9	110.2	130.7	158.6	125.3	129.0	107.9	142.9	114.6	113.2
Conference S.	51.2	59.1	42.5	43.4	55.8	49.3	49.3	38.8	42.7	42.0	40.3
Conference L.	43.3	54.7	36.2	36.1	53.8	41.5	41.7	35.7	36.0	39.6	36.2
P. Plant s. 16	102.4	115.4	123.8	122.4	132.7	95.4	103.7	97.7	117.0	116.5	117.2
Soda Hall	91.8	104.0	144.7	142.1	131.6	108.0	121.4	94.5	118.0	116.7	131.2
Pompeii	72.9	80.7	79.9	118.2	87.1	75.4	74.2	65.5	84.7	87.2	76.2
San Miguel	181.4	188.2	150.5	149.6	205.8	187.8	182.0	125.0	164.4	153.4	150.8
Power Plant	174.4	168.2	220.4	196.8	225.7	185.6	181.8	135.0	220.8	174.2	207.6

Table 15 Average number of traversal steps per ray using the primary rays. The abbreviations of the methods can be seen in table 8.

individual objects (the exception being the Hairball scene).

5.1.2 Architectural scenes

The results of the algorithms are different for the architectural scenes than they were for the individual objects. While for the individual objects the best results (in terms of the costs) were constantly achieved by the full sweep method followed by one of the variants of Bonsai algorithm using pruning, this does not hold on the architectural scenes. The full sweep method achieves the best cost only on the Fairy Forest scene (1 of 12 architectural scenes).

At first, there are two phenomenons observable on this type of the scenes. First, the bottom-up methods (AAC and PLOC) achieve better costs than the top-down methods

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	4.6	4.9	5.8	5.6	4.7	4.7	4.7	5.7	5.2	5.0	5.1
Armadillo	2.0	2.6	2.7	2.8	2.1	2.1	2.1	2.6	2.6	2.8	2.4
Angel	2.0	2.1	2.5	2.8	2.1	2.1	2.1	2.4	2.3	2.5	2.3
Dragon	2.5	2.8	3.7	3.6	2.6	2.5	2.5	4.2	2.9	3.0	3.0
H. Buddha	1.8	2.1	2.3	2.4	1.9	1.9	1.9	3.2	2.4	2.1	2.4
Blade	1.9	3.6	3.4	2.8	2.0	1.9	1.9	6.5	2.5	2.4	2.7
Hairball	44.8	48.0	56.1	66.1	46.8	43.6	45.1	64.5	47.0	47.4	48.4
A. Dragon	2.7	5.9	3.3	3.3	2.7	2.7	2.7	4.6	3.0	3.0	3.1
Sponza S.	41.7	46.1	19.2	23.4	16.7	16.3	36.0	19.9	15.9	19.3	14.6
Sibenik	12.4	30.8	13.5	16.7	20.0	20.5	18.3	11.0	15.9	13.7	15.6
P. Plant s. 9	7.7	21.0	8.3	7.6	9.7	8.0	8.1	7.9	8.6	8.4	8.6
Fairy Forest	9.3	90.4	9.3	9.6	9.4	9.1	9.2	13.5	9.4	9.3	9.4
Sponza L.	39.0	26.4	41.4	30.6	21.4	23.6	26.3	30.2	30.5	27.6	36.5
Conference S.	20.5	57.0	20.7	21.9	19.4	18.7	18.5	23.9	20.7	20.5	20.9
Conference L.	14.1	36.4	16.2	15.4	14.4	16.2	16.1	17.4	16.0	17.4	16.6
P. Plant s. 16	34.1	118.5	35.3	35.2	39.4	35.9	36.2	39.1	32.6	32.6	36.8
Soda Hall	37.3	34.9	21.2	25.3	21.7	23.7	21.1	17.7	19.7	17.1	21.8
Pompeii	45.1	56.5	49.4	56.1	47.0	49.7	49.1	57.5	56.5	58.3	54.2
San Miguel	19.1	40.0	23.0	25.2	23.8	24.4	22.4	33.7	27.2	26.5	29.8
Power Plant	19.1	203.5	36.8	43.3	19.0	18.6	18.0	33.9	33.3	22.3	36.1

Table 16 Average number of intersection tests per ray using the primary rays. The abbreviations of the methods can be seen in table 8.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	2.4	2.3	2.3	2.3	2.4	2.4	2.4	2.3	2.3	2.3	2.3
Armadillo	2.1	2.5	2.2	2.2	2.2	2.2	2.2	2.2	2.2	2.1	2.1
Angel	2.3	2.3	2.3	2.2	2.3	2.3	2.3	2.2	2.3	2.3	2.3
Dragon	2.4	2.6	2.3	2.3	2.4	2.4	2.4	2.3	2.3	2.3	2.3
H. Buddha	2.4	2.6	2.3	2.2	2.4	2.4	2.4	2.2	2.2	2.3	2.3
Blade	2.2	3.5	2.2	2.2	2.2	2.2	2.2	2.3	2.2	2.2	2.2
Hairball	5.8	5.7	5.9	5.5	5.7	5.7	5.8	7.2	5.7	5.8	6.0
A. Dragon	2.3	2.2	2.3	2.2	2.4	2.4	2.4	2.1	2.3	2.2	2.2
Sponza S.	2.8	3.2	2.7	2.7	2.8	2.8	2.8	2.6	2.7	2.7	2.7
Sibenik	2.4	3.1	2.3	2.3	2.4	2.4	2.4	2.3	2.3	2.3	2.3
P. Plant s. 9	2.9	5.0	2.7	2.8	2.9	2.9	2.9	2.5	2.7	2.7	2.8
Fairy Forest	2.6	2.7	2.5	2.4	2.6	2.6	2.6	2.4	2.5	2.5	2.5
Sponza L.	2.3	2.5	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3
Conference S.	2.5	3.1	2.4	2.4	2.5	2.5	2.5	2.3	2.5	2.5	2.5
Conference L.	2.5	3.1	2.4	2.2	2.6	2.6	2.5	2.3	2.4	2.5	2.5
P. Plant s. 16	2.9	8.6	2.7	3.0	2.9	2.9	2.9	2.7	2.7	2.7	2.7
Soda Hall	2.6	3.5	2.6	2.5	2.6	2.6	2.6	2.5	2.5	2.6	2.6
Pompeii	2.8	3.0	2.6	2.4	2.8	2.8	2.8	2.6	2.7	2.7	2.7
San Miguel	3.0	3.1	2.9	2.8	3.0	3.0	3.0	2.9	2.9	2.9	3.0
Power Plant	2.6	3.8	2.7	2.6	2.6	2.6	2.6	2.6	2.5	2.6	2.7

Table 17 Average number of references per leaf. The abbreviations of the methods can be seen in table 8.

and Bonsai. Second, the incremental algorithm, which yielded rather higher costs on the individual objects, yields much better results here. More particularly, the incremental algorithm achieves the best costs of all tested methods on 9 of 12 architectural scenes. On the Soda Hall scene, the cost achieved by the incremental algorithm is notably lower. In the costs, the incremental algorithm is in general followed by the AAC or the PLOC variants. The costs of the AAC-HQ directly follow the incremental method on 4 of 12 scenes. On two other scenes (Power plant and San Miguel) the incremental algorithm is followed by the PL3 PLOC variant and then by the AAC-HQ, which is, however, significantly faster on these scenes than the PL3. The other PLOC variants already yield larger costs on these two scenes, while the construction times of the AAC-HQ are still lower. The situation is similar also for the Sponza Large scene.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	36.8	38.7	43.1	43.0	38.2	38.0	37.7	43.7	41.8	42.4	42.8
Armadillo	24.4	24.9	28.7	27.8	25.4	25.1	25.2	28.3	27.7	28.0	28.3
Angel	23.1	24.4	26.2	26.1	23.9	23.7	23.6	25.7	25.6	26.0	26.3
Dragon	40.3	42.1	47.8	47.1	42.2	41.6	41.4	59.7	45.8	46.4	46.5
H. Buddha	45.4	48.1	53.9	53.8	47.4	46.9	46.7	68.2	52.0	51.8	52.7
Blade	53.1	53.8	68.6	64.1	56.4	55.5	55.2	133.1	62.2	64.0	66.6
Hairball	196.6	206.6	214.2	222.3	206.8	191.8	191.7	199.5	209.5	207.4	205.5
A. Dragon	27.1	28.8	32.0	31.0	28.0	27.8	27.8	46.4	30.8	31.0	31.8
Sponza S.	46.7	51.5	45.1	45.9	55.0	48.5	48.2	42.2	45.2	44.6	44.6
Sibenik	17.9	20.5	17.0	17.9	21.5	21.9	22.8	16.2	17.0	17.1	16.8
P. Plant s. 9	7.9	8.3	5.9	6.0	8.8	7.3	7.0	6.1	6.1	6.1	6.1
Fairy Forest	20.4	20.4	22.3	21.3	21.3	20.9	20.5	24.3	21.6	22.2	22.2
Sponza L.	49.1	53.9	39.1	40.6	57.7	48.5	50.0	39.6	41.3	39.6	39.1
Conference S.	29.0	27.9	21.8	22.0	30.1	27.9	27.9	20.9	22.6	22.7	22.3
Conference L.	23.1	26.4	20.4	21.0	28.6	24.2	24.0	20.1	21.4	20.8	20.7
P. Plant s. 16	18.4	20.1	17.0	17.6	21.6	18.2	18.0	15.5	17.5	17.1	17.0
Soda Hall	51.5	56.5	47.7	48.1	60.4	52.9	52.7	39.9	47.5	46.6	47.5
Pompeii	41.8	46.1	42.2	59.0	47.8	43.0	42.3	34.6	40.7	39.4	37.5
San Miguel	42.7	49.4	36.1	37.0	50.1	46.1	44.7	34.2	38.7	36.8	35.6
Power Plant	19.6	20.3	16.9	17.2	22.6	19.6	19.3	16.1	18.0	17.1	16.9

Table 18 Sum of the surface area of inner nodes divided by the surface area of the whole BVH. The abbreviations of the methods can be seen in table 8.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	9.6	9.7	9.2	9.6	9.5	9.5	9.5	10.0	9.2	9.1	9.0
Armadillo	4.0	5.1	4.1	4.3	4.0	4.0	4.0	4.2	4.1	4.0	4.0
Angel	4.8	4.9	4.6	4.9	4.8	4.8	4.8	4.8	4.6	4.6	4.5
Dragon	8.6	9.3	8.3	8.6	8.6	8.6	8.6	9.2	8.3	8.3	8.2
H. Buddha	10.2	11.0	9.8	10.2	10.2	10.2	10.2	10.7	9.8	9.8	9.7
Blade	9.8	18.8	9.6	10.0	9.7	9.7	9.7	10.5	9.6	9.5	9.5
Hairball	233.8	234.3	235.2	235.2	233.2	233.2	233.5	247.1	233.1	233.4	233.9
A. Dragon	4.6	4.9	4.4	4.6	4.6	4.6	4.6	4.4	4.4	4.3	4.1
Sponza S.	24.1	31.8	24.0	23.2	22.8	22.8	23.9	23.7	23.6	23.7	23.8
Sibenik	9.1	11.8	7.8	7.9	8.7	8.7	8.4	8.0	8.0	7.8	8.1
P. Plant s. 9	8.3	9.3	7.9	8.0	8.1	8.1	8.2	7.9	7.9	7.9	7.9
Fairy Forest	9.6	20.2	9.6	9.8	9.6	9.6	9.6	9.6	9.7	9.6	9.6
Sponza L.	25.7	28.7	25.3	25.1	24.7	24.7	24.9	26.0	25.5	24.9	24.8
Conference S.	12.1	30.2	12.2	12.2	12.1	12.1	12.3	12.6	12.2	12.1	12.2
Conference L.	11.6	28.4	11.5	11.5	11.6	11.6	11.5	11.9	11.6	11.5	11.5
P. Plant s. 16	9.7	18.7	9.4	9.8	9.7	9.7	9.7	9.8	9.5	9.5	9.4
Soda Hall	17.3	26.6	16.7	16.9	16.8	16.8	17.1	17.5	16.5	16.6	16.6
Pompeii	32.9	37.0	31.5	31.7	32.4	32.4	32.9	32.9	32.0	31.9	31.9
San Miguel	15.9	19.7	15.3	15.6	15.6	15.6	15.7	16.7	15.5	15.5	15.5
Power Plant	13.6	29.9	13.4	13.7	13.5	13.5	13.6	13.5	13.3	13.3	13.3

Table 19 Sum of the surface areas of leaves multiplied by the number of triangles per leaf, divided by the surface area of the whole BVH. The abbreviations of the methods can be seen in table 8.

