BVH Split Strategies for Fast Distance Queries

Robin Ytterlid Evan Shellshear
FCC

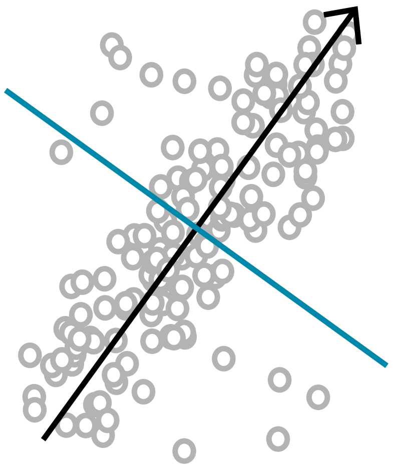

Figure 1. Splitting in the direction of highest covariance

Abstract

This paper presents a number of strategies focused on improving bounding volume hierarchies (BVHs) to accelerate distance queries. We present two classes of BVH split strategies that are designed specifically for fast distance queries, but also work well for collision detection and we compare their performance to existing strategies as well as the new ones introduced here. The classes of split strategies improve distance query performance by constructing a BVH to provide better SSE-utilization during proximity queries and also improve the quality of bounding volumes of the BVH. When combined with each other, our two approaches can offer up to five times better distance query performance and twice as fast collision queries than standard split methods.

1. Introduction

The widespread use of simulation and animation in industry, such as in virtual manufacturing or computer-aided design, requires software developers to create increasingly efficient and powerful mechanisms for handling the interaction of complex-shaped, deformable objects with each other. One of the major bottlenecks in such simulations is the computation of minimal distance (henceforth simply distance) between dynamic and deforming meshes. The distance computation between meshes forms the bottleneck of many simulations and plays a fundamental role in fields such as path planning, [Segeborn et al. 2010; Spensieri et al. 2010; Hermansson et al. 2013; Spensieri et al. 2008], virtual simulations [Oshita and Makinouchi 2001], virtual prototyping [Lin and Manocha 2003], just to name a few.

One of the most prominent approaches to measuring distances between polygon meshes is to construct Bounding-Volume Hierarchies (BVHs) to approximate the shape of each mesh at iteratively finer levels [Larsen et al. 1999]. This results in hierarchies of Bounding Volumes (BVs) where the root node encloses all triangles in a mesh and every leaf node usually encloses just a single triangle. For distance queries, tight fitting BVs are essential because unlike collision detection, the hierarchies need to be traversed to the leaf level to find the two closest triangles and traversal cannot be stopped once bounding volumes are no longer intersecting. It has been demonstrated that the choice of tight fitting BVs such as rectangle swept spheres (RSSs) can provide better performance than simpler BVs such as axis-aligned bounding boxes (AABBs), [Larsen et al. 1999; Lauterbach et al. 2010] and so we use RSSs as our BVs in this article (see Larsen et al. [1999] for an in-depth description of RSSs).

In this article we focus on two classes of split strategies for optimal BVH construction with the goal of improving the performance of distance queries. The first class of split strategies borrows ideas from work done in ray tracing, [MacDonald and Booth 1990], and examines the use of quality measures on the BVs constructed in a BVH. The second class of split strategies looks at utilizing SSE, [Thakkur and Huff 1999], to improve the final triangle-triangle queries and utilizes the primitive tests presented in Shellshear and Ytterlid [2014].

The rest of the paper is structured as follows. In the following section we quickly review research directly related to our goals. In Section 3 we then describe our new split strategies for BVH construction. In Section 4, we discuss our other BVH construction and traversal choices. In Section 5.1 we present our experimental setup and benchmark cases. In Section 5.2 we then present our results and in the final section we conclude.

2. Related Work

There exist significant volumes of work done on BVHs for proximity queries. Most of this work has focused on collision detection, however, our main focus here is on fast distance queries. Larsen et al. [1999] introduced algorithms for creating BVHs that are faster for distance queries than traditional BVHs. They use a family of BVs called Rectangular Swept Sphere (RSS) Volumes, which they demonstrate are up to four times faster than other bounding volume types for distance queries.

