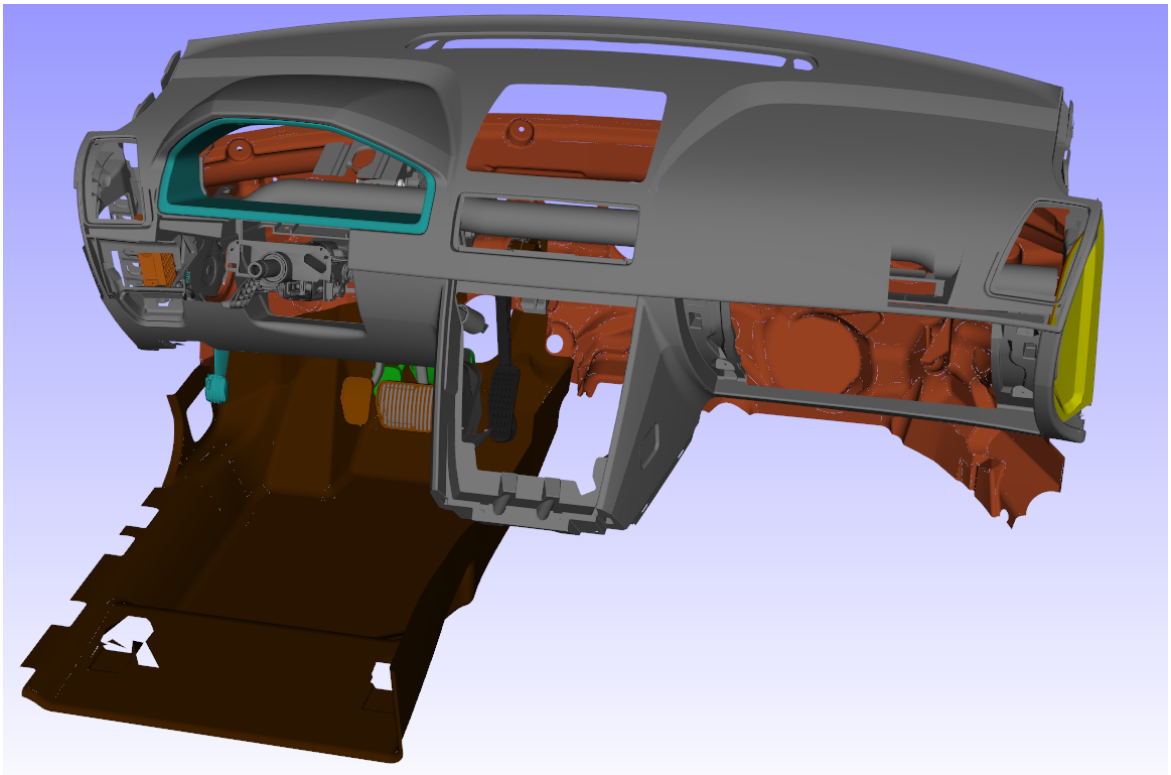
CHALMERS



SIMD Optimized Bounding Volume Hierarchies for Fast Proximity Queries

Master of Science Thesis in Computer Science – Algorithms, Languages and Logic

ROBIN YTTERLID

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2013

SIMD Optimized Bounding Volume Hierarchies for Fast Proximity Queries

ROBIN YTTERLID

Examiner: ULF ASSARSSON

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 10 00

[Cover: 3D model of the front panel of a car]

Department of Computer Science and Engineering
Göteborg, Sweden October 2013

Abstract

3D models of physical objects are used in an ever-growing number of areas to help visualize and simulate digital environments. Applications must often simulate complex processes involving physical phenomena such as forces, velocities and physical interactions between objects. In such environments, it is crucial to be able to effectively determine proximity between objects by using collision- and distance tests. As the number and complexity of 3D models increases, together with an increasing demand for simulation precision and realism, heavy demands are placed on the performance of the proximity tests that are used.

This thesis investigates the possibilities of increasing proximity test performance by combining Bounding Volume Hierarchies, which are common data structures for accelerating proximity tests, with a certain method for parallel computation called SIMD. Some SIMD-based construction strategies are presented and shown to increase proximity test speed by up to 50% and reducing BVH memory footprint by up 60%.

Acknowledgements

This study was performed at the Fraunhofer Chalmers Centre (FCC) in Göteborg during the spring and summer of 2013. I would like to thank the staff at FCC for giving me access to their computers, software and free fruit, as well as giving me my own room to work in, with the office's best window view as an added bonus. Special thanks goes to my supervisor Evan Shellshear at FCC for the immense amount of help and guidance he has offered during the thesis work, and Ulf Assarsson at Chalmers for his valuable feedback.

Contents

1

Introduction

3D modeling of physical environments is used in an ever growing number of areas to help digitize and automate complex simulation calculations. This allows approximations of physical processes to be described and analyzed efficiently by computers. Notable areas where 3D models are used in this way include automation of industrial production processes, path planning, computer vision and digital entertainment.

One of the core usages of 3D modeling is to simulate how different moving objects interact physically with each other and with a static environment. Proximity queries such as distance and collision tests between the objects are often necessary to do this. To determine how the objects themselves behave, the interactions of their respective geometric primitives must be studied. For example, two objects each consisting of a triangular mesh collide if and only if some of their triangles are in physical contact with each other.

New applications demand greater and greater precision and realism, which increases the demand for more detailed models with better primitive resolution. Increasing the number of primitives has the drawback of making queries such as collision detection and distance tests more computationally expensive. To mitigate this effect, special algorithms and data structures such as Bounding Volume Hierarchies are used avoid many of the unnecessary primitive tests. Improving performance of BVHs can be done in multiple ways, for example by tuning the algorithms used, making the memory access more coherent to improve cache coherency, adding parallelism or using more efficient types of bounding volumes.

One way of adding parallelism to a program is to utilize special hardware called SIMD units that are present in modern processors. SIMD is an acronym for Single Instruction Multiple Data, and is based on the concept of running a single processor instruction on more than one piece of similar data simultaneously. SIMD instructions are particularly well suited for 3D applications where collision and distance tests need to be performed between multiple geometric primitives. A similar concept to SIMD is GPGPU, where a Graphics Processing Unit (GPU) is used instead of the CPU to perform simple computations on large sets of data. While both SIMD [1] and GPGPU [2] have been used to improve the performance of collision detection, there is still a need for improvement for other queries such as distance

tests.

1.1 Purpose

The primary goal of this thesis is to explore the possibilities of increasing computation speed of BVH-based distance queries in a proximity query library by using SIMD routines. A secondary goal is to investigate a compiler for a C++-like language that automatically utilizes SIMD, and determine its viability for optimizing proximity queries. The purpose of this investigation is to determine whether sequential C++-based BVH algorithms can be easily, almost directly translated to SIMD-optimized, parallel versions while obtaining the same performance as that of hand-written SIMD-optimized algorithms. Other secondary goals include increasing the speed of collision queries, keeping BVH construction time low, and producing more memory-efficient BVHs. In this thesis, the following questions are answered:

- Can regular, sequential primitives query routines be easily ported to a programming language that automatically utilizes SIMD? Does the performance match that of handwritten SIMD algorithm packages?
- What problems can be expected when using SIMD in conjunction with BVHs?
- How does the structure of a BVH affect the efficiency and number of required SIMD proximity routine calls?
- Is there a good build strategy that produces BVHs that are well-optimized for SIMD enhanced collision and distance queries?

1.2 Method

This section provides the research steps performed to produce the results of this thesis:

- Investigation of the usability and performance of the compiler ISPC by Intel [3], and its programming language that can describe SIMD-optimized code using syntax similar to regular C++.
- Modification of preexisting SIMD routines for collision and distance tests between triangles to make them useable for BVHs with up to four triangles in each leaf node.
- Testing different strategies for constructing BVH leaf nodes and determining whether they lead to improved performance, less memory usage or other benefits.

- Testing different strategies for constructing the rest of the BVH in order to make the RSS tests become as fast as possible and to group the input triangles to the leaf node build step as good as possible.
- Investigation of the performance and memory tradeoffs of having more triangles in each leaf node, thereby reducing the efficiency of the BVH, versus having less triangles and therefore less SIMD utilization in the triangle tests.

1.3 Delimitations

There are a multitude of different processor architectures out on the modern market, and not all of them use the same instruction sets. For practical reasons, this thesis only considers the SIMD instruction set SSE from Intel.

It is possible to create BVHs with different types of geometric bounding volumes, all with different properties, pros and cons. This thesis only considers a type of volume called Rectangle Swept Spheres (RSS). RSS hierarchies are fast for many types of proximity queries, particularly distance queries [4], but can be slightly slower to construct than some other volumes such as AABBs. As such, RSSs may not be the best choice for applications where the scene changes and the BVH has to be rebuilt quickly.

Writing programs that utilizes SIMD is a difficult problem that this thesis addresses, and tools exist that automate parts of the process. This thesis investigates only one such tool, namely the compiler ISPC. While ISPC is relatively easy to use, it is not suitable for all applications.

A lot of research has previously been conducted in the subjects of ray tracing and collision detection, and many of the properties of these problems have been proven both analytically and empirically. The subject of distance tests has been studied to a much smaller extent, and therefore lacks the scientific foundation that exists for collision detection. The main purpose of this thesis is to empirically explore optimizations of distance queries, and because of the lack of theoretical foundation in the subject many of the choices made are instead based on heuristics and theory that works well for collision detection. An argument for why much of the theory of collision detection should also apply to distance calculation is done in section 4.2.1.

1.4 Outline of the thesis

In order to fully understand the topics that are covered in the thesis, the reader needs some basic knowledge of the central concepts. Chapter 2 is meant to help the reader by introducing SIMD and a special type of bounding volume called RSS that will be discussed throughout the rest of the paper.

Chapter 3 gives an overview of the work previously done within the fields of BVH construction, proximity queries, and SIMD optimization.

Chapter 4 provides an in-depth description of the tools, benchmarks and theories used to conduct the study. It also connects results and conclusions from relevant research previously done within the field, and presents the most important factors that affect the performance of the type of BVHs that are used in the thesis. Finally, it presents the BVH construction strategies that were developed.

Chapter 5 shows the results of the benchmarking process, and points out the most important observations.

Chapter 6 interprets the results from chapter 5. It also summarizes the contents of the thesis and the most important conclusions. Finally, some ideas are presented for future research within the field and possible extensions to the study made in this thesis.

2

Theory

This chapter gives some theoretical background for RSSs and SIMD, which are two of the most central concepts explored in this thesis. Note that the reader is assumed to be familiar with the basics of bounding volume hierarchies, which is the third central concept.

2.1 Rectangle Swept Spheres (RSSs)

The Rectangle Swept Sphere (RSS) is one member of a family of bounding volumes called Swept Sphere Volumes [4]. A swept sphere volume can be created from a simple base geometric shape such as a point, line or rectangle by growing it outward in all directions by some offset r. Mathematically, this corresonds to a convolution between the base shape and a sphere with radius r. Figure 1 shows a point, line and rectangle with offsets added. Figure 2 shows their 3D representations.