The main disadvantage of the incremental algorithm is the construction speed on large scenes containing more than 5 million triangles (Pompeii, San Miguel and Power Plant scenes). On these, the construction times of the variants of the AAC and PLOC algorithms are notably lower. On the other hand, the incremental algorithm is faster in construction than the fastest PLOC variant (PL1) on 6 of the other 9 architectural scenes (i.e. the scenes of low and moderate complexity). When comparing the construction speed of the incremental algorithm to the variants of the AAC, we can see, that the AAC-HQ variant (the slower of the two AAC variants) is faster than the incremental method on 11 of 12 architectural scenes, the exception being the Sponza Small scene. The AAC-HQ is also faster on all architectural scenes than both the fastest PLOC variant (PL1) and the PLOC variant achieving the best cost (differs throughout the scenes).

We would also like to sum up the results of the Bonsai algorithm, which achieved interestingly good results on the individual objects. However, the results of the algorithm on the architectural scenes are not that good. Bonsai achieves better cost than the AAC and the PLOC variants only on the Fairy Forest scene. It is true, that both Bonsai variants with pruning achieve lower construction times than the AAC-HQ and all three PLOC variants on all architectural scenes, but they are slower than the AAC-LQ on them. Moreover, the AAC-LQ produces hierarchies with lower costs on 10 of 12 scenes, the exceptions being the Fairy Forest and the Pompeii scenes.

Another interesting fact comes from the comparison of the binning-based top-down method and the AAC-LQ. The binning truly achieves high performance when compared to other BVHs also on architectural scenes. However, the AAC-LQ is faster in construction than the binning method on 10 of 12 scenes while achieving (sometimes even significantly) better costs (with the exception of the Pompeii scene). When the high performance construction is needed for the architectural scenes, the AAC-LQ seems to be a better choice than the binning-based construction.

5.1.3 Summary

So far we have not discussed the rendering times. From the tables containing the values of the rendering times (real and user, respectively) using the primary rays, 13 and 14, we can see, that the rendering times do not exactly correspond to the costs of the hierarchies. This can can be caused by three facts implied by the use of the surface area heuristic, which is present in some form in all tested methods (driving the construction itself or the compaction of the resulting hierarchy). First, the SAH assumes that the rays traversing the scene do not intersect primitives, which does not hold. Second, we have used the traversal and intersection constants, c_T and c_I, as proposed in various works (e.g. by Bittner et al. [BHH13], $c_T = 3.0$ and $c_I = 2.0$). These were, however, not estimated on the hardware we used for the measurement. The rendering times can thus be influenced by this fact. The solution would be to estimate new constants.

There can also be a third possible influence, which was also discussed in the work of Aila et al. [AKL13]. They identified that the surface area heuristic in the original version does not predict the performance of the ray tracing perfectly, which can lead to discrepancy between the SAH-based costs and the rendering times. This can have the impact on the results presented here as well. Since the rendering times also depend on the degree of optimization of ray tracing application, we are thus more interested in the costs of the hierarchies and we will sum the results mainly based on these.

We would like to mention, as already discussed in several previous works, that

different results can be achieved for the scenes containing individual objects and the architectural ones. As implied by our results, these two types of scenes could demand different approaches. While the top-down methods and the Bonsai algorithm (which use also a top-down construction but in two phases) yielded better costs on individual objects than the bottom-up methods, the situation is inverse on the architectural scenes.

On the individual objects, based on our results, we see the Bonsai algorithm (more particularly either of the two variants using pruning) as promising. It yielded low costs of the hierarchies while still constructing them in low times compared to the other methods. When willing to trade the hierarchy quality for construction speed (i.e. to construct faster), one could also use binning-based algorithm for the individual objects. On the individual objects, the binning still achieves lower costs than the bottom-up methods (the AAC and the PLOC). Our results also imply, that the incremental algorithm is not that convenient for this type of scenes, leading to higher costs and in higher construction times than the binning and the Bonsai.

On the architectural scenes, however, bottom-up methods and the incremental method achieve better costs. When aiming for the costs, one could use the incremental algorithm, which yields the best costs on most architectural scenes but at the cost of longer construction. The other two options are the AAC and the PLOC algorithms. Of these two, the AAC allows for faster constructions (both the AAC-HQ and the AAC-LQ variants). The PLOC algorithm constructs hierarchies in higher times. When being able to trade the quality for the construction speed, the choice can be the AAC-LQ method, which allows for high performance construction. Based on the costs, both the binning method and the Bonsai algorithm yield worse results than the previously mentioned ones.

5.2 Second phase - Insertion-based optimization

In the second phase of the measurement we evaluated the results of applying the Insertion-based optimization algorithm on some of the hierarchies constructed by various algorithms. More particularly, we chose the full sweep and the incremental algorithm. Using these we have built the hierarchies such that each of the leaves contain exactly one triangle, as suggested in the work of Bittner et al. [BHH13]. We have also added the binning-based construction, which, however, can terminate the branches of the hierarchy earlier based on the SAH. To get the picture about the results against the non-optimized versions of the algorithms, we present the results relative to these in table 20. The absolute values can be seen in the appendix A. We also present the bar charts of the optimized costs of the three examined algorithms in figures 23 (Full sweep method), 24 (the binning) and 25 (the Incremental algorithm).

5.2.1 Individual objects

When comparing the impact of the optimization on the hierarchies on the scenes containing individual objects, we can see that the costs after the optimization are roughly the same throughout the optimized methods. The algorithm profiting the most from the use of the optimization is the incremental one. Without optimization, this algorithm yielded higher costs when compared to the full sweep method, sometimes significantly. The costs of BVHs constructed by the incremental method and later optimized are now similar to the costs of the full sweep method. The cost reduction of the BVHs constructed with the incremental method ranges from 8,6% to 55% with an average of 24%. The full sweep method, on the other hand, does not profit much from the

Scene	FSW	BIN	INC	FSW	BIN	INC	FSW	BIN	INC
Serapis	100.5	98.5	86.9	107	103.4	84.3	100.8	97.7	85.8
Armadillo	100.7	99.2	89.9	105.2	113.7	90.3	100.8	98.8	89.4
Angel	99.4	95.7	91.4	108.1	111.2	90.3	99.6	94.7	90.7
Dragon	98.9	97	70.7	117.7	104.6	68	99.3	96	68.7
H. Buddha	99	95.3	69.9	112.3	101.2	69.3	99.3	94.2	67.6
Blade	100.4	96.6	45	110.7	101.5	56.7	100.6	95.7	42.5
Hairball	93.2	91.3	91.2	114.8	92.9	87.8	87.2	82.6	87.7
A. Dragon	101.0	97.7	63.1	114.1	100.2	73.5	101.1	97.2	60.6
Sponza S.	84.8	77	92.1	61	59.7	84.1	77.3	67.4	86.5
Sibenik	86.5	77.3	94.6	97.3	99.2	99.6	87.2	67.3	94.4
P. Plant s. 9	80.9	81.4	97.3	91.3	84.6	101.4	70.9	66.3	95.1
Fairy Forest	94.7	92.3	83.9	97.3	132.1	79.1	92.6	86.8	79.8
Sponza L.	77.6	73.5	89.3	81.2	80.5	81.3	69.9	63.6	86.4
Conference S.	75.4	78.3	95.3	90.8	69	96.4	68.3	62.7	94.7
Conference L.	86.3	78.8	95.1	105.6	82.1	102.3	81.8	63.3	94.0
P. Plant s. 16	81.2	77.3	92.9	92.6	75.6	86.7	75.5	63.2	91.0
Soda Hall	74.2	70	90.5	69.7	76.6	78.3	68.9	60	88.5
Pompeii	83	78.9	93.2	97.1	86.4	85.1	75.8	66.8	91.0
San Miguel	77.4	70.3	91.2	76.8	68.9	83.4	72.6	61.5	90.4
Power Plant	79.6	80.3	90.4	79.9	144.1	95.5	70.9	60.6	85.7

Table 20 Relative results of the Insertion-based optimization. Left: BVH cost, middle: rendering, user time [s], right: sum of the surface area of inner nodes. The values are relative to the values of the BVHs without running the optimization. The abbreviations of the methods can be seen in table 8.

-	FSW	BIN	INC	FSW	BIN	INC
From	-0.99	0.82	8.64	5.35	7.69	2.65
To	6.79	8.74	55.01	25.75	29.98	16.05
Average	0.87	3.6	24	18.19	22.03	7.84

Table 21 BVH costs, the summary of relative results using the Insertion-based optimization. Left: individual objects, right: architectural scenes.

-	FSW	BIN	INC	FSW	BIN	INC
From	-17.67	-13.73	9.68	-5.63	-44.09	-2.34
To	-5.21	7.11	43.3	38.99	40.30	21.74
Average	-11.24	-3.58	22.49	13.29	11.78	10.57

Table 22 Rendering user times, the summary of relative results using the Insertion-based optimization. Left: individual objects, right: architectural scenes.

-	FSW	BIN	INC	FSW	BIN	INC
From	-1.11	1.2	9.34	7.35	13.24	4.92
To	12.77	17.38	57.48	31.72	40	20.16
Average	1.41	5.39	25.88	24.02	34.21	10.21

Table 23 Sum of the surface area of inner nodes, the summary of relative results using the Insertion-based optimization. Left: individual objects, right: architectural scenes.