When it comes to empirical results on the construction of BVHs for proximity queries, little work has been done on investigating the build strategies for constructing BVHs with or without SIMD. Exceptions include Klosowski et al. [1998] and Zachmann [2001], however, no positive results are presented for build strategies for BVHs apart from the usual top-down approach of subdividing sets of primitives by splitting in the direction of the principal axis. When looking just at results concerning split strategies, Klosowski et al. [1998] chose to split to minimize BV volumes as they intuitively believed that this would result in better BVs, although they did not back up their claims with any comparative tests.

For SIMD strategies the paper by Min et al. [2009] can be given as an example that addressed the use of SIMD for BVHs, although this article looked at other aspects of BVH construction than those we consider here. Their focus is on a SIMD friendly k-dop while we focus more on packing multiple triangles into a BVH's leaf and exploiting SIMD for the triangle distance and collision tests. Our approach has the advantage of not only faster proximity queries between primitives but also smaller BVHs and hence a smaller memory footprint for a given BVH.

In spite of the lack of research on BVH construction strategies for proximity queries, in the field of ray tracing, the use of quality measures for building BVHs is a well-known area of research [Wald et al. 2007; Müller and Fellner 1999; Jeffrey 2005]. The measures used in ray tracing are based on building a BVH are based on other assumptions or (such as assuming that a ray traverses every node that the ray intersects) one has a different focus such as in Karras et al. [2013] (fast parallel construction). Here we are interested in high quality BVHs (with construction time being less important) and we attempt to adapt the ray-tracing measures, where appropriate, for BVHs used for proximity queries.

3. Splitting Strategies for BVH Construction

The purpose of this section is to discuss some important properties of BVs and BVHs that impact the performance of distance queries. We analyze two types of strategies to improve the construction of BVs in the BVH. The first is used for improving the quality of BVs in the upper levels of the BVH. The second type is also adjusted for an optimal construction of the BVH leaves for SIMD optimizations. For optimal SSE based SIMD utilization as described in Shellshear and Ytterlid [2014], the requirement is that we make four triangle-triangle tests simultaneously. To do this, we need to have sets of two or four triangles in each BVH leaf. A balanced splitting strategy may seem best for this purpose, however, it has been demonstrated that splitting a set of triangles in the mean of the barycenters of all triangles gives far superior BVH performance compared to all other tested options, including splitting in the median to create balanced trees [Klosowski et al. 1998; Held et al. 1995]. Our own tests also support this, although our results are not presented here. This is why we have chosen to investigate a middle-ground approach where a traditional build strategy is used for the upper levels of the BVH, but where we switch to a different, specialized heuristic for optimizing SSE utilization when we reach the leaf level in the construction.

Because our work builds on the work of Larsen et al. [1999], it is worthwhile to describe briefly the algorithms and structures presented there. In the aforementioned paper the authors introduce a BVH built with RSSs. RSS are defined as the Minkowski sum of a rectangle and a sphere, although the rectangle could merely be a line or a point (leading to lozenges or spheres). In [Larsen et al. 1999] the BVH is constructed by splitting the set of triangles in the direction of the principle axis in the mean and using the triangle's barycenters to decide which side of the split a given triangle belongs to. This is continued down to the leaf level where a single triangle is stored in a leaf.

We present and compare four classes of splitting strategies in total. The first two are introduced in Section 3.2, and they contain no special leaf level heuristics. The other two are introduced in Section 3.4 and are identical to the first two, except they use specialized heuristics for the leaf level splitting. More in-depth introductions and motivations for the two pairs of strategies are found in Sections 3.1 and 3.3, respectively.

3.1. Cost Functions for Upper-level BVH Splitting Strategies

For improving the construction of the upper levels of BVHs for faster proximity queries we borrow ideas from the ray tracing literature. In particular, we compare different formulas for minimizing the cost of traversal through the BVHs. We test numerous cost formulas, including one formula borrowed directly from ray tracing. The reason we include a formula directly from ray tracing is to investigate whether the same BVHs created for high-performance ray tracing can also be used effectively for proximity queries.