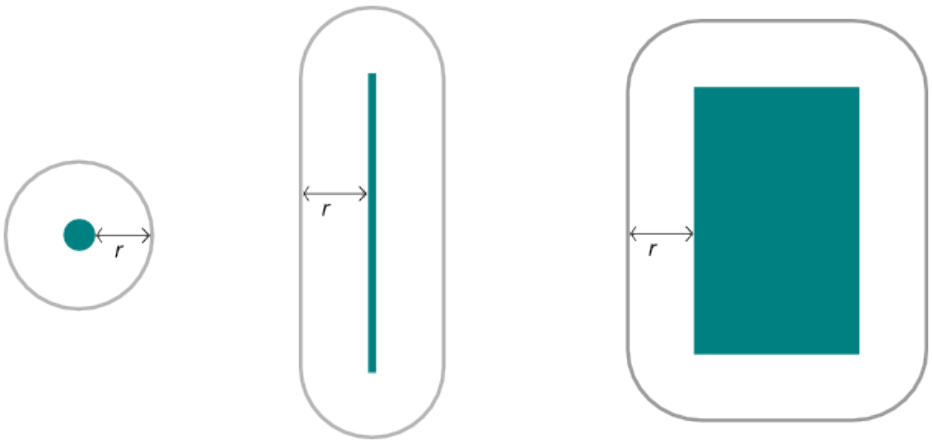


Figure 1: A point, line and rectangle grown outward by a radius r to create three swept sphere volumes.

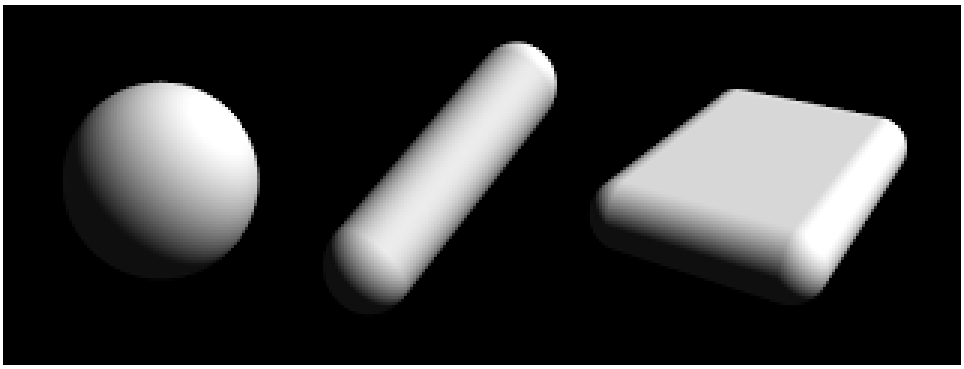


Figure 2: The point swept sphere is a sphere with radius r, the line swept sphere resembles a medical pill, and the rectangle swept sphere resembles a very fluffy pillow.

One of the greatest benefits of swept sphere volumes compared to other shapes such as bounding boxes is that distance calculations become very efficient. The problem of measuring distance between two swept sphere volumes is as simple as calculating the distance between their base shapes and then subtracting their radii. Different kinds of swept sphere geometries can be used within the same hierarchy as long as the distance tests between their base shapes are easy to calculate. Collision detection between two swept sphere volumes can be handled by determining whether or not their distance is zero. Little space is required to store sphere swept volumes since they can be defined comletely by their simple base shape and the radius.

When used as BVs around arbitrary 3D objects, rectangle swept spheres provide a tighter fit around the underlying geometry than point- or line swept spheres, and is therefore more efficient overall [4] . In this thesis, all BVHs consist of RSSs only.

2.2 Single Instruction Multiple Data (SIMD)

SIMD stands for Single Instruction, Multiple Data, and is one of the four modes of parallelism described in Flynn's taxonomy [5]. In a SIMD architecture, a single instruction is executed by several processing units simultaneously, each on its own set of data. Figure 3 shows an illustration of this. One of the better-known modern examples of SIMD architecture is the graphical processing unit (GPU), that can run shader programs to calculate the colors of thousands of screen pixels at once, or utilize general-purpose gpu programming (GPGPU) tools such as NVIDIA's CUDA [6] to solve massively parallel computation problems.

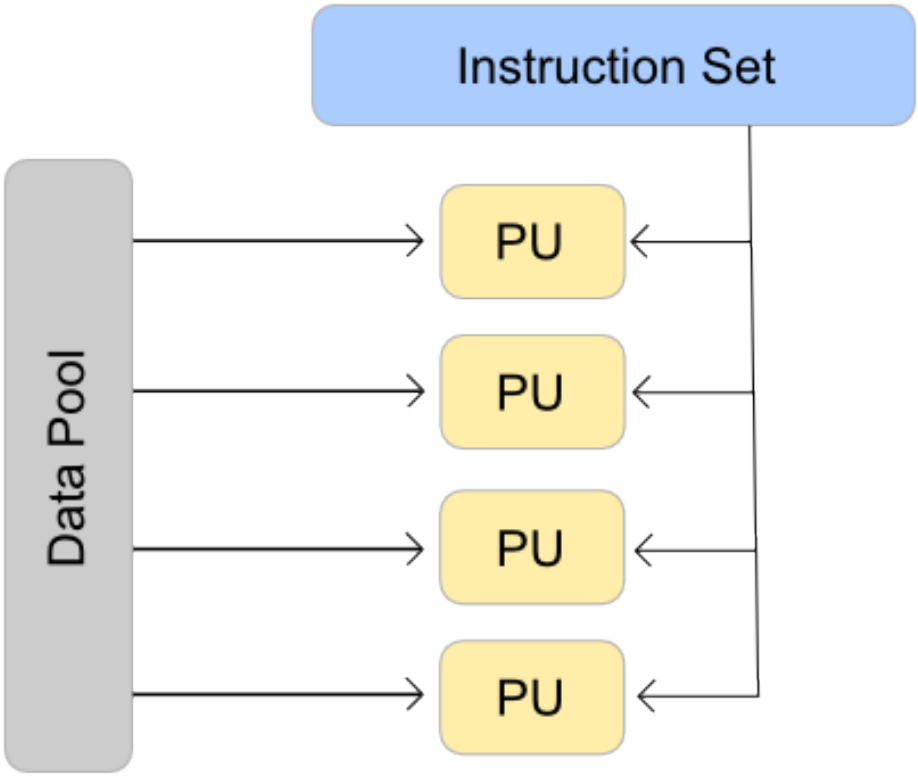


Figure 3: An illustration of the SIMD model in Flynn's taxonomy. A single instruction is executed by many processing units at once, each on its own data.

One of the main disadvantages of SIMD structures is the problem of not being able to use normal control-flow structures. Branching statements such as if-then-else clauses and conditional expressions cause problems in a SIMD environment, where different data that are being processed need to take different routes through the control flow. SIMD can only be utilized optimally when no control-flow structures are present, which greatly reduces the number of possible algorithms that can be used.

When an instruction is to be performed on a piece of data, that data must first be fetched from memory, then returned to its destination afterwards. What this means for SIMD is that in order to run a single instruction, many pieces of data need to be moved before and after its execution. This can slow down the execution process unless all the data can be moved simultaneously. The typical way that systems can do this is by performing vector load and store operations, where a piece of contiguous memory is copied and transferred at once. In order to maximize the benefit of SIMD, the user must therefore be careful with data layout and use continous memory data

structures such as arrays in favor of pointer-based structures such as linked lists.

Intrinsics, SSE and AVX

Modern CPUs for personal computers include SIMD units that programmers can use to accelerate processing by calling certain instruction sets. The instructions can either be called directly using assembly code, or by a special set of functions called intrinsics that are built into modern compilers. The two most widely used SIMD instruction sets for personal computers are SSE and AVX by Intel. SSE performs operations on 128-bit registers, each of which can hold an amount of data equivalent to four single-precision or two double-precision floating point numbers at once. Under optimal conditions, calculations that utilize SSE can therefore run up to four times faster than their purely sequential counterparts. AVX is an instruction set for newer processors that uses 256-bit registers, which can hold the equivalent of eight single precision floating point numbers, giving a potential speedup of up to eight times.

In this thesis, all SIMD-optimized proximity tests were programmed or configured to utilize the SSE instruction set.

3

Related work

This chapter contains short descriptions of previous work related to SIMD, ISPC, BVHs, RSSs and proximity queries.

Investigation of ISPC performance

Smith's master thesis covers the usage of SIMD for 3D applications [7]. A large part of the thesis discusses ISPC, its performance, and its usability. While most of Smith's conclusions are consistent with those made in this paper, there are some notable differences. Section 7.1 in this paper briefly discusses those differences.

Parallelising proximity queries

SIMD is only one way of parallelizing software. Many other parallel methods and algorithms for faster proximity queries have been studied. Ruipu et. al.'s paper [1] is a compilation of some of the most sucessful parallel optimizations for collision detection, and it even contains a benchmark to compare their performance.

Chen Tang's paper introduces a very efficient method for CPU-based continuous collision detection using SIMD [8]. Min Tang has also developed an approach for continuous collisiondetection, but this one uses the GPU instead of CPU. Tang claims that the approach may even have potential for improving distance- and other types of proximity queries [2].

BVH construction

Ingo Wald's paper [9] investigates the problem of increasing construction speed of BVHs, while keeping the traversal speed of the resulting BVHs high. The paper borrows concepts from kd-tree construction and applies them to BVHs with great results. It also explains the concept of tree construction using surface area heuristics (SAH), which is claimed to be the best algorithm currently known for maximizing the traversal efficiency of BVHs and kd-trees. Several CPU-based algorithms are presented, all of which are faster than traditional SAH when constructing BVHs, and some of which use parallelization of different kinds.

Lauterbach et. al. took the task of parallelization a step further, and developed two algorithms for constructing BVHs using GPUs [10]. One of the algorithms uses top-down surface area heuristics similar to those described by Ingo Wald, and a comparison to Wald's results are included in the paper.

Swept Sphere Volumes

In their paper, Larsen et. al. introduced the concept of swept sphere volumes to the subject of BVs [4]. The paper gives a brief explanation of general BVHs and how proximity tests are typically performed between them. It proceeds to explain the benefits of using SSVs instead of traditional BVs for proximity tests, and provides methods for how to construct BVHs of three different kinds of SSV. They conclude that among the three SSVs presented, RSSs are the most tightly fitting and by far the fastest.

4

Procedure

This chapter describes the languages and libraries that were used during the thesis work and justifies the decisions that were made. The most important problems that have to be solved for efficient BVH construction are discussed, and the construction strategies are introduced. The chapter also describes the benchmarking procedure used in the thesis.

4.1 Software and tools

This section contains descriptions of the software used in the thesis, and motivates why they were used.

4.1.1 ISPC

Many libraries exist that act as frameworks for SIMD utilization for modern programming languages, but they often require the user to have some level of understanding of how SIMD works, how to best utilize it and what programming styles should be embraced or avoided in order to reach maximum performance. Intel's ISPC [3] is a good option for abstracting away the technical details as much as possible. ISPC is short for Intel SPDM Program Compiler and works under the Single Program Multiple Data (SPDM) model where program code is written in a sequential manner, but the program then executes on more than one piece of input data in parallel.

ISPC compiles a special language that closely resembles a lightweight version of C++, with control flow statements such as loops and if-statements that are usually not easy to use efficiently in a SIMD environment. The first main benefit of this is that many sequential algorithms that are difficult to express with assembly code or intrinsics can be SIMDoptimized by moving the code to an ISPC kernel. The second benefit is that that the resulting code becomes more readable and maintainable than if low-level intrinsics or assembly had been used.