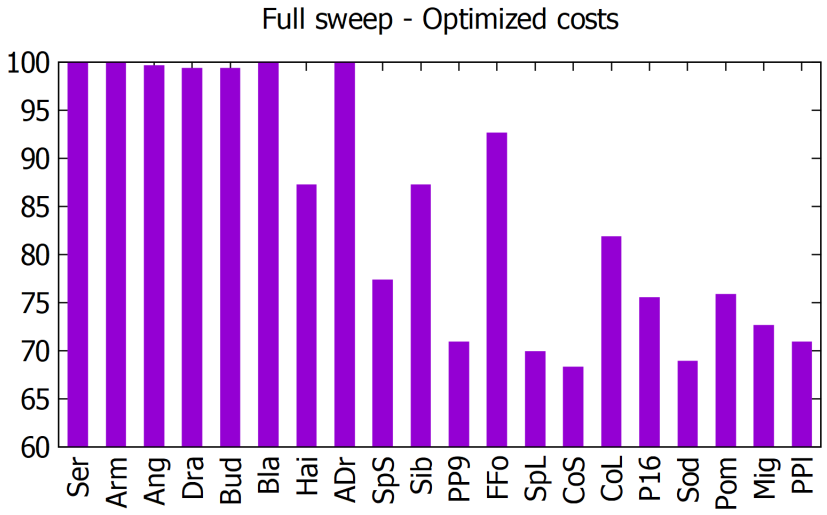


Figure 23 The costs of the full sweep method after optimization relative to the cost before optimization.

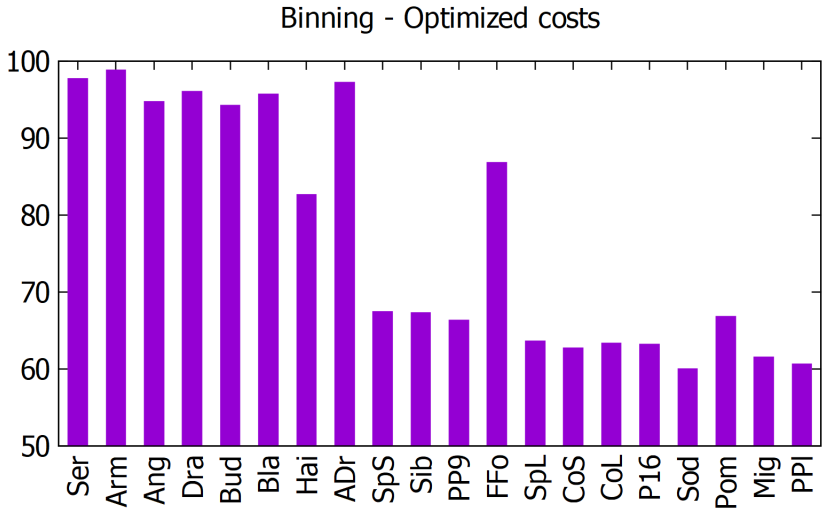


Figure 24 The costs of the binning-based method after optimization relative to the cost before optimization.

optimization with the exception of the Hairball scene (cost reduction of 6,8%), which is similar to the results reported by Bittner et al. [BHH13]. The slight increase in the cost is caused by the oscillation, which happens already in the beginning of the optimization. More particularly, we do not save the best hierarchy (at least at some points of the optimization) but use the final one. Using the stored better hierarchy would resolve the cost increase. In the binning-based method, the results on this type of scenes are similar even though the hierarchy contains more than one triangle per leaf. The cost reduction ranges from 0.8% to 8.7% with an average of 3.6%.

The decrease of the costs of hierarchies built by the incremental algorithm is significant on some scenes (Turbine Blade, Happy Buddha, Dragon and Asian Dragon), but this type of BVH takes also the longest times to optimize (mainly on the Turbine Blade and Asian Dragon scenes). The hierarchies built using the binning are the fastest to optimize on 5 of the 8 individual objects, on the 3 other the fastest are the hierarchies

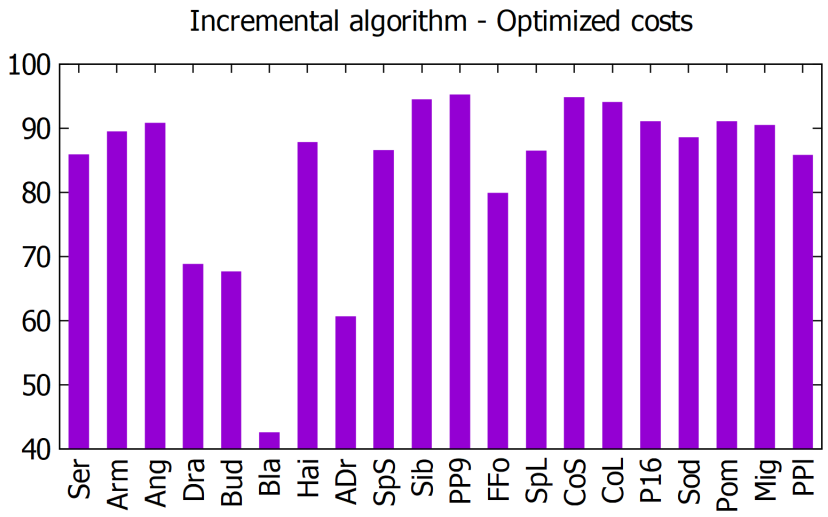


Figure 25 The costs of the incremental algorithm after optimization relative to the cost before optimization.

built using the full sweep. The costs of those hierarchies (built with the binning and then optimized) are also the highest throughout the tested methods with the exception of the Hairball scene.

5.2.2 Architectural scenes

The situation is different on the architectural scenes. First, all tested hierarchies do benefit from the optimization, including the ones constructed using the full sweep. Second, the costs after the optimization are now similar throughout the scenes only for the hierarchies built using the full sweep and the incremental method, while the ones built using binning remain at sometimes even significantly higher costs (Power Plant, Fairy Forest and both Conference scenes). We think that this follows the fact, that the hierarchies constructed with the binning contain more than one triangle per leaf. In their work, Bittner et al. suggested to construct the hierarchies such that exactly one triangle is contained in a leaf, which thus seems to be true. On the other hand, even our tested hierarchies based on the binning do benefit from the optimization, but the final costs are worse than for the other methods (even though the binning benefits from the optimization even more than the other methods).

More particularly, the cost reduction for the full sweep method ranges from 5.3% to 25.7% with an average of 18.2%, for the incremental algorithm from 2.7% to 16% with an average of 7.8% and for the binning from 7.7% to 30% with an average of 22%.

Interestingly, though the cost decrease is smaller in hierarchies built using the incremental method than those using the full sweep method, the time needed to optimize the first ones is higher on 8 of the 12 scenes. The time needed to optimize the hierarchies built using the binning method is the lowest on 9 of the 12 scenes (the exceptions being the Sibenik Cathedral, Fairy Forest and the Soda Hall scenes), but as mentioned, the resulting costs are the highest among the methods.

5.2.3 Summary

To sum the results up, they are again different through the individual objects and the architectural scenes. On the individual objects there is not much point in applying the

Scene	Real Time Rel.	Real Time [s]	User Time [s]
Serapis	52.22	0.047	0.163
Armadillo	52.34	0.190	0.676
Angel	49.31	0.286	1.030
Dragon	48.03	0.427	1.533
H .Buddha	49.13	0.536	1.929
Blade	50.53	0.718	2.563
Hairball	48.40	1.177	4.234
A. Dragon	50.40	3.818	13.825
Sponza S.	53.33	0.032	0.111
Sibenik	54.69	0.035	0.117
P. Plant s. 9	54.22	0.045	0.154
Fairy Forest	49.70	0.083	0.294
Sponza L.	50.41	0.124	0.437
Conference S.	45.64	0.115	0.410
Conference L.	51.10	0.139	0.497
P. Plant s. 16	54.87	0.107	0.368
Soda Hall	49.23	0.892	3.204
Pompeii	48.48	2.568	9.297
San Miguel	47.82	3.877	14.117
Power Plant	50.80	5.722	20.533

Table 24 Results of the parallel binning. Left: Construction real time relative to the sequential version, middle: construction real time [s], right: construction user time [s].

optimization on the full sweep method, since it seems to construct the hierarchies well enough. The quality of the hierarchies built using the binning is somewhat lower due to the quality-speed tradeoff, and the method can benefit from the optimization. The best improvement is achieved in the hierarchies built using the incremental algorithm, which is, however, paid in long optimization times. On the individual scenes, the hierarchies also reach similar costs when the optimization is applied.

The situation is different on the architectural scenes. Though all the methods benefit from the optimization, the higher relative improvements are achieved on the hierarchies built using the full sweep and the binning algorithm. The relative improvement in the hierarchies built using the incremental algorithm is lower, but the costs were already lower compared to the other methods. On this type of scenes, the costs throughout the methods do not reach the similar values, as the quality of the hierarchies built using the binning algorithm remains lower.

5.3 Third phase - Parallel construction

In this section we present the results of our implementation of the parallel version of the binning-based algorithm. This parallel version yields exactly the same costs of hierarchies as the sequential one, because it does not change the construction principle. We have verified that the costs yielded by our implementations are the same. The results can be seen in table 24. Since the only change to the sequential version is the construction time, we present only these results. In the left part of the table, we can see the real times of the construction relative to the sequential version, in the middle part the absolute construction real times and in the right part the absolute construction user times.

We can see that the construction using the parallel implementation lasts approximately half of the time of the sequential one for all the scenes. More specifically, the relative rendering times range from 45,6% to 54,9% of the corresponding times of the sequential version. This means that the speedup is around 2 for all the scenes. We measured the results on a CPU with 2 physical with HyperThreading and comparing

Scene	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
Serapis	100.3	97	99.4	99.7	100.4	113	131	106.5	102	99.1
Armadillo	98.4	100.7	99.7	100.9	100.8	115.5	134.4	107.5	101.7	99.7
Angel	99.7	99.8	100	99.3	99.1	113.5	133.3	107.4	102.8	99.9
Dragon	99.8	99.1	100.8	99.9	100.8	113.8	134.6	106.5	102.2	99.8
H. Buddha	100.1	98.7	98.9	100	100.2	112.1	129	105.6	101.1	99.9
Blade	99.6	99.1	100	100.6	100.2	112.8	132.2	108.7	104.2	100.5
Hairball	100	99.9	100.2	99.9	100.2	111.5	128.8	108.5	104.2	100.8
A. Dragon	99.8	99.6	99.4	100.1	99.8	110.7	129.4	109.9	104.7	100.3
Sponza S.	100.6	100.9	103.3	101.8	102.1	112.6	132.8	106.1	100.6	98.3
Sibenik	99.4	98	101	102.1	100.2	111.5	137.9	109.9	103.5	93.6
P. Plant s. 9	100	99.4	100.9	99.1	98.5	111.4	138.6	112.2	105.5	104.2
Fairy Forest	96.9	97	97.5	95.8	97.3	116	136.6	115	106.6	101.3
Sponza L.	99.8	98.1	98.9	102.4	100.4	113.6	135.6	109.5	103.4	98.8
Conference S.	99.3	98.1	98.5	97.6	98.8	113.6	137.8	113.5	105.4	97.6
Conference L.	99.5	97.9	99.1	98.7	98.9	118	139.7	123.4	114.5	110.6
P. Plant s. 16	99.1	99.3	99.2	99.6	99.9	110.7	143.7	109	102.2	97.4
Soda Hall	100.8	101	100.8	101.8	99	110.4	133	115.8	106.8	99.2
Pompeii	90.4	74.5	93.4	94.6	96.6	117.4	138.2	115.8	108.9	102.8
San Miguel	99.3	99.4	97.6	99.2	101.5	111	132.3	119.8	112.5	104.5
Power Plant	98.8	99.6	98.8	99.9	99.5	110.4	135.3	128.9	115.4	106.1

Table 25 The results of applying the EMC64VAR code to BVH construction. The values are relative to the values when the original Morton code is used (in percents). Left: costs of the BVHs, right: construction, sequential, user times.

to the results of Wald [Wal07] (using 4 threads), we achieve similar speedup for the Fairy Forest and the Conference Small scene. On the Turbine Blade scene (which is larger than the previous two scenes), Wald achieves better speedup than we do.