To develop effective cost formulas for building BVHs, we follow the arguments presented in Larsen et al. [1999], that the logic and arithmetic for RSS-based collision and distance queries are the same, and that the analytical properties of collision tests also hold for distance tests.

To produce a good BV quality measure, an idea might be to build BVs so as to minimize the chance that two BVs intersect each other in order to minimize the number of traversals. There exist well-known results from geometric probability that give the probability of two randomly moving convex sets intersecting each other (see Chapter 16 in Santaló [2004]). For two convex sets in three dimensions, K_1 and K_2, moving via random rigid motions, the measure of all positions of K_1 which intersect K_2 is (Equation 16.1 in Santaló [2004]),

$$8\pi^2(V(K_1) + V(K_2)) + 2\pi(SA(K_1)M(K_2) + SA(K_2)M(K_1)), \quad (1)$$

$V(K)$ stands for the volume, $SA(K)$ stands for the surface area and $M(K)$ stands for the integral of the mean curvature. Our goal is to adjust the BV parameters so as to minimize the value of Equation 1 for a pair of BVs K_1 and K_2 and hence reduce the measure of all positions where the two objects collide. A problem with using Equation 1 when building a BVH is that we only have information about the current BVH and not all possible other BVHs that we will test against. To deal with this, we assume that each BV will be tested for intersection (or distance) with an identical shaped BV (i.e. in our case, two RSSs with the same parameters). For the way we compute the distance or intersection between two BVHs, this assumption is a good approximation due to our traversal strategy which keeps the sizes of BVs being traversed as close as possible (see Section 4). Hence, given a BV, K, we measure the quality by the following equation,

$$16\pi^2 V(K) + 4\pi SA(K)M(K). \tag{2}$$

Equation 2 can be used to compute a global cost estimate of a BVH but, as in ray tracing [MacDonald and Booth 1990], this can be quite computationally expensive so we chose to solve this via a greedy top-down building strategy [MacDonald and Booth 1990].

When building RSSs, unlike AABBs, there exists no known way of computing an optimal RSS around a given set of triangles or points, [Bengtsson 2011]. In addition, there are $O(2^n)$ possible ways of partitioning n triangles into two disjoint sets. Hence, due to these limitations in being able to optimally compute Equation 2, we also chose to test other measures for the quality of the children BVs, K_1 and K_2, of a given BV K. Other measures we also chose to test were the volume and surface area of the children. The cost formulas C_{SA}, C_V, and C_{IM} of all three measures presented so far (Surface Area, Volume, and Intersection Measure) are given in the following three equations:

$$C_{SA}(K) = SA(K_1) + SA(K_2) \tag{3}$$

$$C_V(K) = V(K_1) + V(K_2) \tag{4}$$

$$C_{IM}(K) = 16\pi^2 V(K_1) + 4\pi SA(K_1)M(K_1) + 16\pi^2 V(K_2) + 4\pi SA(K_2)M(K_2) \tag{5}$$

For an RSS with radius r, width w and length l the surface area and volume can be calculated as:

$$SA(l,w,r) = 4\pi r^2 + 2lw + 2\pi rl + 2\pi rw, \tag{6}$$

$$V(l,w,r) = \frac{4}{3}\pi r^3 + 2lwr + \pi r^2 l + \pi r^2 w. \tag{7}$$

To compute the integral of the mean curvature, $M(l,w,r)$, we decompose an RSS into its different surface elements. An RSS has up to two flat faces with constant mean curvature $H = 0$; up to four half-cylinder sides, two with surface area πrl and two with surface area πrw all having a constant mean curvature $H = \frac{1}{r}$; and finally up to four quarter sphere corners each with surface area πr^2 and constant mean curvature $H = \frac{2}{r}$. Hence,

$$\begin{aligned} M(l,w,r) &= \frac{1}{2\pi}\left[\int_{faces} HdA + \int_{sides} HdA + \int_{corners} HdA\right] \\ &= \frac{1}{2\pi}\left[2(lw)0 + 2(\pi rl)\frac{1}{r} + 2(\pi rw)\frac{1}{r} + 4(\pi r^2)\frac{2}{r}\right] \\ &= 2\pi l + 2\pi w + 8\pi r \end{aligned}$$