The compiler produces C++ object files, which can be linked to regular C++ files in the usual manner. This means that a C++ program can directly call functions from and share data with an ISPC program without additional formatting or other overhead, and vice versa. The resulting object

files automatically utilize the hardware's SIMD instruction sets. Switching to a different target architecture or instruction set can be done easily by changing the compiler's options, without need to change the code itself.

The first reason for investigating ISPC in this thesis is to determine whether or not automatic SIMD code generating tools can produce programs that match the efficiency of specialized libraries that directly use assembly or intrinsics code. The second reason is to make a case study in the usability of ISPC, to determine just how easy it can be to add SIMD support to programs by moving existing sequential algorithms into ISPC kernels. In this thesis, ISPC was configured to run the SSE instruction set.

4.1.2 Embree

Embree [11] is a library of kernels designed for ray tracing. It includes SSE- and AVX-based routines that offer the user a higher level of abstraction than pure intrinsics code. In this thesis, Embree was used to develop the proximity routines that served as baselines with which to test the performance of the ISPC routines. The Embree routines were later used for the primitives tests between BVHs. All SIMD-optimized routines used in this thesis except those developed with ISPC use Embree as an interface to the SIMD instruction sets. The Embree routines were all configured to use SSE.

4.1.3 PQP

PQP (Proximity Query Package) is a library for proximity queries using bounding volume hierarchies of rectangle swept spheres [12]. It has methods for constructing BVHs by recursively splitting a set of triangles into two parts using a splitting axis and a coordinate on that axis. The triangles whose centroid is located to the left of the splitting coordinate are placed in the left subtree and vice versa. RSSs are then created for each of the two parts. When BVHs have been constructed for two objects, proximity queries can be called to determine whether or not they collide and their minimum distance to each other.

A heavily modified version of PQP was used during this thesis in numerous ways. First, it provided a baseline implementation with which to benchmark the ISPC and Embree proximity routines and also the construction strategies developed in the thesis. The code for all of the BVH construction strategies presented in section 4.2 was based on routines from the modified PQP. The construction strategies were tested by running them from PQP during the BVH construction phases, and comparing their performance to the original PQP construction.

4.2 Construction strategies for RSS hierarchies

This section describes some of the most important problems that have to be considered in order to construct good SIMD-optimized BVHs for collision- and distance queries. Section 4.2.1 motivates why much of the theory of collision tests and solutions to collision detection problems should also apply to distance tests. Sections 4.2.2–4.2.7 describe the different factors that have proven to affect the efficiency of traditional BVHs, and presents some similar factors for the SIMD-optimized variant. Based on these factors, the final heuristics that were developed and used to construct BVHs in this thesis are presented in section 4.2.8.

4.2.1 Applying RSS-based Collision test theory to distance tests

Distance tests are very closely related to collision tests, especially so when RSSs are used [4], which means that some of the analytical observations regarding collision detection can be applied to distance tests as well.

To illustrate the close relation between distance- and collision tests, consider a function RSS that describes two RSSs by their rectangles R, R' and radii r, r' in the following way: $RSS(R, r)$ and $RSS(R', r')$, and a function $Dist(X, Y)$ that describes the signed distance between two RSSs X and Y. The expression

$$Dist(RSS(R, r), RSS(R', r')) = d$$

is equivalent to

$$Dist(RSS(R, (r + d/2)), RSS(R', (r' + d/2))) = 0$$

i.e. the radii of the RSSs can be grown by $d/2$ each so that they just barely collide. See Figure 4 for an illustration. The problem of determining whether any two RSSs are a particular distance apart can thus be solved by a collision test after their radii have been grown half that distance. Many collision tests can likewise be expressed as distance tests by shrinking the RSS radii. Larsen et. al. [4] argued that the logic and arithmetic for RSS-based distance- and collision tests are the same, and that the analytical properties of collision tests also hold for distance tests. In the rest of this chapter, unless otherwise noted, any observations regarding the efficiency of collision tests are assumed to also hold for distance tests.

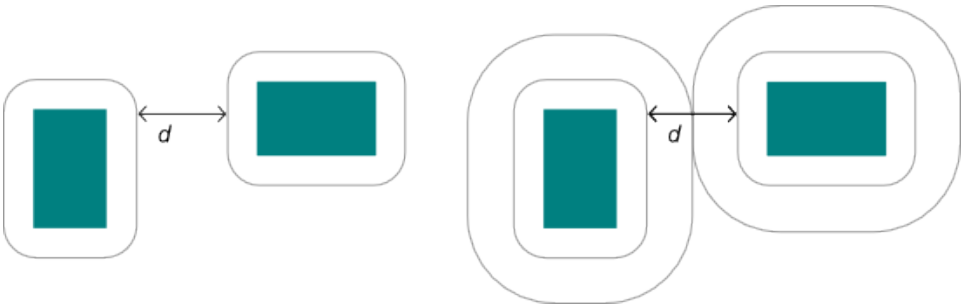


Figure 4: Illustration of two RSSs before and after their radii have been grown by $d/2$

4.2.2 Factors affecting the number of proximity tests between BVs

To make BVHs as efficient as possible when performing distance queries, the number of tests between both the triangles themselves and their bounding volumes should be kept low. In particular, during a test where the current closest distance found is d, it is wasteful to test bounding volumes whose triangles are a distance $> d$ apart. Minimizing the ratio of the number of bounding volumes with a distance $> d$ between them and the number of triangles with distance $< d$ should be a good idea. Zachmann claims that the performance of BVH collision detection depends on two properties of the BVs: their tightness of fit around the triangles, which affects the number of BV collisions, and their simplicity, which affects the efficiency of the overlap tests [13]. RSSs are both tight-fitting and easy to test for overlap and minimum distance [4], and are therefore perfect candidates for proximity testing in BVHs. Section 4.2.5 discusses the topic of tightness further.

Suri *et al.*proved that in hierarchies of axis-aligned bounding boxes, the ratio between the number of collisions between the BVs and the number of collisions between the objects inside are bounded by two factors: *Aspect Ratio* and *Scale Factor* [14]. Scale factor was proven to be the less important of the two, and is defined as the size difference between the largest and the smallest of the objects enclosed by AABBs in the BVH. Aspect ratio is a measure of the elongatedness of the objects. In their paper, Suri et. al. chose to define the three-dimensional aspect ratio of an object as the size difference between the smallest axis-aligned cube that can contain the object and the largest axis-aligned cube that fits inside it. The aspect ratio for the entire BVH is equal to the maximum aspect ratio among the objects inside.

The aspect ratios of objects inside the BVs of a BVH are important indicators of its efficiency. It is impractical to directly calculate aspect ratios as defined by Suri, however, since that would require calculating the sizes of the biggest axis-aligned cubes that fit inside the objects. Also, when

a BVH is constructed, the objects inside are repeatedly split into smaller pieces. Since the final shapes and sizes of the objects are unknown until the construction is finished, it seems difficult to measure and make intelligent decisions based on aspect ratios during the construction process. Because of these problems, other ways of measuring elongatedness were used during the thesis work. Section 4.2.3 describes the most important of those methods.

4.2.3 Measuring elongatedness

Elongatedness can be described as a relation between the length and size of an object. There are multiple ways of measuring length and size, and their relation to each other. This section describes the different measurements used in this thesis.

A simple way of measuring length is to calculate the maximum distance between two points within the object. The objects can be of arbitrary shape, which complicates the task of finding two such points in the objects themselves. It is, however, simple to calculate the length of the surrounding bounding volumes. The maximum distance in any sphere-swept geometry can be calculated by determining the maximum distance in the geometry and adding the diameter of the sphere. For an RSS the following holds:

$$MaxDist = \sqrt{l^2 * w^2} + 2 * r$$

r, l, and w denote the sphere radius, rectangle length and rectangle width, respectively. Figure 5 shows a two-dimensional illustration of the maximum distance inside an RSS.

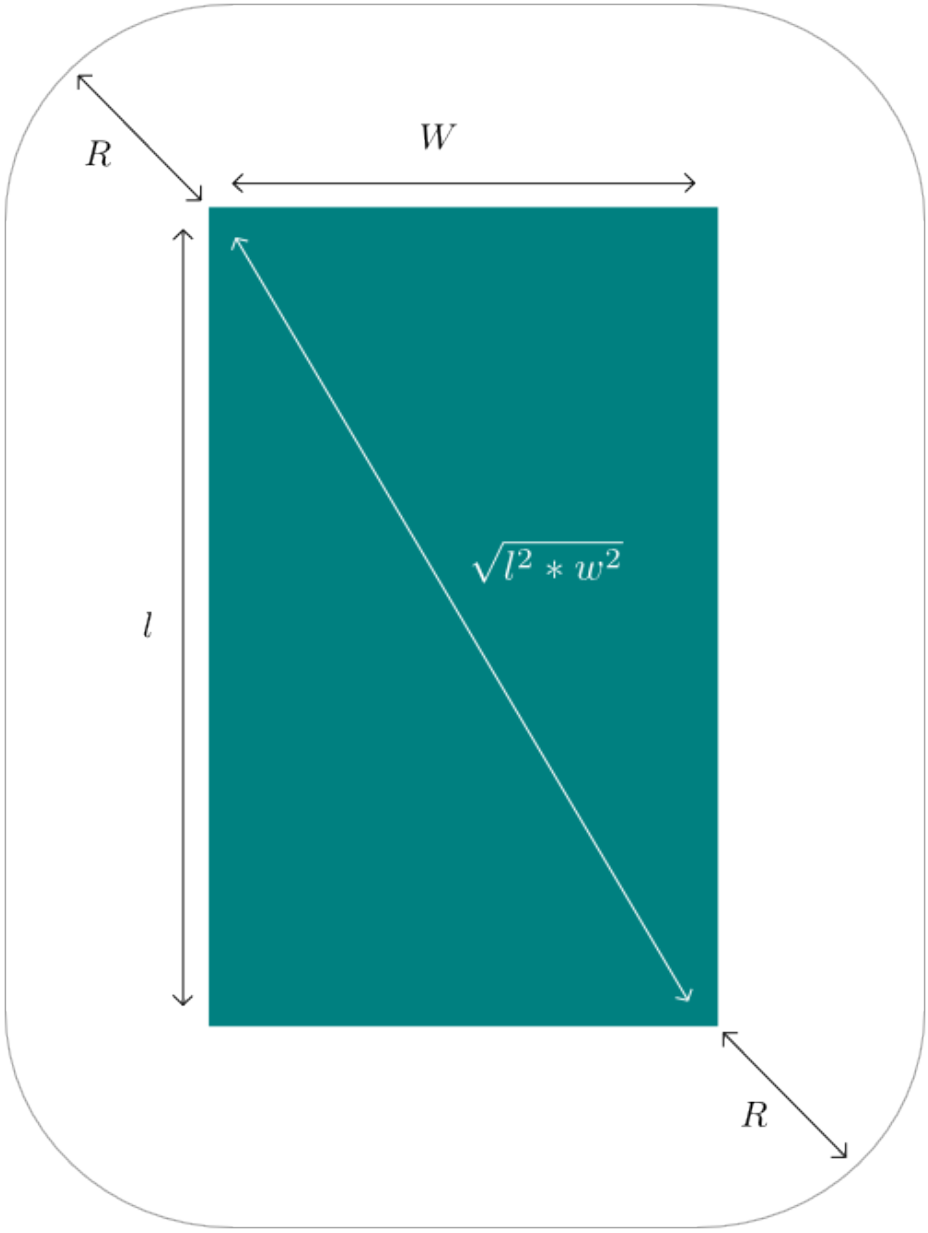


Figure 5: The maximum distance between points in an RSS is equal to twice the radius plus the length of the rectangle's diagonal.