One of the reasons can lie in the absence of the SIMD instruction usage in our code. The other reason can lie in the parallelization schemes switching (for which Wald in fact does not provide any values). We tuned the switching to achieve the best results across the scenes and tried two approaches - switching based on the number of primitives in the nodes and also on the depth in the hierarchy (which is aimed to generate a number of tasks for the vertical scheme that is a multiple of the number threads). In the end, we ended up with the first approach (which is also the one described by Wald in his work), because it provided more balanced results throughout the scenes and we present these results.

5.4 Fourth phase - Extended Morton codes

In this phase of the measurement we examined the effect of using the Extended Morton codes on the construction of several of our implemented hierarchies. We compared the impacts of the extended codes to the codes the methods use as they were proposed. These methods are the AAC and the PLOC. We examined the the impacts of both of the extended codes proposed in the work of Vinkler et al. [VBH17], EMC64SORT and EMC64VAR. From these two codes, the first (which uses less of the ideas proposed in the work) did not bring us any appreciable results. The more advanced code, EMC64VAR, on the other hand, brought improvements to the hierarchies. We therefore present and further discuss the results of the EMC64VAR code only.

We present the results of using the EMC64VAR code relative to the original methods (using the codes and lengths originally proposed). The new costs of hierarchies and the construction user times can be seen in table 25 and the rendering user times in table 26. We also present the absolute results in the appendix A. These are absolute values of costs, construction and rendering times, but also all the other metrics considered in

Scene	AHQ	ALQ	PL1	PL2	PL3
Serapis	95.2	96.1	98.4	96.1	97
Armadillo	98.8	98.4	90.2	90.3	107.8
Angel	98.3	94.9	99.7	91	100.3
Dragon	92.6	96.7	106.9	96.4	99.8
H. Buddha	97.4	93.8	89.6	98.5	91.8
Blade	89.7	98	94.3	108.9	101
Hairball	104.5	87.8	106.6	98.8	101.7
A. Dragon	101.3	108.3	104.3	97.4	94.5
Sponza S.	101.1	83.3	109.1	94.4	95.2
Sibenik	103.3	96.5	102.9	109	106.2
P. Plant s. 9	100.4	101.4	98.3	100.9	100
Fairy Forest	96.8	100.3	96.2	101.1	103.3
Sponza L.	99.2	110.9	104.7	116.4	94.2
Conference S.	102	96.8	99.4	101.2	99.3
Conference L.	97.8	98.4	104.5	90.8	101.7
P. Plant s. 16	101.6	102.6	98.2	100.3	102.9
Soda Hall	75.1	85.4	106.9	127.6	99.1
Pompeii	92.6	72.9	85.8	80	89.6
San Miguel	102.1	103.2	97.1	101	96.5
Power Plant	99.8	96.7	121.4	112.1	85.7

Table 26 Rendering, user times, when using EMC64VAR code for BVH construction. The values are relative to the values when the original Morton code is used (in percents).

-	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
From	-0.27	-0.65	-0.84	-0.87	-0.83	0.79	-0.95	-3.34	-2.37	-2.09
To	1.59	2.97	1.08	0.69	0.91	9.59	25.49	6.56	5.39	3.41
Average	0.3	0.78	0.19	-0.05	-0.2	1.34	3.06	0.92	0.62	0.61

Table 27 BVH costs, the summary of relative results using the EMC64VAR code. Left: individual objects, right: architectural scenes.

-	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
From	-15.53	-34.57	-9.94	-4.68	-0.81	-18.00	-43.7	-28.88	-15.41	-10.60
To	-10.71	-28.83	-5.62	-1.10	0.90	-10.39	-32.34	-6.10	-0.59	6.39
Average	-12.86	-31.60	-7.61	-2.85	0.00	-13.05	-36.80	-14.90	-7.12	-1.19

Table 28 Construction user times, the summary of relative results using the EMC64VAR code. Left: individual objects, right: architectural scenes.

-	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
From	-4.55	-8.27	-6.91	-8.86	-7.79	3.34	-10.86	-21.42	-27.59	-6.16
To	10.31	12.23	10.43	9.74	8.17	24.94	27.14	14.21	20.03	14.32
Average	2.77	3.26	1.26	2.85	0.76	2.34	4.31	-2.06	-2.89	2.20

Table 29 Rendering user times, the summary of relative results using the EMC64VAR code. Left: individual objects, right: architectural scenes.

this thesis. In this section we also present the summary of results divided between the individual objects and architectural scenes. The summary for BVH costs can be seen in table 27, for construction real times in table 28 and for rendering real times in table 29.

From our results we can see small improvement in the costs of the hierarchies for the AAC variants (14 out of 20 scenes for the AAC-HQ and 17 out of 20 for the AAC-LQ). The costs also improve for the PLOC algorithm (more specifically on 12 for PL1, on 12 for PL2 and on 10 for PL3). The costs improve for all algorithm variants for the Fairy Forest and the Pompeii scenes. This is because these scenes span mainly in two dimensions and the variable axis order of the EMC64VAR code accounts for that. For the AAC-LQ variant, which had the cost higher for the Pompeii, the cost is now similar

to the AAC-HQ variant.

While the construction times increased in our implementation (which could be, however, further optimized), interesting are the results of the rendering times. While on some scenes we observe an increase in the rendering time, on others the times are notably lowered. The rendering times are lower than when using the original codes on 12 scenes for the AAC-HQ, on 14 scenes for the AAC-LQ, on 12 for the PL1, on 10 for the PL2 and on 11 for the PL3. Significant reduction an be seen for the Pompeii scene (mainly for the AAC-LQ, PL2 and PL1), again, and somewhat lower for the Happy Buddha and Serapis scenes. For the AAC methods, there is also notable decrease of costs on the Soda Hall scene and on the Armadillo scene for the PL1 and PL2 PLOC variants. On the other scenes, there is a decrease for some of the methods, e.g. for the individual objects for the AAC methods.

6 Conclusion

In this thesis we examined the algorithms for the construction of an acceleration data structure, the bounding volume hierarchy, for the use in the ray tracing method. In our work we mainly focus on the algorithms designed for a CPU rather than a GPU. We presented the introduction to the topic, where we described the main attributes of the BVHs, the motivation of using them and their differences to k-d trees. After that we presented a categorization of the methods and described each of the principles upon which the respective category is based. We also recalled some of the state of the art algorithms and included them in the respective categories.

In the next part of our thesis, we selected 6 BVH algorithms (5 construction ones and 1 optimization algorithm), which we saw as promising in the context of the CPU methods. We reimplemented these algorithms in the nanoGOLEM framework using the C++ programming language. We focused on the design to be meaningful. We also focused on the modern traits of the C++ language and the coding style. We also added the description and the reimplementation of the Extended Morton codes. The hierarchies constructed by any of the methods can be subsequently optimized using the Insertion-based optimization method.

We evaluated the implemented methods on a set of 20 scenes, from which at least 14 scenes are publicly available and often used when evaluating the acceleration structures for the ray tracing. For the evaluation we used the primary, shadow and randomly generated rays. We present the results of our implementation by reporting not only construction and rendering times and the costs of the hierarchies, but also other metrics, which are independent on the optimization and generally more suitable for the comparison with other algorithms. Because of that, they also allow to verify the other implementations of these methods. We summed the achieved results and compared the convenience of using the methods while distinguishing between the scenes containing individual objects and the architectural scenes. Based on our results, we also add suggestions about the examined methods.

Apart from that, we also implemented one of the methods to run in parallel using the threads from the C++11 standard. Initially, we expected that we would implement all the methods in such manner or even using the SIMD instructions. In the end, we were unfortunately unable to achieve that. This is mainly because we spent much larger amount of time on verifying our implemented methods than we initially expected. It was easy to implement the methods to correctly construct the hierarchies that allow for correctly rendered images, while not running unsufficiently long. However, it was harder to tune the implementation to yield the costs of the hierarchies based on the surface area heuristic same to the results reported in the respective works. This was mainly because of the small details in the algorithms, which, however, had impact on the costs if implemented different from the original methods. To present valid results, the verification was, however, more important than the optimization.

Nevertheless, the results of our thesis can be used for comparing the algorithms with others, future verification of other implementations of the same algorithms and also to compare the implemented algorithms among themselves.

References

[AKL13] Timo Aila, Tero Karras, and Samuli Laine. "On Quality Metrics of Bounding Volume Hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 101–107. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492056. URL: http://doi.acm.org/10.1145/2492045.2492056.

[BHH13] Jiri Bittner, Michal Hapala, and Vlastimil Havran. "Fast Insertion-Based Optimization of Bounding Volume Hierarchies". In: vol. 32. 1. Blackwell Publishing Ltd, 2013, pp. 85–100. DOI: 10.1111/cgf.12000. URL: http://dx.doi.org/10.1111/cgf.12000.

[BHH15] Jiri Bittner, Michal Hapala, and Vlastimil Havran. "Incremental BVH Construction for Ray Tracing". In: *Computers and Graphics* 47.0 (2015), pp. 135–144. ISSN: 0097-8493. DOI: http://dx.doi.org/10.1016/j.cag.2014.12.001. URL: http://www.sciencedirect.com/science/article/pii/S0097849314001435.

[DP15] Leonardo R Domingues and Helio Pedrini. "Bounding volume hierarchy optimization through agglomerative treelet restructuring". In: *Proceedings of the 7th Conference on High-Performance Graphics*. ACM. 2015, pp. 13–20.

[EG07] M. Ernst and G. Greiner. "Early Split Clipping for Bounding Volume Hierarchies". In: *2007 IEEE Symposium on Interactive Ray Tracing*. Sept. 2007, pp. 73–78. DOI: 10.1109/RT.2007.4342593.

[Fue+16] V. Fuetterling et al. "Parallel Spatial Splits in Bounding Volume Hierarchies". In: *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*. EGPGV '16. Groningen, The Netherlands: Eurographics Association, 2016, pp. 21–30. ISBN: 978-3-03868-006-2. DOI: 10.2312/pgv.20161179. URL: https://doi.org/10.2312/pgv.20161179.

[Gan+15] P Ganestam et al. "Bonsai: rapid bounding volume hierarchy generation using mini trees". In: *Journal of Computer Graphics Techniques (JCGT)* 4.3 (2015).

[GPM11] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. "Simpler and faster HLBVH with work queues". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM. 2011, pp. 59–64.

[GS87] Jeffrey Goldsmith and John Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". In: *Computer Graphics and Applications, IEEE* 7 (June 1987), pp. 14–20. DOI: 10.1109/MCG.1987.276983.

[Gu+13] Yan Gu et al. "Efficient BVH construction via approximate agglomerative clustering". In: *Proceedings of the 5th High-Performance Graphics Conference*. ACM. 2013, pp. 81–88.

[Hav00] Vlastimil Havran. "Heuristic Ray Shooting Algorithms". Ph.D. Thesis. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Nov. 2000. URL: http://www.cgg.cvut.cz/~havran/phdthesis.html.