The fourth and final cost function that we test is from ray tracing. It involves multiplying the surface area by the number of triangles and adding the costs of traversal and triangle-triangle distance tests, [Wald et al. 2007]. The cost function C_{RT} is presented in Equation 8, whereby T_{RSS} stands for the cost of RSS traversal, T_{TRI} stands for the cost of a triangle-triangle distance test and $N(K)$ stands for the number of triangles in a BV child K.

$$C_{RT}(K) = 2T_{RSS} + \frac{SA(K_1)}{SA(K)}N(K_1)T_{TRI} + \frac{SA(K_2)}{SA(K)}N(K_2)T_{TRI}. \qquad (8)$$

Note that the original ray tracing cost function is used to decide whether or not to split a given node, while our version decides how to do the actual splitting (to not split is never an option). Hence, in Equation 8, we do not include a cost to determine if the splitting should be stopped at an arbitrary level to create a leaf. Our tests indicated for all measures here (even when the number of triangles were factored into the equation in e.g. Equations 4 or 5) that creating a leaf at an arbitrary level resulted in worse performance (less distance queries per second). Our own tests show triangle tests to be approximately three times slower than RSS tests which could be the reason for these results — see Table 6 in Section 5.3 for more detailed results. This does not exclude there being a measure that could produce better results with an early leaf creation clause but we leave it for future research to investigate this possibility.

In addition to the measurements used in the four cost functions presented above, we conducted additional tests with measurements such as diameter; maximum side length; radius; ratios of side lengths, surface areas, and volumes; and many different combinations of measurements such as side length plus radius; surface area divided by volume and vice versa. None of these other measurements produced better results than any of the four cost functions above in any of our test cases, and so are not discussed or investigated further in this paper.

In addition to the above formulas one could argue for factoring in the number of triangles in each BV child into each measure. Note that this is done in the ray tracing formula, Equation 8, when compared to the surface area measure. One can see that for any split method, the parent BV has constant surface area, and the T_{TRI}

and T_{RSS} are also constant. Hence, for measures such as the ray tracing measure, the $SA(K_1)N(K_1)$ and the K_2 equivalents in Equation 8 are the only factors that determines which split is chosen. As can be seen from the results in Section 5.2, the addition of the number of triangles $N(K_1)$ and $N(K_2)$ in the measure did not improve the performance. We also tested all other measures similarly and this produced similar results (i.e. performance being worse than the original measure without the number of triangles factored in), hence we do not include them in the analysis below.

3.2. Upper-Level BVH Splitting Strategies

Given the quality measures in Section 3.1, we now present our first two classes of split strategies based on these measures. The first strategy that we considered we call MeanSplit , and it is the same as the original splitting method presented by Larsen et al. [1999]. It splits a node by using a splitting plane placed in the mean point of the triangles. The normal of the splitting plane points in the direction of highest covariance among triangle barycenters, shown in Figure 3.2. This choice of plane direction was shown in Klosowski et al. [1998] to produce the best results.

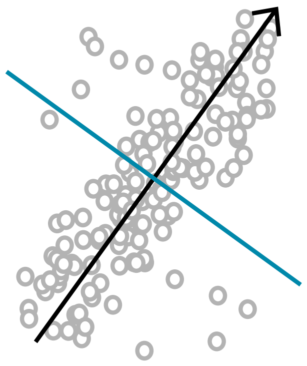

Figure 2. MeanSplit is the BVH construction strategy from Larsen et al. [1999]. It splits triangles using a splitting plane (blue line) at the triangles' mean position, facing the direction of highest covariance (black arrow) among triangle positions.