The volume and surface area of the enclosing BV are perhaps the most straightforward ways of measuring an object's size. Surface area can in many cases also be used as a measurement of length since longer convex objects generally have bigger surfaces than more compact ones. Both these properties can be easily calculated for an RSS by splitting it into different

parts and analyzing each part separately. The 3D volume defined by an RSS can be split into three subparts: a central cuboid B, two pairs of halfcylinders X_1, X_2, Y_1, Y_2 at the sides, and four parts C_1, C_2, C_3, C_4 of a sphere in the corners. Figure 6 shows a 2D representation of this concept.

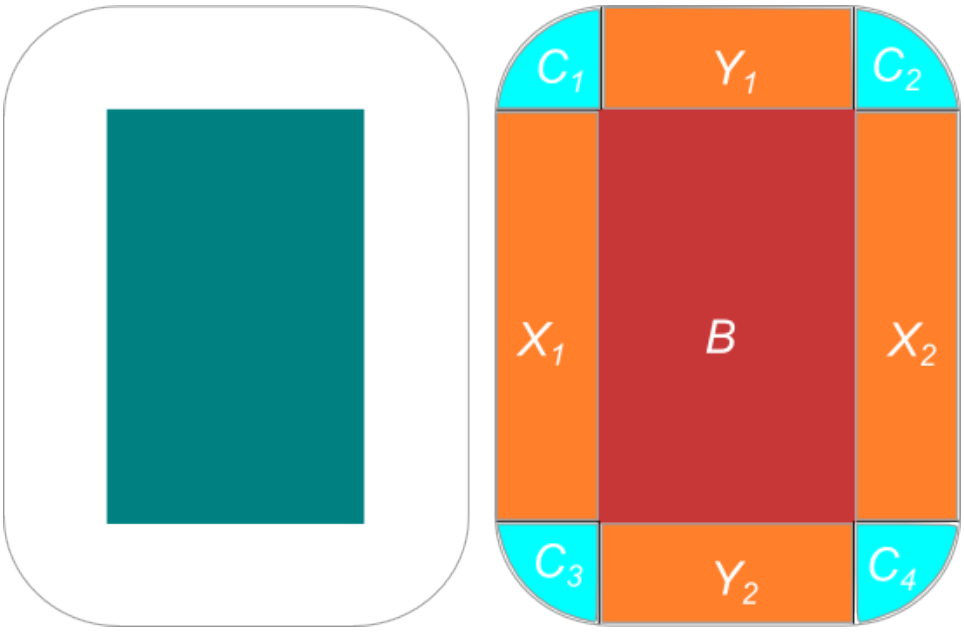


Figure 6: An RSS defined by its rectangle and radius to the left, and the volume of the RSS split into its subparts to the right. The subparts are a central cuboid B, two pairs of halfcylinders X_1, X_2, Y_1, Y_2, and four parts of a sphere C_1, C_2, C_3, C_4.

The properties of the subparts of an RSS can be used to calculate the volume and surface area of the RSS itself. Equations 1 and 2 describe the volume VRSS and surface area ARSS of an RSS expressed in terms of the properties of its subparts, which in turn are expressed in terms of the RSS's radius and rectangle width and length, r, w, l. Note that in equation 2, only the surfaces of the subparts that belong to the surface of the RSS are included, e.g. only the top and bottom faces of the cuboid, not the sides.

$$\begin{aligned} V^{RSS} &= V^{Sphere} + V^{Cuboid} + V_X^{Cylinder} + V_Y^{Cylinder} \\ &= \frac{4\pi r^3}{3} + 2rlw + \pi r^2 l + \pi r^2 w \end{aligned} \tag{1}$$

$$\begin{aligned} A^{RSS} &= A^{Sphere} + A^{Cuboid} + A_X^{Cylinder} + A_Y^{Cylinder} \\ &= 4\pi r^2 + 2lw + 2\pi rl + 2\pi rw \end{aligned} \tag{2}$$

4.2.4 Reducing elongatedness of RSSs

To reduce elongatedness of an RSS, the parameters r, l and w can be tweaked to decrease $MaxDist$ or increase the volume. Consider an RSS with a fixed volume. The most efficient way to decrease $MaxDist$ while keeping the volume unaffected would be to slightly increase r while greatly decreasing l and/or w. The reason is that the RSS volume is proportional to r^3, l and w. A small increase in r would require a large decrease in l or w to keep the volume fixed. Meanwhile, $MaxDist$ is proportional to r, w and l, which means that the small increase in r combined with the large decrease in w and/or l would greatly reduce $MaxDist$. Reducing the size of the RSSs rectangle and increasing its radius should therefore be an efficient way of making it less elongated. In the extreme case where the length and width of the rectangle is reduced to 0, the resulting RSS is a perfect sphere.

Since all points on the surface of a sphere are a constant distance from the center, they are the geometric shapes with minimal length for a given volume. Spheres are therefore the shapes with the least amount of elongation.

4.2.5 BV tightness and surface areas

When used as BVs, spheres do not always fit the enclosed object tightly enough [4, 15]. According to Wald and Zachmann, tighter fitting BVs are more efficient [9, 13], so to build efficient BVHs, tightness should be considered as well as elongation and size factor. The best known method for producing tight-fitting bounding volumes and kd-tree nodes is to use surface area heuristics (SAH) [9]. The basic idea behind SAH is to recursively separate a 3D volume into two subvolumes by splitting the primitives in such a way as to minimize some cost function based on the subvolumes' surface areas. Wald provides such a cost function in his paper:

$$Cost(V \rightarrow \{L, R\}) = K_T + K_I(\frac{SA(V_L)}{SA(V)}N_L + \frac{SA(V_R)}{SA(V)}N_R)$$

Here, $Cost(V \rightarrow L, R)$ denotes the cost of dividing the volume V into a right and a left half R and L. K_T is the cost of one traversal step through the hierarchy. K_I is the cost of one primitive proximity test. $SA(V)$ is the surface area of V, and N_L and N_R are the number of primitives in the left and right halves, respectively.

4.2.6 Sampling splitting locations

Popov and Hunt both conducted studies that investigated how to best split scenes to produce efficient kd-trees [16, 17]. They used sampling algorithms to produce a number of possible partitions of the triangles in the scene and cost approximation functions that predicted which partitions would be

the most effective. In the context of RSS-based BVHs, it would likewise be good to have a sampling algorithm and cost function to make more informed splitting decisions.

BVH construction is a more complicated process than kd-tree splitting [10]. In each level of a kd-tree the triangles are separated by a splitting plane that is defined by two out of the three fixed axes, but in a BVH the triangles can be separated in an arbitrary way, which makes the task of sampling different partitions more complex. In the BVH case it is also hard to make a cost function that can accurately approximate the efficiency of partitions unless their BVs are first constructed and analysed, which requires additional processing.

Despite the fact that sampling is more difficult in a BVH than in kd-trees, some of the results of Popov and Hunt can still be used as general guidelines for RSS hierarchy splitting. Although not explicitly mentioned in their papers, the efficiency of their partitions tends to be better when the splits are made close to the middle of the groups of triangles. The splitting heuristics used in this thesis were developed with this observation in mind.

4.2.7 Grouping triangles for efficient SIMD utilization

Since the hierarchies used in this thesis can have multiple triangles in each leaf node, and can utilize SIMD to test many triangles in a leaf at once, there is a need to construct the hierarchies in such a way that many triangles are grouped together in leaf nodes so that they can be tested simultaneously. To provide the BVH with ample opportunities to use SIMD, it appears to be a good idea to try to avoid creating nodes with only a single triangle, or more generally to try to have as many triangles in each leaf as possible. There are complications with such a direct approach, however. Grouping many triangles in leaves means that if one of them needs to be tested, the others are forced to be tested as well. This can lead to extra work if many of the triangles are tested unnecessarily. The main point of using BVHs is to avoid those unnecessary tests by splitting dissimilar triangles that are unlikely to be tested together into different BVs. A balance must be struck between packing lots of triangles into a small number of leaves, thereby making good use of SIMD, and splitting leaves with groups of dissimilar triangles into many smaller leaves, to increase the BVHs pruning ability. The solution is to find a heuristic that decides whether or not a group of triangles are sufficiently likely to be tested together to justify not splitting them.

The likelihood of a group of triangles being tested together depends on their respective shapes and positions, but also on the shapes and positions of the triangles they are supposed to be tested against. Therefore, even if a certain grouping of triangles in an object is optimal for testing against a grouping of triangles in a second object, it may not be optimal when testing against other groupings, or against a third object. Also, if the objects are

allowed to move, the grouping may no longer be optimal after one of the involved objects changes position or orientation. For simplicity, this thesis focuses on optimizing for the general case, when the exact properties of the objects involved and the proximity tests to be performed are unknown.

When a triangle is involved in a collision or distance test against another object, it means that the triangle is considered a likely candidate for containing the closest point to that object. Intuitively, when a triangle has a high probability to contain a certain point, other triangles that are far away from that triangle are less likely to contain that point. It appears to be wasteful to group triangles that are scattered far away from each other into the same leaf node. The construction strategies developed and tested in the thesis use several heuristics for determining whether or not a group of triangles are close enough to each other to be put into the same leaf.

The proximity routines are all based on the SSE instruction set, and can test up to four triangle pairs in parallel. This means that the queries are as efficient as possible whenever the number of triangle tests to be performed between two RSSs equal some factor of four. More specifically, the most efficient cases for primitive tests are 1vs4, 2vs2, 2vs4, 3vs4 and 4vs4 triangles. Note, however, that only tests with up to 1vs4 or 2vs2 triangles can be performed in a single step by SSE-based routines. For example, testing two leaves of four triangles each requires 16 triangle tests, which translates to four SSE routine calls. In general, leaves with odd numbers of triangles should be avoided if possible.

4.2.8 Splitting strategies

Based on some of the observations made in sections 4.2.1 4.2.7, two categories of heuristics for constructing RSS hierarchies for fast distance and collision queries were developed. The first category of construction strategies is responsible for splitting nodes with five or more triangles. Those strategies work in similar ways as traditional construction methods, and they ensure that the BVs are tight and that the BVHs pruning abilities are good. The second category works solely on nodes with four or less triangles and specializes in optimizing the BVH leaves for SIMD-based primitive proximity tests. The methods from the first category are called *general splitting strategies*, and those from the second category are called *leaf level splitting strategies*. During the construction of a BVH, exactly one general and one leaf level strategy is used.

Leaf level strategies can only decide if and how the very small groups of triangles in leaf nodes should be split. Their task is to determine whether the benefits of grouping the triangles together, and utilizing SIMD to a greater extent, outweighs the benefits of splitting them and improving the pruning power of the BVH. Leaf level strategies can produce better results and more useful triangle groupings when a larger portion of its input triangles are

somewhat similar, i.e. not too spatially scattered. Therefore, it is also important for the general strategies to gather similar triangles together into the same nodes.