[HB02] Vlastimil Havran and Jiri Bittner. "On Improving KD-Trees for Ray Shooting". In: *Journal of WSCG* 10.1 (Feb. 2002), pp. 209–216.

[HPP00] Vlastimil Havran, Jan Prikryl, and Werner Purgathofer. *Statistical Comparison of Ray-Shooting Efficiency Schemes*. Tech. rep. TR-186-2-00-14. human contact: technical-report@cg.tuwien.ac.at. Favoritenstrasse 9/186, A-1040 Vienna, Austria: Institute of Computer Graphics, Vienna University of Technology, May 2000, pp. 1–13.

[HMB17] Jakub Hendrich, Daniel Meister, and Jiri Bittner. "Parallel BVH Construction using Progressive Hierarchical Refinement". In: *Computer Graphics Forum*. Vol. 36. 2. Wiley Online Library. 2017, pp. 487–494.

[Kar12] Tero Karras. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees". In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association. 2012, pp. 33–37.

[KA13] Tero Karras and Timo Aila. "Fast Parallel Construction of High-quality Bounding Volume Hierarchies". In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 89–99. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492055. URL: http://doi.acm.org/10.1145/2492045.2492055.

[KK86] Timothy L. Kay and James T. Kajiya. "Ray Tracing Complex Scenes". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 269–278. ISBN: 0-89791-196-2. DOI: 10.1145/15922.15916. URL: http://doi.acm.org/10.1145/15922.15916.

[Ken08] Andrew Kensler. "Tree Rotations for Improving Bounding Volume Hierarchies". In: *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*. Aug. 2008, pp. 73–76.

[Kop+12] Daniel Kopta et al. "Fast, Effective BVH Updates for Animated Scenes". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. Costa Mesa, California: ACM, 2012, pp. 197–204. ISBN: 978-1-4503-1194-6. DOI: 10.1145/2159616.2159649. URL: http://doi.acm.org/10.1145/2159616.2159649.

[Lau+09] Christian Lauterbach et al. "Fast BVH construction on GPUs". In: *Computer Graphics Forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 375–384.

[MB90] David J. MacDonald and Kellogg S. Booth. "Heuristics for Ray Tracing Using Space Subdivision". In: *Vis. Comput.* 6.3 (May 1990), pp. 153–166. ISSN: 0178-2789. DOI: 10.1007/BF01911006. URL: http://dx.doi.org/10.1007/BF01911006.

[McG17] Morgan McGuire. *Computer Graphics Archive*. https://casual-effects.com/data. July 2017. URL: https://casual-effects.com/data.

[MB18] Daniel Meister and Jiri Bittner. "Parallel locally-ordered clustering for bounding volume hierarchy construction". In: *IEEE transactions on visualization and computer graphics* 24.3 (2018), pp. 1345–1353.

[18] *Nvidia Turing GPU architecture*. Tech. rep. Nvidia Corporation, Sept. 2018.

[PL10] 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*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 87–95. URL: http://dl.acm.org/citation.cfm?id=1921479.1921493.

[Shu+12] Julian Shun et al. "Brief announcement: the problem based benchmark suite". In: *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2012, pp. 68–70.

[SFD09] Martin Stich, Heiko Friedrich, and Andreas Dietrich. "Spatial splits in bounding volume hierarchies". In: *Proceedings of the Conference on High Performance Graphics 2009*. ACM. 2009, pp. 7–13.

[TM] Greg Turk and Brendan Mullins. *Large Geometric Models Archive*. https://www.cc.gatech.edu/projects/large_models/.

[Uni] Leland Stanford Junior University. *The Stanford 3D Scanning Repository*. http://graphics.stanford.edu/data/3Dscanrep/.

[VBH17] Marek Vinkler, Jiri Bittner, and Vlastimil Havran. "Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction". In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM, 2017, 9:1–9:8. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105782. URL: http://doi.acm.org/10.1145/3105762.3105782.

[Wal07] Ingo Wald. "On fast construction of SAH-based bounding volume hierarchies". In: *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*. IEEE. 2007, pp. 33–40.

[WBB08] Ingo Wald, Carsten Benthin, and Solomon Boulos. "Getting rid of packets-efficient simd single-ray traversal using multi-branching bvhs". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 49–57.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. "Ray tracing deformable scenes using dynamic bounding volume hierarchies". In: *ACM Transactions on Graphics (TOG)* 26.1 (2007), p. 6.

[WH06] Ingo Wald and Vlastimil Havran. "On building fast kd-trees for ray tracing, and on doing that in O(N log N)". In: *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*. Sept. 2006, pp. 61–69.

[Wal+08] Bruce Walter et al. "Fast agglomerative clustering for rendering". In: *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*. IEEE. 2008, pp. 81–86.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. "Improved Computational Methods for Ray Tracing". In: *ACM Trans. Graph.* 3.1 (Jan. 1984), pp. 52–69. ISSN: 0730-0301. DOI: 10.1145/357332.357335. URL: http://doi.acm.org/10.1145/357332.357335.

[Whi80] Turner Whitted. “An Improved Illumination Model for Shaded Display”. In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: http://doi.acm.org/10.1145/358876.358882.

Appendix A

Other results

In this appendix we present additional results of our measurement.

A.1 First phase - primary and shadow rays, random rays

These are the results rendering the scenes using other than primary rays, in particular primary rays added by shadow rays and random rays generated using the algorithm proposed in the work of Havran et al. [HPP00]. For using the combination primary and shadow rays, the rendering times are presented in tables 30 (real) and 31 (user), the average number of traversal steps per ray in table 32 and of the intersection tests per ray in table 33. For the rays generated using the random scheme, the rendering times are presented in tables 34 (real) and 35 (user), the average number of traversal steps per ray in table 36 and of the intersection tests per ray in table 37.

A.2 Second phase - Insertion-based optimization, absolute values

In this section we present the absolute values of the results achieved in the second phase of our measurement. The costs of the hierarchies after the optimization together with the real and user times of the optimization procedure are presented in table 38, the sum of the surface area of inner nodes divided by the surface area of the hierarchy together with the sum correponding to the leaves and the average number of primitive

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.521	0.544	0.707	0.663	0.541	0.542	0.538	0.696	0.670	0.669	0.678
Armadillo	0.341	0.365	0.478	0.447	0.370	0.361	0.360	0.477	0.464	0.485	0.462
Angel	0.395	0.410	0.505	0.499	0.410	0.413	0.413	0.551	0.480	0.516	0.510
Dragon	0.305	0.319	0.483	0.434	0.320	0.322	0.323	0.515	0.402	0.420	0.429
H. Buddha	0.240	0.251	0.348	0.331	0.251	0.257	0.257	0.423	0.346	0.324	0.362
Blade	0.358	0.392	0.686	0.539	0.395	0.384	0.383	0.836	0.518	0.527	0.588
Hairball	2.307	2.446	3.371	3.616	2.491	2.437	2.503	3.395	2.880	2.917	2.979
A. Dragon	0.546	0.594	0.801	0.719	0.567	0.579	0.588	1.047	0.748	0.769	0.834
Sponza S.	1.680	2.053	1.441	1.686	1.602	1.520	1.725	1.468	1.409	1.501	1.390
Sibenik	1.366	1.744	1.545	1.522	1.966	1.893	1.754	1.456	1.491	1.439	1.451
P. Plant s. 9	0.293	0.459	0.265	0.260	0.356	0.294	0.286	0.254	0.276	0.273	0.271
Fairy Forest	0.761	1.494	0.797	0.801	0.871	0.799	0.765	0.970	0.834	0.786	0.815
Sponza L.	2.652	3.027	2.881	2.763	2.845	2.377	2.459	2.317	2.741	2.581	2.658
Conference S.	1.066	1.504	0.963	0.957	1.137	1.000	0.983	0.975	0.961	0.984	0.914
Conference L.	1.032	1.508	0.936	0.942	1.247	1.075	1.038	0.974	0.950	1.012	0.984
P. Plant s. 16	2.052	3.502	2.408	2.358	2.637	1.993	2.126	2.153	2.306	2.247	2.320
Soda Hall	2.266	2.371	2.540	2.529	2.376	2.059	2.242	1.931	2.251	2.155	2.420
Pompeii	1.757	2.000	2.063	2.658	2.024	1.881	1.891	1.897	2.211	2.296	2.067
San Miguel	2.960	3.438	2.911	2.926	3.527	3.264	3.109	2.881	3.163	3.022	3.030
Power Plant	3.098	5.715	4.028	3.817	3.986	3.249	3.249	2.779	4.110	3.346	3.896

Table 30 Rendering, real time, primary and shadow rays [s].

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.521	0.542	0.707	0.662	0.540	0.542	0.536	0.695	0.668	0.669	0.678
Armadillo	0.341	0.365	0.477	0.447	0.370	0.361	0.360	0.477	0.463	0.485	0.462
Angel	0.394	0.409	0.505	0.499	0.409	0.412	0.413	0.551	0.480	0.515	0.510
Dragon	0.305	0.319	0.482	0.434	0.319	0.321	0.322	0.515	0.402	0.420	0.428
H. Buddha	0.240	0.249	0.348	0.330	0.249	0.256	0.257	0.424	0.346	0.324	0.362
Blade	0.359	0.391	0.685	0.539	0.394	0.384	0.382	0.835	0.518	0.525	0.587
Hairball	2.307	2.444	3.370	3.615	2.491	2.436	2.503	3.393	2.880	2.915	2.979
A. Dragon	0.546	0.594	0.802	0.719	0.567	0.579	0.586	1.047	0.748	0.769	0.834
Sponza S.	1.679	2.051	1.440	1.685	1.602	1.517	1.725	1.467	1.408	1.501	1.389
Sibenik	1.366	1.742	1.544	1.522	1.964	1.892	1.753	1.455	1.490	1.439	1.449
P. Plant s. 9	0.293	0.458	0.264	0.260	0.355	0.293	0.284	0.253	0.276	0.273	0.271
Fairy Forest	0.760	1.492	0.797	0.800	0.870	0.798	0.765	0.970	0.834	0.786	0.813
Sponza L.	2.652	3.027	2.880	2.762	2.844	2.377	2.457	2.317	2.740	2.580	2.657
Conference S.	1.066	1.502	0.962	0.956	1.135	0.999	0.982	0.974	0.960	0.984	0.914
Conference L.	1.031	1.508	0.936	0.942	1.246	1.075	1.038	0.973	0.950	1.012	0.984
P. Plant s. 16	2.051	3.501	2.408	2.358	2.634	1.992	2.126	2.152	2.306	2.247	2.320
Soda Hall	2.266	2.370	2.540	2.529	2.374	2.059	2.241	1.930	2.249	2.155	2.419
Pompeii	1.757	1.999	2.063	2.657	2.024	1.880	1.890	1.895	2.210	2.295	2.066
San Miguel	2.959	3.436	2.910	2.925	3.524	3.263	3.109	2.879	3.162	3.022	3.029
Power Plant	3.097	5.714	4.028	3.816	3.983	3.249	3.248	2.778	4.109	3.345	3.896

Table 31 Rendering, user time, using the primary and shadow rays [s].