The second class of strategies is inspired by the cost function heuristics in ray tracing and comes in several different variants, where each variant uses a different cost function as presented in Section 3.1. Regardless of which variant is used, the strategy will be referred to as AxisSplit here . AxisSplit uses a sampling algorithm that tries three different ways of splitting a node, and then employs a heuristic based on the selected cost function to choose which of the sampled splits to use. The basis for our choice of three different ways is based on the success of current splitting strategies. It has been demonstrated that splitting along the direction of maximum variation in triangle barycenters produces the best results [Klosowski et al. 1998], hence we take this as our starting point. To then get as different splitting axes as possible, we then choose axes that are perpendicular to the first one in the direction of the second most variation (with the same reasoning for this choice as in the case with maximum variation). The third splitting direction is perpendicular to the other two and is fixed by the first two choices. For each sampled split, BVs for the resulting child nodes are created, and we compute the quality of these BVs with the cost function. The split that produced the children with the best quality is kept and the others are discarded. Figure 3 illustrates an example case where AxisSplit produces a different result than MeanSplit when using the surface area cost function C_{SA}.

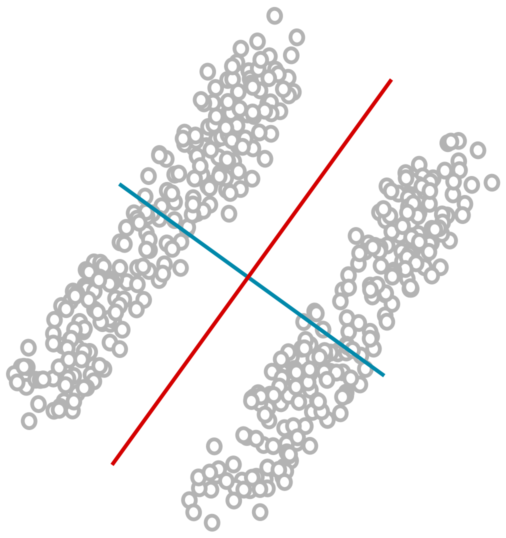

Figure 3. A case where AxisSplit splits on the red plane, resulting in BVs with less total area than if the blue plane used (e.g., would be done by MeanSplit).

As discussed in the beginning of Section 3.1, unlike in ray tracing, adjusting our strategies to split in a position other than the mean, or to sample different splitting positions (including the mean) did not produce better results than splitting at the mean only. We tested splitting at the median, first quartile, third quartile, one-third and two-thirds along the projection. None of these strategies produced better results than splitting at the mean. This confirms the results presented in Klosowski et al. [1998], and so we do not present the results here.

3.3. SSE-Optimized BVH Leaf Nodes

In this section we look at how to utilize SSE for tests between the triangles in the leaves of two BVHs. Using SSE in conjunction with BVHs for proximity tests between triangles might seem simple and intuitive. To be able to utilize SSE, the triangle tests need to be grouped together so that instead of testing one triangle against one other triangle at a time, we can test multiple triangle pairs at once. This can be done easily by constructing BVHs where leaf nodes contain more than one triangle, and then testing all triangles from one leaf against all triangles from another via SSE routines. The most straightforward way is to use a normal top-down splitting approach, such as the one in Section 3.2, but stop splitting when the number of triangles in a node is small enough. This approach can slightly reduce construction time and greatly reduce the number of nodes and thereby depth and memory footprints of the BVHs. Intuitively, this straightforward method has the potential to increase the performance of proximity queries, since we no longer have to traverse BVHs as deeply. This is only true under certain conditions, however, since the pruning ability of the BVHs is reduced when we don't split their nodes all the way. In essence, we remove the BV-BV tests of the lower BVH levels and replace them with additional triangle-triangle tests. As long as these extra triangle tests take less time than the removed BV tests, the overall performance should increase. To give the reader an idea of the relative execution time of BV-BV tests and triangle-triangle tests, Table 6 in Section 5.3 shows that a single triangle-triangle test is approximately three times slower than an RSS-RSS test, but Shellshear and Ytterlid [2014] show that SSE can improve the triangle-triangle test performance by two to seven times.