MeanSplit

This general splitting strategy finds the mean position of all triangles, and splits in this position using a splitting plane. The plane is oriented approximately in such a way as to split the objects longest side. The orientation is defined by a vector that is constructed using the mean covariance of the x-, y- and z-values of the vertices in all triangles. Figure 7 shows how a splitting plane can be positioned and oriented. After splitting the object into two smaller objects, RSSs are constructed and MeanSplit is called recursively for each object. When the number of triangles in an object is small enough, a leaf splitting strategy is called instead.

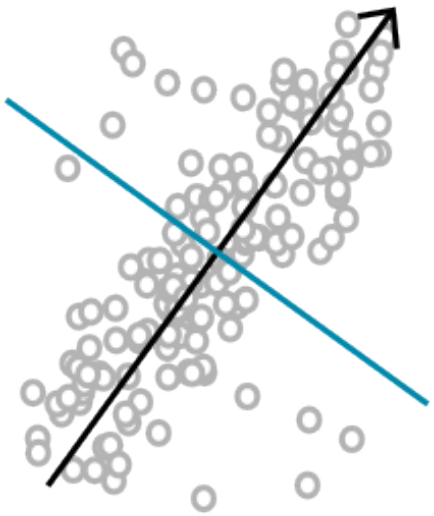


Figure 7: The dots represent the positions of triangles. The black arrow is a vector that is defined by the covariance of the triangles' positions, and roughly shows the object's "longest side". The blue splitting plane is placed in the middle of the triangles, and is aligned according to the arrow.

The first motivation for this splitting strategy is that splitting the object in the middle will produce two new objects of roughly equal size, which keeps the scale factor low. The second motivation is that since the split is performed through the longest side, the child objects will be elongated as little as possible, and thus have a smaller aspect ratio. The third motivation is that BVH construction time is kept relatively low by only selecting a single splitting point and orientation of the splitting plane, instead of testing

multiple samples for splitting positions and orientations, which would require more computations. The remainder of this section describes the MeanSplit strategy in greater detail.

Each triangle consists of three vertices, each of which is a vector of three values that together define a point in 3D space. The covariance of the x-, y- and z-dimensions of a vertex can be described using a $3x3$ covariance matrix, where each entry (i, j) is equal to the covariance between dimensions i and j. The covariance of two dimensions i and j for a single vertex V is calculated as:

$$cov(V, i, j) = V_i V_j - M_i M_j$$

Here, V_x denotes the value of dimension x in vertex V, and M_x denotes the mean value of dimension x among all vertices in all triangles. The mean covariance of two dimensions is simply the average of the covariances of those dimensions among all vertices.

When the mean covariance matrix has been constructed, another $3x3$ matrix which this thesis refers to as the *axis matrix* is built. Every column of the matrix defines an axis that is a linear combination of x, y and z. The first axis defines the direction where the mean covariance of x, y and z is the greatest, *i.e.* the direction that best describes the longest side of the object. The second axis is always perpendicular to the first axis, and defines the direction with greatest xyz-covariance that fulfills this constraint. The third axis is perpendicular to the first two. In MeanSplit, only the first axis is used as the orientation of the splitting plane. The second and third axes are ignored, but they are valuable for other strategies such as AxisSplit, which is described below.

AxisSplit

Intuitively, using MeanSplit for splitting objects through their longest sides should help reducing their elongatedness and aspect ratios. As mentioned in section 4.2.5, reducing elongatedness may not always lead to tightly fitting BVs, however, so it could be worthwhile to try additional ways of splitting in order to produce more efficient BVHs. Figure 8 shows a case where splitting through an object's longest side is clearly not the optimal solution. AxisSplit is a general splitting strategy that samples multiple ways of splitting the triangles in an object, and uses a simplified surface area heuristic to try to select the best candidate for maximizing the efficiency of the final BVH.

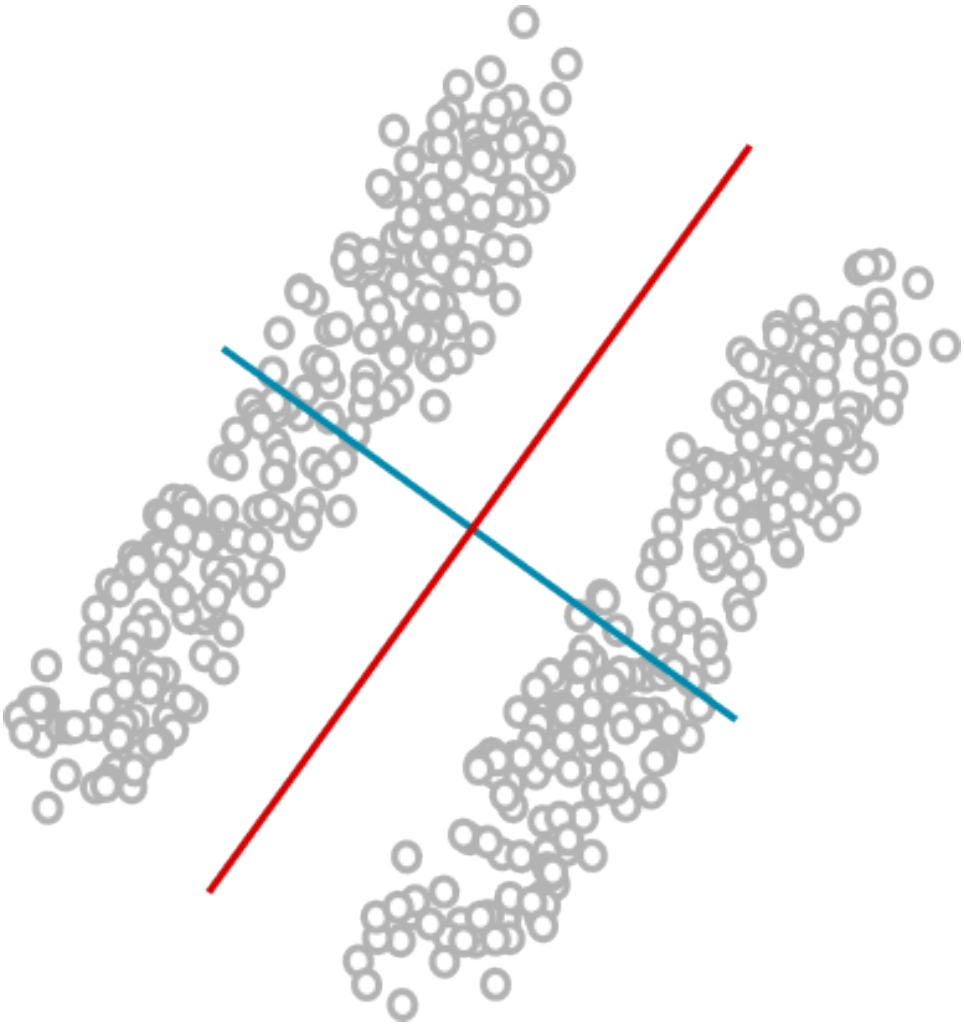


Figure 8: Splitting along the object's longest side using the blue splitting plane will produce less elongated children than using the red plane, but their BVs will not fit as tightly. In this case, MeanSplit is a very bad splitting strategy.

AxisSplit is based on and resembles MeanSplit in most ways, but sacrifices some construction speed for tighter and more efficient BVHs. Both strategies use splitting planes that are placed in the middle of objects, but while MeanSplit always splits the objects through their longest sides, AxisSplit samples multiple orientations of the splitting plane and imitates SAH by selecting the one that would minimize the combined surface areas of the RSSs of the child objects.

The previous section described MeanSplit, as well as the composition of an axis matrix whose three columns represent axes that can be used as orientations for splitting planes. The first axis describes the longest side of

the object to be split, and the other two axes are perpendicular to the first. AxisSplit places a splitting plane in the middle of the object, and samples three different splits with the plane facing the directions defined by the three axes from the matrix. RSSs are created for the child nodes produced by the splits, and their surface areas are calculated. The split that produced the RSSs whose sum of surface areas are the lowest is then chosen as the final split. AxisSplit is called recursively for each child until their numbers of triangles are small enough to be processed by a leaf level strategy.

ChainSplit

ChainSplit is a leaf level splitting strategy, and is therefore responsible for determining whether a small set of triangles are close enough to each other to be left in the same leaf node. The basic principle of ChainSplit is that triangles are close enough if they share vertices with each other. If two triangles share a vertex, they belong to the same chain, and are put in the same leaf node. If there are two or more chains within the set of input triangles, the triangles from the first chain are split away and put into their own leaf. ChainSplit is then called recursively for the remaining chains. Figure 9 shows an example of how a set of triangles are split by ChainSplit.

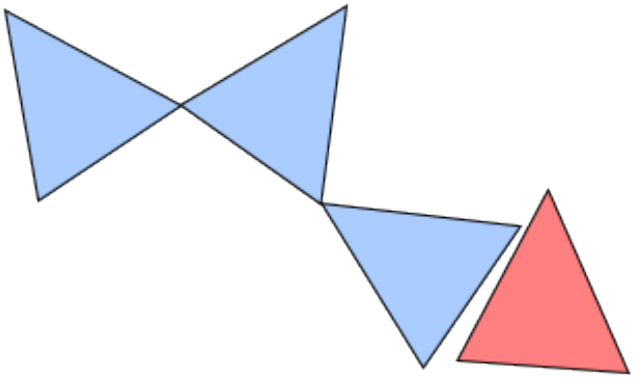


Figure 9: ChainSplit determines that there are two chains within this set of four triangles. The blue triangles form a chain by sharing vertices with each other. Even though the red triangle is very close to the rightmost blue triangle, they do not share vertices, and are put in different chains and different leaf nodes.

EdgeSplit

EdgeSplit is very similar to ChainSplit, and also works by separating triangles into different chains. According to the EdgeSplit heuristics, two triangles belong to the same chain if they share an edge, e.g. two vertices. This

means that EdgeSplit produces many more, smaller chains than ChainSplit, but the triangles that are put into the same EdgeSplit chain are always at least as close, and most often closer to each other than those put into a ChainSplit chain. Figure 10 shows a case where EdgeSplit will make different decisions than ChainSplit.

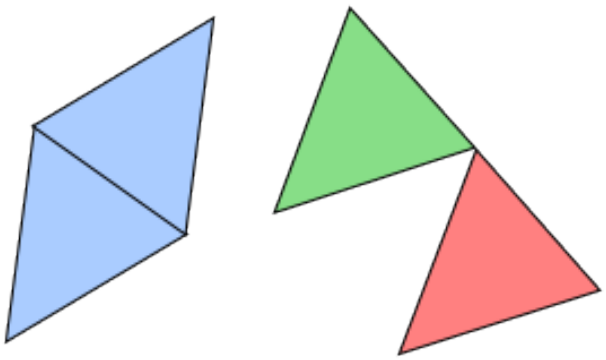


Figure 10: The blue triangles share two vertices, and are put into the same chain by both ChainSplit and EdgeSplit. The green and red triangles share one vertex, and are put into the same chain by ChainSplit. Since they only share one vertex, they are split into different chains by EdgeSplit.