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	34.1	35.5	48.5	44.6	36.0	35.4	34.9	42.6	44.5	45.0	45.6
Armadillo	22.9	24.0	32.2	29.9	24.9	23.8	23.8	27.4	29.7	31.2	30.4
Angel	22.4	23.0	28.3	27.6	23.5	23.0	22.9	25.5	25.2	27.6	27.9
Dragon	17.1	18.0	27.5	25.1	18.4	17.9	17.8	27.0	21.6	22.9	23.4
H. Buddha	12.0	12.5	17.6	16.8	12.7	12.5	12.4	19.5	16.5	15.3	17.3
Blade	22.7	23.3	41.0	33.0	24.6	23.9	23.6	49.5	30.3	31.3	34.7
Hairball	81.2	84.5	114.4	115.6	88.0	81.2	81.7	104.6	90.6	95.5	99.1
A. Dragon	28.1	29.5	38.8	35.8	29.3	28.9	28.9	45.0	34.2	35.3	37.9
Sponza S.	113.1	138.2	110.4	129.8	130.9	125.6	115.7	105.1	108.8	115.8	114.3
Sibenik	122.1	128.2	137.5	128.1	168.0	165.2	152.7	123.9	124.6	121.6	119.8
P. Plant s. 9	19.2	22.2	14.9	15.2	23.1	17.3	17.2	13.6	15.5	15.5	14.9
Fairy Forest	60.9	63.4	65.5	62.0	69.7	64.7	61.2	76.1	65.6	61.1	64.4
Sponza L.	207.8	266.6	208.2	229.6	250.8	196.9	208.4	167.0	212.5	205.3	191.7
Conference S.	74.1	84.0	61.7	59.2	82.8	72.0	72.5	57.8	59.3	62.9	55.8
Conference L.	77.7	101.7	65.4	66.0	99.1	76.7	75.9	63.5	63.4	68.3	65.8
P. Plant s. 16	168.1	191.8	198.1	196.5	218.0	155.7	170.8	158.0	191.3	189.9	184.5
Soda Hall	164.0	189.5	242.6	236.7	234.2	191.0	209.7	168.5	208.1	201.4	222.0
Pompeii	105.9	116.2	115.0	160.9	125.3	108.9	107.4	95.2	121.4	123.6	108.8
San Miguel	254.5	265.3	219.3	218.1	299.5	266.2	258.4	182.1	235.5	222.4	217.2
Power Plant	296.7	294.3	342.6	327.9	390.3	316.1	309.6	219.4	356.6	301.0	334.6

Table 32 Number of traversal steps per single ray while using primary and shadow rays.

references per leaf are presented in table 39. The average number of intersection tests and traversal steps per ray using the optimized hierarchies can be seen in table 40. Table 41 shows the rendering real and user times achieved when using the optimized hierarchies. All these results are achieved using the primary rays only.

A.3 Fourth phase - Extended Morton codes, absolute values

We present the results of using the EMC64VAR extended Morton code in BVH construction in this section. In the table 42 we can see the user times of the construction using the code as well as the average number of primitive references per leaf. In the table 43 real and user times of the rendering using the BVHs built using the EMC64VAR code are shown. The average numbers of traversal steps and intersection tests per ray can be seen in table 44 and the sums of the surface areas of inner nodes and the leaves

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	5.7	6.1	6.9	6.8	5.8	5.8	5.8	7.0	6.3	6.1	6.1
Armadillo	2.1	2.7	2.8	2.9	2.2	2.1	2.1	2.6	2.7	2.8	2.4
Angel	3.1	3.3	3.6	4.0	3.2	3.1	3.1	3.6	3.4	3.5	3.3
Dragon	2.5	2.8	3.7	3.6	2.6	2.5	2.5	4.2	2.9	3.0	3.0
H. Buddha	1.8	2.1	2.3	2.4	1.9	1.9	1.9	3.2	2.4	2.1	2.4
Blade	2.0	3.8	3.4	2.9	2.1	2.0	2.0	6.9	2.6	2.5	2.7
Hairball	62.3	66.8	73.1	83.9	64.4	60.3	62.4	84.7	64.7	64.6	65.3
A. Dragon	2.8	6.2	3.4	3.4	2.9	2.8	2.8	4.8	3.2	3.1	3.2
Sponza S.	54.1	65.3	28.5	33.4	24.9	23.4	51.2	29.7	25.4	32.0	21.7
Sibenik	19.6	48.7	21.4	24.4	31.2	31.5	28.3	18.0	24.5	20.1	23.0
P. Plant s. 9	8.6	25.9	9.2	8.4	10.8	8.9	9.1	8.7	9.5	9.4	9.6
Fairy Forest	11.6	94.0	11.3	11.6	11.9	11.7	11.8	16.0	11.4	11.3	11.4
Sponza L.	57.3	40.0	70.3	40.4	31.1	35.4	38.8	42.9	41.9	40.2	57.0
Conference S.	26.6	71.8	27.9	28.0	25.1	23.6	23.2	31.6	27.0	27.7	27.0
Conference L.	23.8	52.5	24.7	24.8	23.5	27.0	25.6	27.2	26.0	27.4	26.1
P. Plant s. 16	47.6	174.3	47.7	47.7	55.6	50.3	50.7	54.1	45.9	45.2	49.3
Soda Hall	74.5	68.0	31.0	35.2	31.6	38.7	39.4	30.7	33.8	31.0	33.2
Pompeii	53.9	67.1	58.5	65.4	55.9	59.6	59.0	68.8	67.7	69.2	65.7
San Miguel	25.5	56.1	32.7	34.9	32.6	34.1	30.8	44.4	35.6	37.6	39.7
Power Plant	24.1	310.2	42.7	60.2	23.6	22.9	22.2	42.9	40.3	29.8	43.9

Table 33 Number of intersection tests per single ray while using primary and shadow rays.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.618	0.660	0.684	0.645	0.619	0.621	0.598	0.813	0.683	0.705	0.673
Armadillo	0.059	0.056	0.059	0.082	0.056	0.056	0.057	0.058	0.076	0.055	0.061
Angel	0.059	0.051	0.061	0.059	0.053	0.055	0.056	0.065	0.066	0.057	0.059
Dragon	0.520	0.487	0.715	0.736	0.454	0.461	0.488	1.042	0.588	0.672	0.638
H. Buddha	0.035	0.032	0.040	0.040	0.033	0.033	0.033	0.049	0.037	0.044	0.040
Blade	0.055	0.057	0.087	0.076	0.056	0.056	0.057	0.120	0.076	0.082	0.082
Hairball	2.279	2.205	3.457	2.942	2.350	2.233	2.385	3.053	2.819	2.694	3.277
A. Dragon	0.523	0.548	0.642	0.615	0.560	0.558	0.557	0.868	0.654	0.642	0.697
Sponza S.	0.724	1.047	0.735	0.852	0.736	0.813	0.870	0.650	0.854	0.851	0.852
Sibenik	0.518	0.694	0.573	0.656	0.664	0.685	0.695	0.522	0.504	0.523	0.496
P. Plant s. 9	0.223	0.819	0.248	0.239	0.276	0.214	0.213	0.217	0.233	0.259	0.292
Fairy Forest	0.660	0.598	0.605	0.622	0.542	0.531	0.530	0.608	0.635	0.617	0.581
Sponza L.	1.157	1.075	1.158	1.128	1.599	1.270	1.227	1.017	1.097	1.018	0.987
Conference S.	0.729	1.030	0.681	0.707	0.862	0.785	0.752	0.615	0.772	0.717	0.671
Conference L.	0.107	0.140	0.097	0.101	0.125	0.111	0.101	0.092	0.107	0.103	0.098
P. Plant s. 16	0.196	0.291	0.260	0.230	0.246	0.207	0.204	0.200	0.220	0.242	0.211
Soda Hall	0.638	0.802	0.685	0.640	0.647	0.636	0.596	0.773	0.790	0.633	0.792
Pompeii	1.078	1.285	1.185	1.279	1.179	1.124	1.195	0.997	0.950	1.049	1.015
San Miguel	1.335	1.508	1.511	1.760	1.615	1.587	1.501	1.442	1.632	1.764	1.535
Power Plant	0.778	1.327	0.810	0.831	1.031	0.834	0.814	0.713	0.919	0.884	0.932

Table 34 Rendering, real times, using the random rays [s].

in the same spirit as in the previous sections can be seen in table 45.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	0.617	0.659	0.684	0.644	0.619	0.621	0.598	0.813	0.682	0.704	0.672
Armadillo	0.058	0.056	0.059	0.082	0.056	0.056	0.057	0.058	0.076	0.055	0.061
Angel	0.059	0.050	0.061	0.058	0.053	0.055	0.056	0.065	0.066	0.057	0.059
Dragon	0.519	0.487	0.715	0.736	0.453	0.461	0.488	1.042	0.588	0.672	0.638
H. Buddha	0.034	0.031	0.039	0.039	0.032	0.033	0.033	0.047	0.038	0.044	0.040
Blade	0.056	0.057	0.087	0.076	0.056	0.056	0.057	0.121	0.076	0.082	0.081
Hairball	2.279	2.204	3.457	2.941	2.350	2.233	2.384	3.053	2.819	2.694	3.277
A. Dragon	0.522	0.548	0.641	0.615	0.560	0.558	0.556	0.868	0.654	0.642	0.697
Sponza S.	0.724	1.046	0.735	0.852	0.736	0.813	0.870	0.650	0.853	0.851	0.852
Sibenik	0.518	0.694	0.573	0.656	0.664	0.685	0.695	0.522	0.504	0.523	0.496
P. Plant s. 9	0.223	0.820	0.248	0.238	0.276	0.214	0.214	0.217	0.233	0.259	0.292
Fairy Forest	0.659	0.597	0.606	0.622	0.542	0.531	0.530	0.608	0.635	0.617	0.581
Sponza L.	1.157	1.075	1.158	1.127	1.599	1.270	1.227	1.017	1.097	1.017	0.986
Conference S.	0.729	1.030	0.681	0.707	0.862	0.785	0.752	0.615	0.772	0.717	0.672
Conference L.	0.107	0.140	0.097	0.101	0.125	0.111	0.101	0.092	0.107	0.103	0.098
P. Plant s. 16	0.196	0.291	0.260	0.230	0.246	0.206	0.204	0.199	0.220	0.242	0.211
Soda Hall	0.638	0.802	0.685	0.640	0.647	0.635	0.596	0.772	0.790	0.633	0.792
Pompeii	1.078	1.285	1.185	1.278	1.178	1.123	1.195	0.997	0.950	1.049	1.015
San Miguel	1.335	1.508	1.511	1.760	1.614	1.586	1.501	1.441	1.632	1.763	1.535
Power Plant	0.777	1.327	0.810	0.830	1.031	0.833	0.814	0.713	0.918	0.883	0.931

Table 35 Rendering, user time, using the random rays [s].