When we remove the lowest levels of a BVH to fit more triangles into leaf nodes, we lose some of the benefits of the BVH pruning. If a triangle in a leaf must be tested against a triangle in another leaf, all triangles from the first leaf are forced to be tested against all triangles from the second — these extra tests cannot be pruned since we are already at the leaf level. Consider the case where the two leaves contain four triangles each — if only one triangle pair from these leaves actually needs to be tested, we are forced to test the 15 additional pairs as well, which could lead to significant amounts of wasted computation power. An even worse case can occur when the child nodes would have been small enough to prune away all 16 triangle tests. To mitigate these effect, we should use a BVH construction strategy that can estimate whether it is more beneficial to keep a leaf node as it is to utilize SSE as much as possible, or to split it further to receive greater benefit from BVH pruning. Note that unless the triangles in a node are sufficiently similar — i.e. likely to be tested against the same set of other triangles during a query — the benefit of splitting them generally outweighs the benefit of keeping them together. An important implication of this is that a heuristic for splitting leaf nodes can produce the best results only when the triangles in an input node are already somewhat similar.

Another important factor effecting the performance of a BVH is the following. When a top-down BVH construction approach is used, allowing up to four triangles per leaf does not guarantee that all leaves will contain four triangles. A node with seven triangles can never be split in such a way as to produce leaf nodes that contain four triangles each, for instance. This affects performance, since some combinations of number of triangles in leaves are more efficient to test than others. If we use SSE, we can test up to four triangle pairs at once, which means that some computation power is wasted whenever we need to test three or fewer pairs. Furthermore, if we need to test five or more pairs, one SSE routine call is not enough, which leads to a significant performance drop. Table 1 shows the number of SSE routine calls needed to test any pair of leaf nodes with up to $4x4$ triangles. Note that only pairs of leaves with $1x4$ or $2x2$ triangles utilize SSE routines to its fullest. Note that if AVX was to be used instead of SSE, pairs of $1x8$ or $2x4$ triangles would be the most efficient instead.

	1	2	3	4
1	1	1	1	1
2		1	2	2
3			3	3
4				4

Table 1. Number of SSE routine calls required to test distance between X vs Y triangles.

3.4. BVH Splitting Strategies with Leaf-Level Heuristics for SSE

We now present the BVH construction heuristics that are focused on producing leaf nodes suitable for SSE. These heuristics can be seen as extensions of the first two strategies presented in Section 3.2. They are meant to be used in conjunction with the MeanSplit and AxisSplit strategies, by replacing the ordinary splitting heuristics when splitting nodes with four triangles or less. The leaf level splitting heuristics decide how to split nodes to produce a good balance between SSE utilization and BVH pruning ability. Several versions of leaf level heuristics (among numerous) that were tested are presented here.

NoSplit never splits, so all nodes with four or less triangles become leaf nodes. We get tightly packed triangles, but pruning ability suffers greatly.

AreaSplit makes a split in the same way as MeanSplit or AxisSplit (depending on the splitting strategy used to that point) and builds the children BVs and calculates their surface areas. If the combined BV surface area of the children is greater than the parent's BV surface area, we discard the split, and the parent becomes a leaf node. Otherwise we keep the split and recursively split the children.

VolumeSplit works as AreaSplit, but compares the parent's BV volume instead of surface area to its children.

ChainSplit groups triangles into chains. Two triangles are linked to each other if they share at least one vertex. If a triangle is linked to a number of other triangles, they all belong to the same chain. An unlinked triangle forms its own chain. ChainSplit splits away one chain at a time until each leaf node contains a single chain.

EdgeSplit works as ChainSplit, but triangles need to share an edge to be considered linked.

The performance of proximity queries in BVHs made using ChainSplit was significantly better than that of those made using other leaf level heuristics in every one of our test cases. Because of this, when using SSE for proximity queries between the primitives in the leaves we always used ChainSplit to split nodes with four triangles or less in them. In the rest of this paper we call the strategies that use MeanSplit and AxisSplit initially and then swap to ChainSplit for nodes with four triangles or less MeanSSE and AxisSSE .

4. Remaining BVH Construction and Traversal Choices

4.1. RSS Fitting and Construction

As we cautioned in Section 3.1, there is no known algorithm to optimally construct an RSS from a set of triangles or points. Instead we construct our RSSs via computing a minimum width slab based on the orientation computed from principal component analysis of the contained triangles. We then use the minimum width slab to find an RSS around the set of triangles. See Bengtsson [2011] for more details.