NoSplit

NoSplit is an extreme leaf level splitting strategy. It never splits its input triangles, and always puts them in the same leaf node. For any given general strategy run beforehand, NoSplit produces the mimimum number of leaf nodes, all with the maximum number of triangles. Note that this does not mean that all leaves will contain four triangles. See Figure 11 for an illustration of a case where the leaves are created with two or three triangles.

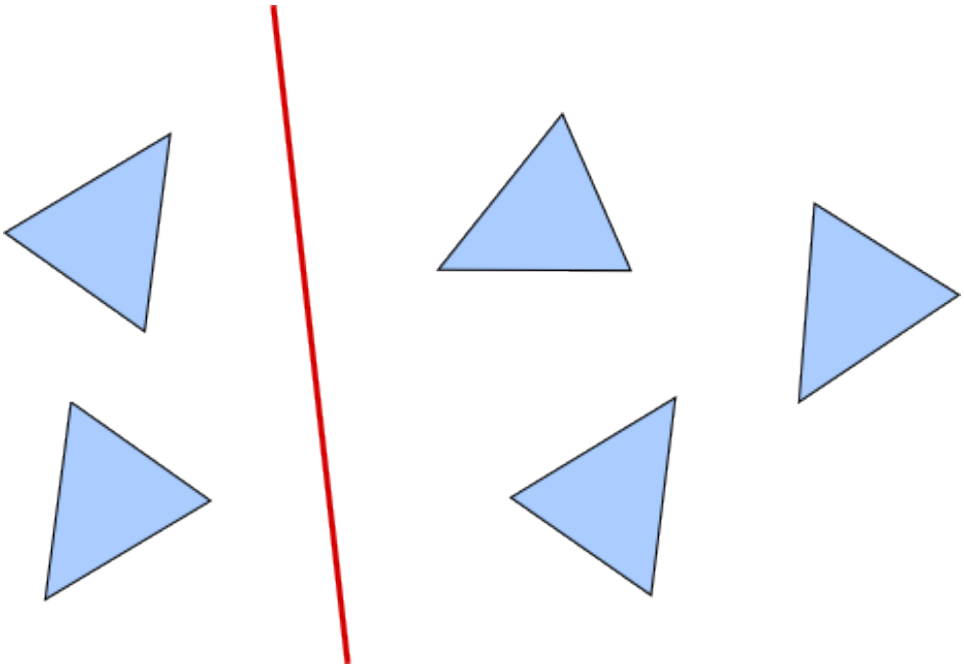


Figure 11: A set of triangles are to be split. Since the number of triangles are greater than four, a general splitting strategy is used. After splitting using the red splitting plane, the child nodes contain two and three triangles, respectively. NoSplit is then run for each of the two children. None of the triangles in the children are split, and the nodes are left as leaf nodes.

AllSplit

AllSplit is another extreme leaf level splitting strategy that is the complete opposite of NoSplit. The input triangles are always split, until they all have their own nodes, i.e. when the leaf nodes are all singular. AllSplit cannot decide how the splitting is to be performed on its own. Instead, it keeps calling the general splitting strategy to perform the actual splitting. For example, if MeanSplit is used as the general strategy, the triangles will continue to be split based on their mean positions and the directon of highest covariance, until only one triangle remains in each leaf. This corresponds to construction of BVHs without SIMD-utilization, where no leaf level strategies are necessary and the general strategies keep splitting until the leaves are singular.

5

Benchmarks and hardware

This section describes the benchmark suites and hardware used to test the implemented ISPC routines and splitting strategies used in BVH construction.

All benchmarks conducted in this thesis were performed on a 32 bit 2.66 GHz Intel Core 2 Duo machine with 2 GB RAM. The machine supports the SSE and SSE2 instruction sets, but not AVX.

Splitting strategy benchmark

The splitting strategies were tested by constructing BVHs of models and running collision- and distance queries between those BVHs. Proximity queries were conducted between two models at a time, one representing a dynamic object and the other representing an environment with one or more static objects. Each benchmark suite consists of the two models and a path defining a number of orientations and positions for the dynamic object.

The benchmarks worked by first constructing a BVH for each model and measuring construction times. The dynamic object was then translated and rotated according to the predefined path. One proximity test was made for each step along the path and the execution times of the queries were measured.

There are three sets of pairs of models that were used in the benchmarks: Cembox, LargeEnvironment and TunnelConsole. The models used in the benchmarks are proprietary and consist of 30 000–150 000 triangles for the moving objects and 300 000–1 700 000 triangles for the environment. Approximately 10% of the steps in the path of the Cembox and LargeEnvironment suites and 75% of the steps in TunnelConsole result in collisions between the objects. Figures 12, 13 and 14 show the objects in the TunnelConsole test suite.

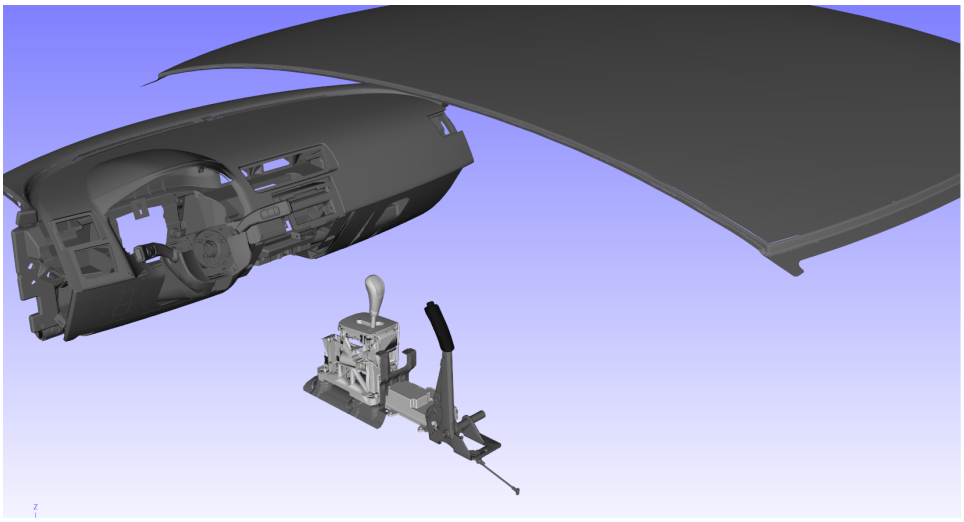


Figure 12: The static environment in TunnelConsole. The test suite models a car being assembled.

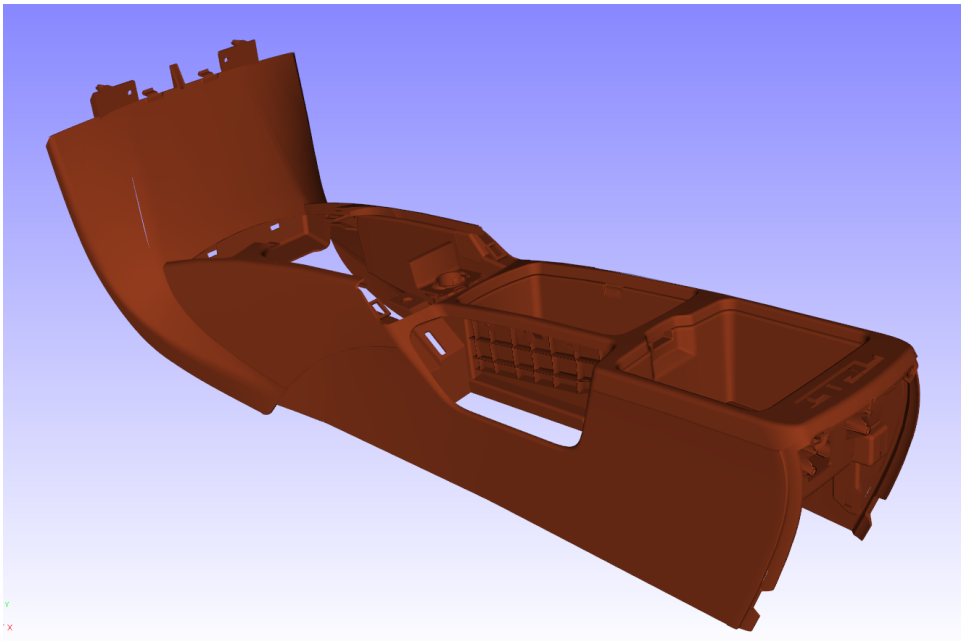


Figure 13: The moving, dynamic object in TunnelConsole.

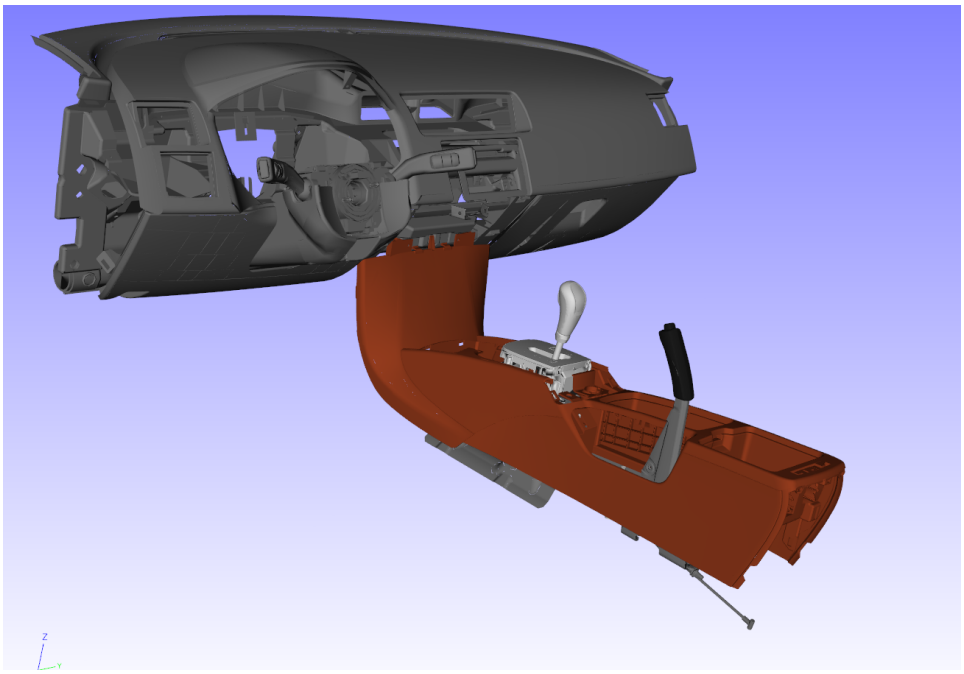


Figure 14: The dynamic object moves and rotates through the static environment according to a predefined path. Distance and collision tests are made between the objects at each step along the path. As can be seen in the picture, the moving part just barely fits around the static parts, and 75% of the steps of the path result in collisions.

In the benchmarks for splitting strategies, Embree routines were used for all primitives tests. Note that when two leaf nodes with only one triangle each are to be tested against each other, the Embree routines behave exactly like the C++ PQP routines, and runs the exact same piece of sequential code. When at least one of the leaf nodes to be tested contains two or more triangles, the Embree routines run SIMD-optimized code instead.