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	48.5	52.2	57.1	50.4	51.3	50.4	48.2	60.8	54.1	54.7	53.6
Armadillo	4.0	3.9	3.8	7.2	4.0	3.8	4.2	3.2	6.7	3.6	4.6
Angel	4.0	3.4	4.1	3.9	3.7	3.9	4.0	4.0	5.2	3.6	4.0
Dragon	41.4	39.0	61.0	59.7	37.3	37.9	40.1	78.7	47.9	55.5	52.3
H. Buddha	0.8	0.7	1.0	1.0	0.8	0.8	0.8	1.3	0.9	1.2	1.0
Blade	2.5	2.3	4.0	3.6	2.6	2.6	2.6	5.6	3.7	3.9	4.0
Hairball	107.9	114.4	180.1	148.7	115.6	107.0	112.7	144.2	135.3	136.2	155.0
A. Dragon	40.5	43.1	51.5	50.0	44.4	42.5	43.7	59.7	50.3	47.9	52.4
Sponza S.	61.0	70.6	68.9	72.6	62.1	69.0	70.9	47.6	71.4	72.2	70.4
Sibenik	43.5	50.3	47.7	51.6	57.9	58.0	58.6	44.3	41.3	44.7	42.5
P. Plant s. 9	19.1	33.8	23.5	20.3	25.4	18.4	17.7	18.5	21.1	20.5	21.9
Fairy Forest	56.6	46.1	46.3	50.3	43.9	41.6	43.8	46.1	50.1	50.6	45.6
Sponza L.	89.6	87.9	81.4	87.7	139.3	102.9	89.0	70.6	79.7	79.6	75.5
Conference S.	57.1	69.9	48.0	45.2	74.4	58.9	59.5	38.5	50.9	46.6	46.2
Conference L.	6.1	7.7	5.2	5.7	8.4	6.3	6.2	5.2	5.7	5.1	5.0
P. Plant s. 16	17.4	23.5	24.3	21.0	22.2	18.6	18.0	16.3	19.1	21.8	18.2
Soda Hall	58.2	45.3	57.6	52.6	57.1	55.3	50.9	53.4	57.8	49.7	60.1
Pompeii	88.8	99.8	92.6	104.9	103.4	92.5	91.8	75.0	74.1	79.2	81.2
San Miguel	115.4	126.5	118.7	131.3	139.4	134.3	128.5	97.3	125.9	133.1	118.3
Power Plant	74.5	94.8	74.0	73.6	101.5	80.3	76.9	58.9	81.0	83.6	85.4

Table 36 Numbers of traversal steps per single ray when using the random rays.

Scene	FSW	BIN	AHQ	ALQ	BP0	BP1	BP2	INC	PL1	PL2	PL3
Serapis	11.6	14.2	11.5	13.0	12.5	12.5	11.9	14.6	12.1	11.7	11.0
Armadillo	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Angel	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.0
Dragon	6.7	7.2	8.2	10.6	6.4	6.5	6.7	12.4	6.9	7.3	7.1
H. Buddha	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.1	0.1	0.1
Blade	0.3	0.6	0.4	0.4	0.3	0.3	0.3	0.6	0.3	0.4	0.3
Hairball	114.8	106.5	134.7	125.1	114.3	113.1	118.8	138.5	119.4	109.7	133.8
A. Dragon	5.1	5.1	4.8	5.0	5.2	5.2	5.2	8.2	5.0	5.1	5.2
Sponza S.	17.4	44.0	10.9	18.5	17.6	19.1	22.3	17.3	18.4	17.6	20.9
Sibenik	12.2	22.3	12.4	15.6	14.2	15.3	14.2	8.1	10.6	8.8	9.1
P. Plant s. 9	4.6	66.3	4.0	4.0	5.4	3.8	4.8	4.0	4.0	6.1	9.3
Fairy Forest	11.0	16.7	11.9	12.6	11.5	11.5	10.0	13.3	11.9	11.2	11.6
Sponza L.	34.0	27.0	33.7	26.7	32.0	27.8	34.7	28.5	27.5	23.4	23.9
Conference S.	20.4	38.9	21.3	25.1	19.3	21.7	19.8	21.8	24.9	24.3	21.5
Conference L.	3.2	5.4	2.0	2.1	3.0	3.2	2.2	1.9	2.4	2.6	2.4
P. Plant s. 16	2.1	6.6	2.1	2.1	2.7	2.1	2.3	2.3	2.9	2.2	2.1
Soda Hall	11.0	40.9	12.8	13.2	11.0	10.6	11.6	19.2	18.5	14.1	17.9
Pompeii	22.3	35.2	21.1	19.8	19.7	20.5	30.8	17.9	18.2	20.1	18.1
San Miguel	19.8	26.3	25.0	32.9	26.5	26.7	24.0	31.0	25.8	26.1	26.5
Power Plant	7.9	43.5	8.7	11.2	8.5	8.7	8.3	8.1	9.1	8.6	9.3

Table 37 Numbers of intersection tests per single ray when using the random rays.

Scene	FSW	BIN	INC	FSW	BIN	INC	FSW	BIN	INC
Serapis	130.1	133.5	131.2	0.521	0.682	3.716	0.520	0.681	3.716
Armadillo	81.9	84.3	84.0	2.233	2.072	10.421	2.224	2.069	10.412
Angel	78.4	79.5	79.3	10.796	10.511	20.996	10.786	10.508	20.985
Dragon	136.7	140.4	139.7	35.258	13.496	45.582	35.237	13.484	45.562
H. Buddha	155.1	158.5	157.9	31.372	18.388	51.994	31.361	18.371	51.969
Blade	179.6	192.3	189.2	12.446	22.407	200.830	12.426	22.386	200.806
Hairball	985.6	993.1	996.5	221.301	46.862	235.713	221.241	46.827	235.661
A. Dragon	91.4	94.0	93.4	38.190	73.355	562.322	38.067	73.220	562.145
Sponza S.	159.6	167.9	160.4	1.518	0.932	2.018	1.517	0.930	2.015
Sibenik	62.1	65.7	61.1	1.293	1.365	2.201	1.292	1.363	2.200
P. Plant s. 9	32.6	35.4	33.0	2.361	0.434	2.033	2.360	0.434	2.028
Fairy Forest	76.1	93.6	77.4	1.016	1.187	3.175	1.015	1.186	3.174
Sponza L.	154.1	161.2	152.4	8.957	1.230	5.789	8.953	1.227	5.783
Conference S.	83.9	112.9	84.0	7.361	1.575	4.572	7.360	1.570	4.565
Conference L.	79.9	107.2	80.0	4.002	2.032	7.341	3.999	2.031	7.330
P. Plant s. 16	60.6	75.6	61.3	12.600	1.992	7.608	12.592	1.990	7.603
Soda Hall	140.4	156.0	140.0	28.819	28.950	32.612	28.783	28.923	32.569
Pompeii	158.8	167.6	158.2	136.439	65.271	181.245	136.336	65.203	181.142
San Miguel	123.8	131.8	124.1	375.857	129.199	449.883	375.702	129.100	449.709
Power Plant	68.4	96.8	68.1	340.905	188.276	499.551	340.665	188.124	499.313

Table 38 Left: cost after optimization, middle: optimization, real time [s], right: optimization, user time [s].

Scene	FSW	BIN	INC	FSW	BIN	INC	FSW	BIN	INC
Serapis	37.1	37.8	37.5	9.5	10.0	9.3	2.4	2.5	2.4
Armadillo	24.6	24.6	25.3	4.0	5.2	4.0	2.1	2.6	2.2
Angel	23.0	23.1	23.3	4.7	5.1	4.7	2.3	2.4	2.3
Dragon	40.0	40.4	41.0	8.3	9.6	8.3	2.4	2.7	2.4
H. Buddha	45.1	45.3	46.1	9.9	11.3	9.9	2.4	2.7	2.4
Blade	53.4	51.5	56.6	9.7	18.9	9.7	2.2	3.5	2.2
Hairball	171.5	170.7	175.0	235.6	240.5	235.8	6.2	6.6	6.2
A. Dragon	27.4	28.0	28.1	4.7	5.0	4.5	2.4	2.3	2.3
Sponza S.	36.1	34.7	36.5	25.7	31.9	25.5	2.7	3.3	2.7
Sibenik	15.6	13.8	15.3	7.7	12.1	7.6	2.4	3.2	2.3
P. Plant s. 9	5.6	5.5	5.8	7.9	9.4	7.8	2.7	5.2	2.7
Fairy Forest	18.9	17.7	19.4	9.6	20.2	9.7	2.6	2.8	2.5
Sponza L.	34.3	34.3	34.2	25.6	29.1	24.8	2.3	2.5	2.3
Conference S.	19.8	17.5	19.8	12.3	30.3	12.4	2.5	3.2	2.4
Conference L.	18.9	16.7	18.9	11.6	28.5	11.7	2.6	3.3	2.5
P. Plant s. 16	13.9	12.7	14.1	9.5	18.8	9.5	2.8	8.9	2.8
Soda Hall	35.5	33.9	35.3	16.9	27.2	17.1	2.6	3.6	2.6
Pompeii	31.7	30.8	31.5	31.9	37.6	31.9	2.8	3.2	2.8
San Miguel	31.0	30.4	30.9	15.5	20.3	15.6	3.0	3.3	3.0
Power Plant	13.9	12.3	13.8	13.3	30.0	13.4	2.6	4.0	2.6

Table 39 Left: Sum of surface area of inner nodes, middle: sum of surface area of leaves, right: number of references per leaf.

Scene	FSW	BIN	INC	FSW	BIN	INC
Serapis	4.8	5.1	4.8	24.3	24.6	25.0
Armadillo	2.1	3.1	2.2	18.0	19.3	19.5
Angel	2.1	2.4	2.2	15.0	15.7	15.7
Dragon	2.6	3.0	2.6	18.1	18.3	19.1
H. Buddha	1.8	2.1	1.9	12.4	12.5	13.1
Blade	2.0	3.8	2.3	16.8	16.6	19.8
Hairball	44.4	45.2	48.6	56.3	55.7	63.8
A. Dragon	2.8	3.6	3.1	23.6	24.3	26.2
Sponza S.	17.4	27.1	19.0	48.2	56.0	56.7
Sibenik	9.9	38.5	12.6	71.4	76.0	75.5
P. Plant s. 9	7.9	22.7	8.0	11.2	10.7	11.5
Fairy Forest	9.5	150.6	9.8	41.3	35.7	48.4
Sponza L.	31.2	39.5	23.5	89.5	90.1	86.6
Conference S.	20.8	44.7	21.8	38.3	34.9	37.4
Conference L.	19.2	38.1	18.8	37.5	33.9	36.8
P. Plant s. 16	31.2	88.9	32.5	85.6	85.0	85.4
Soda Hall	18.9	32.3	17.3	63.6	66.0	66.8
Pompeii	48.6	55.4	47.1	58.2	58.9	56.6
San Miguel	22.6	32.0	22.8	106.9	106.8	111.6
Power Plant	20.7	352.2	25.6	114.8	133.9	127.2

Table 40 Left: number of intersection tests per ray, right: number of traversal steps per ray.

Scene	FSW	BIN	INC	FSW	BIN	INC
Serapis	0.412	0.428	0.437	0.411	0.427	0.436
Armadillo	0.303	0.348	0.360	0.303	0.348	0.361
Angel	0.295	0.317	0.337	0.295	0.317	0.336
Dragon	0.354	0.343	0.364	0.353	0.342	0.363
H. Buddha	0.265	0.262	0.289	0.264	0.262	0.289
Blade	0.321	0.341	0.399	0.321	0.341	0.398
Hairball	1.565	1.424	1.801	1.565	1.424	1.800
A. Dragon	0.543	0.558	0.690	0.542	0.558	0.689
Sponza S.	0.715	0.872	0.817	0.715	0.871	0.817
Sibenik	0.853	1.172	0.922	0.853	1.172	0.922
P. Plant s. 9	0.220	0.331	0.219	0.220	0.330	0.219
Fairy Forest	0.570	1.753	0.627	0.569	1.753	0.626
Sponza L.	1.398	1.472	1.262	1.397	1.472	1.260
Conference S.	0.669	0.817	0.665	0.669	0.816	0.665
Conference L.	0.639	0.777	0.612	0.638	0.777	0.612
P. Plant s. 16	1.231	1.782	1.262	1.230	1.781	1.261
Soda Hall	0.861	1.039	0.882	0.862	1.039	0.882
Pompeii	1.306	1.363	1.255	1.306	1.364	1.254
San Miguel	1.600	1.718	1.668	1.599	1.717	1.665
Power Plant	1.556	5.357	1.833	1.556	5.356	1.832

Table 41 Rendering [s], left: real time, right: user time.