4.2. BVH Traversal

We traverse two BVHs via visiting the children BVs of the BV with the larger volume first (as long as it is not a leaf) until we reach a leaf at which point we keep visiting the children of the other BV until we reach a leaf. At this point we compute the minimum distance between the triangles in both leaves and remember the smallest triangle-triangle distance found so far. When we are about to traverse two nodes, we calculate the distance between their BVs, and compare it to the current minimum triangle-triangle distance. If the distance between the BVs is greater, we know that all the distances between triangles in those BVs are greater as well, and we prune this branch of the traversal. See Algorithm 1 for pseudocode.

Note that this method of traversal is in contrast to that presented in Larsen et al. [1999]. Again this was due to the fact that, in our tests, a priority based queue (using distances to determine the priority) produced slower distance queries than the simple method mentioned above. This was due to cache locality and the additional gain (in terms of a reduced number of tests) being less than the cost of the priority queue.

4.3. SSE-Optimized Triangle-Triangle Proximity Querries

While we focused in this paper on the split strategies for better SSE utilization, we note that the properties of the triangle-triangle test routines themselves can significantly affect the performance of the proximity queries. We used two sets of proximity

Algorithm 1 Pseudocode of the BVH traversal algorithm, $Traverse(Node1, Node2)$.

```
1: Input: Two nodes $M$ and $N$ of two different BVHs.
2: Output: The distance between the closest triangles.
3: Let $d_{MN}$ be the minimal distance between the nodes $M$ and $N$ and $d_{min}$ be the
   minimum triangle-triangle distance computed so far (equal to infinity initially).
4: if $d_{MN} < d_{min}$ then
5:    if $M$ is a leaf and $N$ is a leaf then
6:       Compute distance between triangles in $M$ and $N$ and update $d_{min}$ if neces-
         sary.
7:    else
8:       if $M$ is a leaf or $Volume(N) > Volume(M)$ then
9:          for Each child $n_i$ of $N$ do
10:            $Traverse(M, n_i)$
11:         end for
12:      else
13:         for Each child $m_i$ of $M$ do
14:            $Traverse(m_i, N)$
15:         end for
16:      end if
17:   end if
18: end if
```

query routines: one with SSE support (for MeanSSE and AxisSSE), and one without (for MeanSplit and AxisSplit). For the non-SSE version we used the triangle-triangle collision and distance routines present in the PQP package [2014]. For the SSE version we used those presented in Shellshear and Ytterlid [2014]. However, because we are not guaranteed to have four triangles in each leaf BV we also require an algorithm to take care of the variable number of triangles. The algorithm we used at the leaf level to compute the distance between triangles in two leaves is presented in Algorithm 2. The $SSETriangleDistance()$ function is presented in Shellshear and Ytterlid [2014] and we briefly describe it here.

The $SSETriangleDistance()$ function presented in Shellshear and Ytterlid [2014] algorithms utilized SSE4 instructions to compute four triangle distances simultaneously. In essence the algorithm was simply a direction translation from the triangle distance routine used by the PQP package, [Larsen et al. 1999]. Unexpectedly, the SSE version resulted in seven times faster distance routines.

Note that, in combination with certain splitting strategies, these SSE optimized tests led to the total query times being up to five times as fast as the standard splitting at the mean strategies (see e.g. Table 4 where the number of distance queries for

Algorithm 2 Computing the distance between two sets of triangles.

```
1: Input: Two sets of triangles $A$ and $B$ with up to four triangles in each set.
2: Output: The minimum distance between the sets of triangles.
3: if $|A| = 1$ and $|B| = 1$ then
4:     For $a \in A$ and $b \in B$ return $SingleTriangleDistance(a, b)$.
5: end if
6: if $|A| = 2$ and $|B| = 2$ then
7:     Let $a_1, a_2$ be the triangles in $A$ and $b_1, b_2$ be the triangles in $B$.
8:     Return $SSETriangleDistance(a_1, a_2, a_1, a_2, b_1, b_2, b_2, b_1)$
9: end if
10: Assume $|A| \geq |B|$ (the other case is handled analogously). Set $d_{min}$ to infinity.
11: for All triangles $b \in B$ do
12:     Let $a_1, a_2, a_3, a_4$ be the triangles in $A$. If there are not four triangles we repeat
        the final triangle enough times so there are four triangles.
13:     Let $d = SSETriangleDistance(b, b, b, b, a_1, a_2, a_3, a_4)$
14:     if $d < d_{min}$ then
15:         $d_{min} \leftarrow d$
16:     end if
17: end for
18: Return $d_{min}$.
```