ISPC benchmark

The benchmarks of the ISPC routines used different test suites that worked by selecting two equally large subsets of triangles from a dynamic and static model, respectively, and testing all combinations of triangles from one subset against the other. Distance queries were benchmarked using two non-colliding objects, and collision queries with two colliding objects.

6

Results

This chapter describes the results obtained from the benchmarks, most notably the performance difference between sequential C++ code, Embree and ISPC of different libraries and techniques.

6.1 Using ISPC for collision and distance queries

The code of one collision routine and three distance routines was translated from the C++ PQP code to ISPC code. The collision routine is Triangle-Triangle, and the distance routines are Point-Triangle, Line-Line, and Triangle-Triangle. Point-Line and Line-Line tests are performed inside the Triangle-Triangle distance routine, and they are not benchmarked on their own. This section provides the results of benchmarking the ISPC Triangle-Triangle collision and distance routines against their Embree and original PQP counterparts.

The ISPC routines all work by taking two arrays A and B of primitives as input and testing all primitives from A against all primitives of B. The sizes of the input arrays can vary, but are capped at 1024 primitives. For example, this means that if two arrays consisting of 2000 triangles each were to be tested against each other, four calls would have to be made with $1024x1024$, $1024x976$, $976x1024$, and $976x976$ triangles as input. Figure 15 illustrates this process. The C++ routines always test two primitives, and the Embree routines always test two pairs of primitives against each other. Testing the arrays A and B using the regular C++ routines or Embree routines would therefore take 4 000 000 or 1 000 000 calls, respectively.

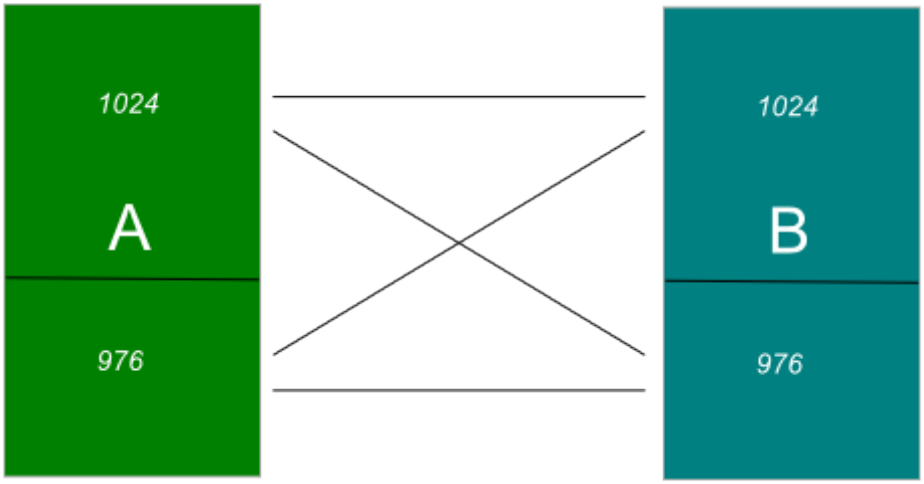


Figure 15: An ISPC test where 2000 triangles from A is tested against 2000 triangles from B. The test is performed in four chunks of up to 1024 triangle pairs at a time.

The execution times of the ISPC Triangle-Triangle collision- and distance routines were measured and compared against the execution times of the original C++ routines from PQP as well as those of the corresponding Embree routines. Unless otherwise noted, all ISPC routine calls were made using the input size 1024.

The results of the collision and distance benchmarks are presented in Tables 1 and 2, respectively.

	3000 x 3000 tris	5000 x 5000 tris	10 000 x 10 000 tris
C++ PQP	1.03 s	2.86 s	11.39 s
Embree	0.22 s	0.59 s	2.34 s
ISPC	0.09 s	0.27 s	1.06 s

Table 1: Results from the Triangle-Triangle collision detection benchmark tests, with ISPC input size 1024.

	3000 x 3000 tris	5000 x 5000 tris	10 000 x 10 000 tris
C++ PQP	1.74 s	4.74 s	19.17 s
Embree	1.06 s	2.94 s	11.64 s
ISPC	1.38 s	3.63 s	14.45 s

Table 2: Results from the Triangle-Triangle distance detection benchmark tests.

Additional tests were made with the ISPC Triangle-Triangle collision

and distance routines, where the input size was set to different values. A comparison of the results is shown in Tables 3 and 4. Of special interest are input sizes 1 and 2, since they are equivalent to the input sizes of the C++ and Embree routines, respectively. Note also that the ISPC routines can be recompiled for systems with AVX support in addition to SSE, which should give faster running times for input sizes greater than 2.

Input size	3000 x 3000 tris	5000 x 5000 tris	10 000 x 10 000 tris
1	1.95 s	5.42 s	21.67 s
2	0.77 s	2.08 s	8.36 s
4	0.34 s	0.95 s	3.83 s
8	0.20 s	0.58 s	2.28 s
16	0.14 s	0.41 s	1.59 s
128	0.11 s	0.28 s	1.09 s
1024	0.09 s	0.27 s	1.06 s

Table 3: Running time of the ISPC Triangle-Triangle collision test, with different input sizes.

Input size	3000 x 3000 tris	5000 x 5000 tris	10 000 x 10 000 tris
1	6.70 s	18.63 s	74.45 s
2	3.14 s	8.67 s	34.64 s
4	1.44 s	4.02 s	16.08 s
8	1.38 s	3.78 s	15.08 s
16	1.34 s	3.70 s	14.77 s
128	1.33 s	3.63 s	14.47 s
1024	1.33 s	3.63 s	14.45 s

Table 4: Running time of the ISPC Triangle-Triangle distance test, with different input sizes.

As can be seen in the tables, ISPC makes faster collision tests than PQP and Embree when it uses a large input size. When only a few triangles are tested at once, however, which is the case with input sizes 1, 2 and 4, Embree is faster. With input sizes 1 and 2, even the original PQP routines are faster than the SIMD-utilizing ISPC routines.

For distance tests, ISPC is faster than PQP but slower than Embree even with maximum input size. Once again, at input sizes 1 and 2, even PQP is faster than ISPC.

As shown in Table 2, Embree is always the fastest choice for distance tests, and 60–65% faster than regular PQP. Since increased distance query speed is the primary goal for this thesis, Embree should be considered the best alternative.

6.2 Performance of splitting strategies

This section shows the results of the splitting strategy benchmarks. Five different measurements were done during the benchmarking process. Three of them were gathered during the construction of the BVHs:

1. The number of triangles per leaf node of the BVHs
2. The total memory usage of the BVHs
3. BVH construction time

The last two measurements were made after the construction:

1. Number of collision queries per second
2. Number of distance queries per second

The results are split into two sections. In the first, the general strategies are compared to each other. In the second, the leaf level strategies are compared. Recall that there are three benchmark suites, two general splitting strategies, and four leaf level strategies. Most of the possible combinations of benchmark suites and splitting strategies are omitted in this thesis, and only the most notable and important results are shown. AxisSplit is the overall fastest general strategy, and ChainSplit is the fastest leaf level strategy for distance- and collision queries. Because distance query speed is prioritized in this thesis, the comparison of general strategies uses ChainSplit on leaf level, and the leaf level comparison uses AxisSplit on the general level.

Comparison of general level splitting strategies

Cembox with ChainSplit		
	MeanSplit	AxisSplit
Build time dynamic	0.23 s	0.39 s
Memory dynamic	2.5 MB	2.3 MB
Build time static	3.72 s	6.31 s
Memory static	41 MB	35 MB
Collision queries/s	1992.75	3841.68
Distance queries/s	205.03	595.13

Table 5: Comparison of general level splitting strategies for the Cembox benchmarking suite. ChainSplit was used at leaf level. The table shows build times and memory usage for the static and dynamic objects, as well as the proximity query speed of the benchmarking.

TunnelConsole with ChainSplit		
	MeanSplit	AxisSplit
Build time dynamic	0.56 s	0.92 s
Memory dynamic	6.7 MB	5.7 MB
Build time static	2.42 s	4.05 s
Memory static	28 MB	24 MB
Collision queries/s	1229.61	2240.00
Distance queries/s	229.72	358.45

Table 6: Comparison of general level splitting strategies for the TunnelConsole benchmarking suite.

LargeEnvironment with ChainSplit		
	MeanSplit	AxisSplit
Build time dynamic	0.97 s	1.61 s
Memory dynamic	9.5 MB	8.6 MB
Build time static	14.61 s	25.97 s
Memory static	150 MB	128 MB
Collision queries/s	3004.93	16223.40
Distance queries/s	18.07	58.26

Table 7: Comparison of general level splitting strategies for the LargeEnvironment benchmarking suite.

Tables 5, 6 and 7 show that there is a notable difference in construction time between MeanSplit and AxisSplit. The additional cost of sampling different ways of splitting triangles, and calculating RSS surface areas of potential new child nodes when making the splitting decision, seems to increase the construction time by approximately 60–70%. AxisSplit is, however, about 10% more memory efficient, potentially because it produces better balanced trees with fewer nodes, or because it allows ChainSplit to pack more triangles together at leaf level. AxisSplit's collision- and distance query performance is also much better than that of MeanSplit. The speedup of collision queries is between 80–400% in all three benchmarks, and distance queries are 50–220% faster.

Comparison of leaf level splitting strategies

TunnelConsole with AxisSplit				
	NoSplit	ChainSplit	EdgeSplit	AllSplit
1-tri nodes dynamic	1104	7828	20136	73425
2-tri nodes dynamic	6248	9089	10066	0
3-tri nodes dynamic	9871	8657	6479	0
4-tri nodes dynamic	7553	5362	3430	0
1-tri nodes static	5469	36310	88144	300401
2-tri nodes static	25513	38221	40539	0
3-tri nodes static	41062	35639	26613	0
4-tri nodes static	30180	20183	12835	0

Table 8: Comparison of the final number of leaf nodes with one, two, three, and four triangles for the leaf level splitting strategies, when used on the dynamic and static objects in the TunnelConsole benchmarking suite. AxisSplit was used as the general level splitting strategy.

As expected, table 8 shows that AllSplit only produces singular leaf nodes, and cannot utilize SIMD at all. NoSplit has an average of ≈3 triangles per leaf, which forces a large number of leaf leaf proximity tests to run more than one SIMD primitives tests. The reason for this is that the SSE-based routines in this thesis can only handle tests involving a maximum of 1vs4 or 2vs2 triangles in one step. If AVX or another technique with wider SIMD registers are used, NoSplit might perform better. ChainSplit has an average of about 2.3–2.5, and EdgeSplit about 1.8 triangles per leaf.

Cembox with AxisSplit				
	NoSplit	ChainSplit	EdgeSplit	AllSplit
Build time dynamic	0.39 s	0.39 s	0.42 s	0.47 s
Memory dynamic	2.2 MB	2.3 MB	3.1 MB	5.0 MB
Build time static	6.17 s	6.31 s	6.42 s	7.28 s
Memory static	30 MB	35 MB	44 MB	70 MB
Collision queries/s	3841.68	3841.68	3841.68	3841.68
Distance queries/s	563.14	595.13	578.44	528.00

Table 9: Comparison of leaf level splitting strategies for the Cembox benchmarking suite. AxisSplit was used at general level.