Scene	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
Serapis	0.208	0.093	0.343	0.668	2.310	2.3	2.3	2.3	2.3	2.3
Armadillo	0.833	0.371	1.228	2.194	7.165	2.2	2.2	2.2	2.1	2.1
Angel	1.146	0.516	1.940	3.642	12.397	2.3	2.2	2.3	2.3	2.3
Dragon	2.156	0.942	3.783	7.298	24.916	2.3	2.3	2.3	2.3	2.3
H. Buddha	2.724	1.187	4.845	9.464	32.629	2.3	2.3	2.3	2.3	2.3
Blade	4.188	1.747	5.475	10.523	34.934	2.2	2.2	2.2	2.2	2.2
Hairball	7.502	3.146	10.258	18.648	61.541	5.9	5.5	5.7	5.8	5.9
A. Dragon	17.176	7.376	22.999	43.079	140.715	2.3	2.2	2.3	2.2	2.2
Sponza S.	0.188	0.077	0.261	0.512	1.771	2.7	2.7	2.7	2.7	2.7
Sibenik	0.204	0.080	0.299	0.614	2.359	2.3	2.3	2.3	2.3	2.3
P. Plant s. 9	0.303	0.115	0.442	0.886	3.022	2.7	2.7	2.7	2.7	2.7
Fairy Forest	0.427	0.179	0.681	1.380	4.869	2.5	2.5	2.5	2.5	2.5
Sponza L.	0.678	0.278	0.946	1.817	6.115	2.3	2.3	2.3	2.3	2.3
Conference S.	0.717	0.288	0.927	1.847	6.546	2.5	2.5	2.5	2.5	2.5
Conference L.	0.826	0.327	1.085	2.174	7.875	2.5	2.5	2.5	2.5	2.5
P. Plant s. 16	0.919	0.365	1.205	2.354	7.904	2.7	3.3	2.7	2.7	2.7
Soda Hall	5.771	2.283	8.039	15.700	53.694	2.6	2.6	2.6	2.6	2.6
Pompeii	15.684	6.086	22.374	44.568	152.978	2.7	2.7	2.7	2.7	2.7
San Miguel	21.128	8.459	33.179	65.043	224.766	2.9	2.8	2.9	2.9	3.0
Power Plant	33.926	12.640	41.569	78.739	255.741	2.6	2.6	2.6	2.6	2.6

Table 42 Statistics using the EMC64VAR code. left: construction, sequential, user times[s], right: number of references per leaf.

Scene	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
Serapis	0.503	0.474	0.484	0.492	0.491	0.501	0.473	0.483	0.491	0.490
Armadillo	0.399	0.367	0.352	0.389	0.415	0.398	0.365	0.351	0.389	0.415
Angel	0.346	0.333	0.326	0.344	0.350	0.346	0.332	0.325	0.343	0.349
Dragon	0.440	0.412	0.418	0.425	0.420	0.439	0.412	0.418	0.425	0.420
H. Buddha	0.332	0.304	0.309	0.331	0.326	0.332	0.304	0.309	0.331	0.326
Blade	0.523	0.436	0.414	0.480	0.497	0.522	0.436	0.413	0.479	0.497
Hairball	2.254	2.089	1.806	1.744	1.890	2.252	2.088	1.806	1.744	1.890
A. Dragon	0.703	0.681	0.685	0.667	0.699	0.704	0.681	0.685	0.667	0.698
Sponza S.	0.893	0.896	0.947	0.881	0.837	0.891	0.895	0.947	0.880	0.837
Sibenik	0.989	0.956	0.960	1.009	1.000	0.989	0.956	0.958	1.008	1.000
P. Plant s. 9	0.229	0.224	0.230	0.233	0.229	0.227	0.225	0.229	0.233	0.228
Fairy Forest	0.607	0.605	0.614	0.622	0.660	0.607	0.605	0.614	0.622	0.660
Sponza L.	1.584	1.868	1.947	1.789	1.554	1.584	1.868	1.946	1.789	1.554
Conference S.	0.699	0.694	0.701	0.689	0.677	0.698	0.694	0.700	0.689	0.676
Conference L.	0.572	0.558	0.600	0.570	0.595	0.573	0.558	0.600	0.570	0.595
P. Plant s. 16	1.599	1.592	1.469	1.492	1.607	1.599	1.591	1.469	1.492	1.607
Soda Hall	1.193	1.340	1.420	1.614	1.464	1.192	1.340	1.420	1.614	1.464
Pompeii	1.465	1.553	1.498	1.426	1.439	1.463	1.552	1.497	1.425	1.438
San Miguel	1.998	2.029	2.134	2.093	2.012	1.997	2.026	2.131	2.092	2.012
Power Plant	2.879	2.501	3.515	2.360	2.328	2.878	2.499	3.514	2.361	2.327

Table 43 Rendering times using hierarchies built using the EMC64VAR code. Left: real time [s], right: user time [s].

Scene	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
Serapis	31.2	30.0	30.0	30.2	31.0	5.0	5.3	5.2	5.0	4.8
Armadillo	23.4	22.9	21.0	23.6	24.9	2.6	2.7	2.3	2.5	2.6
Angel	18.4	18.1	17.1	18.1	18.7	2.4	2.5	2.2	2.3	2.3
Dragon	24.9	23.9	23.4	23.5	23.9	3.1	3.4	3.1	3.1	2.9
H. Buddha	16.5	15.7	14.9	16.0	16.2	2.2	2.3	2.1	2.2	2.1
Blade	27.9	24.2	22.0	25.1	26.6	2.8	2.7	2.4	2.7	2.7
Hairball	89.8	83.5	70.4	69.8	77.3	56.7	58.7	50.5	47.6	49.1
A. Dragon	31.0	30.2	28.3	28.6	29.7	3.3	3.6	3.2	2.9	2.9
Sponza S.	72.3	72.5	75.8	67.6	68.4	16.2	16.7	16.8	18.8	15.6
Sibenik	85.1	78.4	81.8	84.4	85.1	12.7	16.1	13.1	14.5	11.7
P. Plant s. 9	12.3	12.2	12.3	12.6	12.3	8.4	7.8	8.5	8.3	8.2
Fairy Forest	46.6	46.1	44.7	46.9	51.4	9.6	9.7	9.5	9.6	9.8
Sponza L.	118.4	143.5	146.6	131.5	115.8	30.2	34.5	34.0	32.9	25.6
Conference S.	43.2	42.4	41.4	40.9	40.4	21.5	21.2	21.6	21.7	20.8
Conference L.	36.1	35.4	37.1	35.9	36.8	16.3	15.2	17.0	15.8	16.3
P. Plant s. 16	122.7	115.1	112.6	113.8	116.6	35.9	37.8	31.9	33.0	40.4
Soda Hall	113.9	118.2	129.4	146.3	136.1	15.6	19.1	21.2	19.9	18.3
Pompeii	71.5	74.9	71.3	68.8	70.8	50.7	50.9	51.3	50.0	48.9
San Miguel	149.1	144.9	156.3	159.5	145.7	25.9	28.1	26.8	25.7	25.1
Power Plant	213.7	195.5	255.5	188.0	183.6	41.5	35.8	52.9	28.5	28.6

Table 44 Metrics of hierarchies built using the EMC64VAR code. Left: traversal steps per ray, right: intersection tests per ray.

Scene	AHQ	ALQ	PL1	PL2	PL3	AHQ	ALQ	PL1	PL2	PL3
Serapis	43.3	41.6	41.5	42.2	43.0	9.2	9.6	9.2	9.1	9.0
Armadillo	28.2	28.0	27.6	28.3	28.5	4.1	4.3	4.1	4.0	4.0
Angel	26.1	26.0	25.6	25.8	26.0	4.6	4.9	4.6	4.6	4.5
Dragon	47.7	46.6	46.2	46.3	47.0	8.3	8.6	8.3	8.3	8.2
H. Buddha	54.0	53.0	51.4	51.8	52.8	9.8	10.1	9.8	9.8	9.7
Blade	68.3	63.5	62.3	64.5	66.8	9.6	10.0	9.6	9.5	9.5
Hairball	214.2	222.1	210.1	207.2	206.4	235.3	235.1	233.1	233.5	233.9
A. Dragon	31.9	30.9	30.7	31.1	31.8	4.4	4.6	4.4	4.2	4.1
Sponza S.	45.7	46.6	47.4	45.9	46.0	23.8	23.0	23.3	23.2	23.6
Sibenik	16.8	17.4	17.2	17.4	17.0	7.8	7.9	8.1	8.1	7.8
P. Plant s. 9	5.9	5.9	6.1	6.0	5.9	7.9	8.0	7.9	7.9	7.9
Fairy Forest	21.3	20.4	20.9	21.0	21.4	9.6	9.9	9.7	9.6	9.6
Sponza L.	39.2	39.6	40.6	40.8	39.2	24.9	25.1	25.6	25.1	24.8
Conference S.	21.6	21.4	22.1	21.9	22.0	12.2	12.3	12.2	12.2	12.2
Conference L.	20.3	20.4	21.1	20.4	20.5	11.6	11.5	11.6	11.6	11.5
P. Plant s. 16	16.8	17.2	17.3	17.1	17.0	9.4	10.1	9.4	9.4	9.4
Soda Hall	48.2	48.7	48.0	47.7	46.9	16.7	16.8	16.5	16.5	16.5
Pompeii	35.9	38.2	36.6	36.0	35.5	31.9	32.4	32.1	32.0	31.9
San Miguel	35.7	36.7	37.7	36.6	36.5	15.5	15.6	15.2	15.3	15.3
Power Plant	16.7	17.1	17.7	17.1	16.8	13.4	13.7	13.3	13.3	13.3

Table 45 Metrics of the hierarchies built using the EMC64VAR code. Left: sum of the surface area of inner nodes divided by the surface area of the whole hierarchy. Right: sum of the surface area of leaves multiplied by number of triangles contained in them.

Appendix B

DVD Content

In this appendix, we describe the contents of the DVD added to the printed version of this thesis. There are several folders on the DVD:

- scenes - this folder contains the 20 testing scenes we used for our measurement. The scenes are provided in .obj format. It also contains the scenes in binary format. We implemented the application to load the scenes using these files (if available) rather than the .obj files, because it is significantly faster. This folder also contains the camera definition files we used for our measurement and the rendered images of the scenes.
- text - contains a folder with all the latex files and images we used and created while writing the text. It also contains the PDF version of our thesis.
- documentation - contains the documentation of our source code generated using the Doxygen software.
- src - contains the source codes of our implemented methods and the nanoGOLEM ray tracing framework. Also contains the MS Visual Studio project we used during the development.
- bin - contains the executables for MS Windows and Ubuntu.