TunnelConsole with AxisSplit				
	NoSplit	ChainSplit	EdgeSplit	AllSplit
Build time dynamic	0.88 s	0.92 s	0.91 s	1.093 s
Memory dynamic	4.9 MB	5.7 MB	7.0 MB	11.4 MB
Build time static	3.94 s	4.05 s	4.13 s	4.75 s
Memory static	20 MB	24 MB	29 MB	47 MB
Collision queries/s	2240.00	2240.00	2199.78	2125.81
Distance queries/s	335.38	358.45	338.98	282.58

Table 10: Comparison of leaf level splitting strategies for the TunnelConsole benchmarking suite.

LargeEnvironment with AxisSplit				
	NoSplit	ChainSplit	EdgeSplit	AllSplit
Build time dynamic	1.53 s	1.61 s	1.63 s	1.91 s
Memory dynamic	7.9 MB	8.6 MB	10.4 MB	19.1 MB
Build time static	24.97 s	25.97 s	25.83 s	29.80 s
Memory static	111 MB	128 MB	160 MB	265 MB
Collision queries/s	3150.83	16223.4	16310.20	16310.20
Distance queries/s	35.80	58.26	37.89	38.66

Table 11: Comparison of leaf level splitting strategies for the LargeEnvironment benchmarking suite.

Tables 9, 10 and 11 indicate that compared to splitting at the general level, leaf splitting takes less time in general. Changing from no splitting at all to the maximum amount of splitting possible at leaf level by switching from NoSplit to AllSplit increases total construction time by between 15–30% only. The amount of memory that can be saved by packing triangles together into a smaller number of leaf nodes can be very high when any leaf level strategy except AllSplit is used. NoSplit only uses between 41–44% of the memory that AllSplit requires. ChainSplit seems to be the overall best leaf level strategy, with memory efficiency close to that of NoSplit, the fastest distance tests in all test cases, and only marginally slower collision tests in the LargeEnvironment benchmark. Switching from AllSplit, which is equivalent to using traditional, sequential BVHs, to ChainSplit increases distance test performance by 12–50%.

7

Discussion

This chapter discusses the results of the thesis, and compares them to some other previous work in the field. Conclusions are drawn regarding the usability of ISPC, the possibility of improving proximity tests using SIMD, and the efficiency of the developed splitting strategies. Finally, some ideas for extending the work of the thesis and possible research that can be conducted in similar areas are presented.

7.1 Analysis of ISPC and ISPC-based proximity routines

ISPC works well for both collision- and distance tests when a large number of triangles from one object are to be tested simultaneously against a large number of triangles from another object. However, that does not happen in practice, since the BVH prunes away the unnecessary tests so that only a very small number of triangles are tested at any one time. Because there are at most four triangles in each leaf node when using an SSE-based BVH implementation, the ISPC routines can only work with input sizes between one and four. As was shown in chapter 5, ISPC's distance tests are only slightly faster than the original PQP routines when the input size is four, and slower for smaller input sizes. Meanwhile, Embree is always faster than both the PQP and ISPC routines and is clearly the superior choice when fast distance tests is the primary goal.

For collision tests, ISPC is faster than PQP for input sizes two and up. If the BVH construction strategy is good enough, so that most leaf nodes contain at least two triangles, ISPC should be faster than PQP in most cases, and could therefore be a reasonable way of gaining a slight speedup. There is also potential for ISPC to be more efficient for collision detection involving primitives other than triangles, such as lines and rays.

Compared to assembly- and intrinsics programming, ISPC code is extremely simple to write, and very similar to basic C/C++. The ISPC proximity routines developed in this thesis were all quickly and easily translated from the original PQP code, with only minor modifications to the way that data was stored. The only potential difficulty that users of ISPC must face when porting algorithms from C++ is to align their data into contiguous blocks of memory to allow the system to move data using vector load and

store instructions instead of so-called gathers and scatters.

It is relatively easy to configure an ISPC kernel for different architectures. The ISPC routines used in this thesis were developed on an SSE-based 32-bit system running Windows, but were easily adjusted to work on a 64-bit Linux machine with AVX by changing the ISPC compiler's command-line arguments.

As was mentioned in chapter 3, Smith's master thesis also investigated the usability and performance of ISPC, but in the environment of 3D applications that are typically run on large sets of data using GPUs [7]. In the results of that thesis, ISPC performed almost exactly as well as intrinsics code for all applications tested. This thesis shows that under certain circumstances, ISPC performance can be worse than that of code based on intrinsics, particularly when the input data set is very small. ISPC seems to be better suited for applications where a greater amount of data needs to be processed in parallel.

7.2 Analysis of BVH construction

Optimizing BVHs with SIMD for faster proximity queries seems to work reasonably well. While collision detection does not seem to benefit much from SIMD, distance tests can be accelerated by up to 50%. Note, however, that other forms of optimization, such as those described in section 3, may be even more beneficial than SIMD for accelerating proximity queries.

A very promising feature of SIMD-based BVHs is the amount of memory that can be saved. BVHs built using NoSplit used almost 60% less memory than the equivalent sequential BVHs built by AllSplit. Even ChainSplit can save 50% memory or even more in some cases. In applications where memory-efficiency is important, SIMD optimized BVHs seem like perfect choices, and can even provide the additional benefit of a slight speed boost.

The popular method of sampling splitting locations and using the surface area heuristic seems to work well for constructing SIMD optimized BVHs, as was demonstrated by AxisSplit. The idea of making BVs as tight-fitting and as small as possible naturally causes the enclosed triangles to be grouped into small, tight shapes. As was mentioned in section 4.2.7, an important requirement of general level strategies is to be able to make groups of triangles that are close to each other, so that the leaf level strategies can avoid splitting them. It may be possible to create a general level strategy that is so good at grouping similar triangles together that no leaf level strategy is needed at all. Neither MeanSplit nor AxisSplit are good enough for this purpose, though, since ChainSplit clearly outperformed AllSplit in the benchmarks.

7.3 Summary and conclusion

This thesis investigated how CPU-based SIMD can be used to enhance performance of proximity tests between BVHs. It was shown that Intel's SPMD compiler has great potential for improving computations involving lots of data, but that more specialized, lower-level libraries such as Embree work better when the amount of data is smaller, which is typically the case in BVHs, where only a few triangles are tested for minimum distance or collisions at once.

In this study, it was shown that SIMD can only be used effectively in triangle-triangle proximity tests if the BVHs are constructed in certain ways. The right balance must be struck between splitting the triangles as much as possible to increase the BVH's innate ability of pruning away unnecessary tests, and grouping them together to increase the number of proximity tests that can be performed in parallel using SIMD.

Many of the same principles that are used to produce good BVHs for sequential proximity tests can also be applied when building SIMD-optimized hierarchies. The best general level splitting strategy investigated in this thesis uses a simplified surface area heuristic to determine which one of a number of sample splitting locations will produce the most efficient child nodes.

Besides improving the speed of distance queries by up to 50%, the SIMD-optimized BVHs that were presented in this thesis have another important property. When leaf nodes are allowed to contain more than one triangle, a smaller number of leaf nodes are needed overall. By not splitting nodes with four or less triangles, the memory consumption of the BVHs in this thesis could be lowered by up to 60%.

7.4 Future research

SIMD is a form of optimization and parallelism that can be relatively easily combined with other optimizations. There may therefore be ways to add SIMD-support for other BVH constrution methods than the basic ones used in this thesis. Some of the principles used in the construction strategies developed in the thesis are far from state-of-the-art, so it is likely that there are other strategies, such as real SAH-based construction, that can benefit from SIMD and improve performance even further. Since tightly grouped triangles are a requirement for maximum SIMD utilization, tighter, more well constructed BVHs should benefit more from SIMD support than badly constructed ones. There seems to be great potential for the absolute best among BVH construction methods to become even better by combining them with SIMD.

There are a multitude of algorithms for BVH construction out there, and only a handful of them were investigated extensively in this thesis.

SIMD-BVHs have slightly different properties than traditional BVHs, so there might also exist algorithms that do not work very well for BVHs in general, but that work extremely well for the SIMD optimized variants. Further investigation of splitting strategies designed specifically for SIMD-BVHs could be an interesting research subject.

There is a clear lack of scientific theory regarding distance tests, when compared to the vast amount of research that has been conducted for ray tracing and collision detection. Larsen et. al. tried to unify the two fields by reasoning about the equivalence of RSSs-based collision- and distance tests [4]. It may be possible to unify the subjects even further.

The BVHs used in this thesis were all based on SSE. This placed a natural limit on how many triangles could be packed into the same leaf before performance started to drop. By using AVX or other technologies where the SIMD-registers can hold a larger amount of data, it should be possible to pack triangles even more tightly to increase performance further.

References

[1] T. Ruipu, Z. Wei, and L. Jing, "The Study of Parallel Collision Detection Algorithms," *2010 International Conference on Multimedia Technology*, pp. 1–4, Oct. 2010.

[2] M. Tang, D. Manocha, J. Lin, and R. Tong, "Collision-Streams: Fast GPU-based Collision Detection for Deformable Models," *Symposium on Interactive 3D Graphics and Games*, vol. 1, no. 212, pp. 63–70, 2011.

[3] Intel Corporation, "Intel SPMD Program Compiler." `http://ispc.github.io/`, 2011. [Online; accessed 23-August-2013].

[4] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast Proximity Queries with Swept Sphere Volumes," tech. rep., Department of Computer Science, UNC Chapel Hill, 1999.

[5] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sept. 1972.

[6] NVIDIA, "What is CUDA." `https://developer.nvidia.com/what-cuda`, 2013. [Online; accessed 23-August-2013].

[7] P. Smith, "Using the CPU to Improve Performance in 3D Applications," Master's thesis, Bournemouth University, 2011.

[8] C. Tang, S. Li, and G. Wang, "Fast Continuous Collision Detection using Parallel Filter in Subspace," *Symposium on Interactive 3D Graphics and Games*, vol. 1, no. 212, pp. 71–80, 2011.

[9] I. Wald, "On fast Construction of SAH-based Bounding Volume Hierarchies," *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 33–40, Sept. 2007.

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

[11] S. Woop, "Embree: Photo-Realistic Ray Tracing Kernels." `http://software.intel.com/en-us/articles/embree-highly-optimized-visibility-algorithms-for-monte-carlo-ray-tracing`, 2012. [Online; accessed 23-August-2013].

[12] GAMMA research group, Department of Computer Science, UNC Chapel Hill, "PQP - A Proximity Query Package." `http://gamma.cs.unc.edu/SSV/purpose.html`, 1999. [Online; accessed 23-August-2013].

[13] G. Zachmann, "Minimal hierarchical collision detection," *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '02*, p. 121, 2002.

[14] S. Suri, P. M. Hubbard, and J. F. Hughes, "Analyzing Bounding Boxes for Object Intersection," *ACM Transactions on Graphics (TOG)*, vol. 18, no. 3, pp. 257–277, 2000.

[15] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.

[16] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Experiences with Streaming Construction of SAH KD-Trees," *2006 IEEE Symposium on Interactive Ray Tracing*, vol. V, pp. 89–94, Sept. 2006.

[17] W. Hunt, W. R. Mark, and G. Stoll, "Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic," *2006 IEEE Symposium on Interactive Ray Tracing*, pp. 81–88, Sept. 2006.