AxisSSE C_{SA} is about five times more than MeanSplit). As was shown in Shellshear and Ytterlid [2014], a single SSE based triangle distance test is between two and seven times faster than four triangle-triangle distance tests. Given that triangle-triangle distance tests only constitute a fraction of the total computation (the rest being mainly BV-BV tests) done when traversing two BVHs, we can see that one cannot expect an automatic two to seven times speed-up as discussed previously. Table 6 in Section 5.3 shows an example of how triangle-triangle test and RSS-RSS test execution times relate to each other.

5. Evaluation

5.1. Benchmarks Used

To test the performance of the construction strategies mentioned in the previous two sections, we created a number of real-life benchmarks to test them on. Three benchmarks were used to test performance, build times and memory consumption of BVHs created by our construction strategies. All benchmarks simulate situations from automated car assembly and involve complex objects moving through tightly cluttered environments, see Figures 4, 5 and 6. All benchmarks were run on a 64 bit 2.67

GHz Intel Core i7 CPU (with four cores and hyperthreading) with 8 GB RAM under Windows 7 using Microsoft Visual Studio 2012 with full optimization. All data was tested using 4 byte floats meaning that it was possible to pack four floats into a single SSE SIMD register. SSE instructions up to SSE 4.2 were available for the processor to use.

The benchmarks all consist of one object moving around and sometimes colliding with a large immobile object. Both objects are rigid, and a BVH is only created once for each object at the start of every benchmark run. The smaller object moves according to a predefined path by being translated and rotated in discrete steps. At each step along the path, collision detection and distance queries are performed between the two objects. We measure the construction times and memory footprints of the two BVHs, and the average running times of the collision detection and distance queries over all steps of the simulations.

We now describe each of the geometries. The geometries were generously supplied to us from Volvo Cars Corporation.

Cembox: We move a small box (33K triangles) into the front panel of a car (457K triangles). There are 33 rigid body movements and 8 of the steps result in collisions.

TunnelConsole: We insert a part (73K triangles) into the interior of a car (300K triangles). The part fits tightly around the stick shift and into the front panel. There are 196 rigid body movements and 136 of the steps result in collisions.

LargeEnvironment: We move a robotic arm (127K triangles) along a large circular path inside a car assembly station (1743K triangles). There are 61 rigid body movements and 13 of the steps result in collisions.

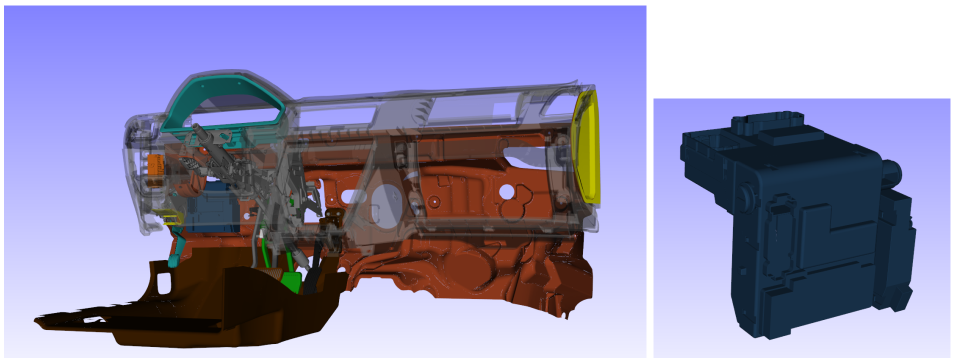
Figure 4. Cembox surroundings with object to be removed in blue. Photo courtesy of Volvo Cars.

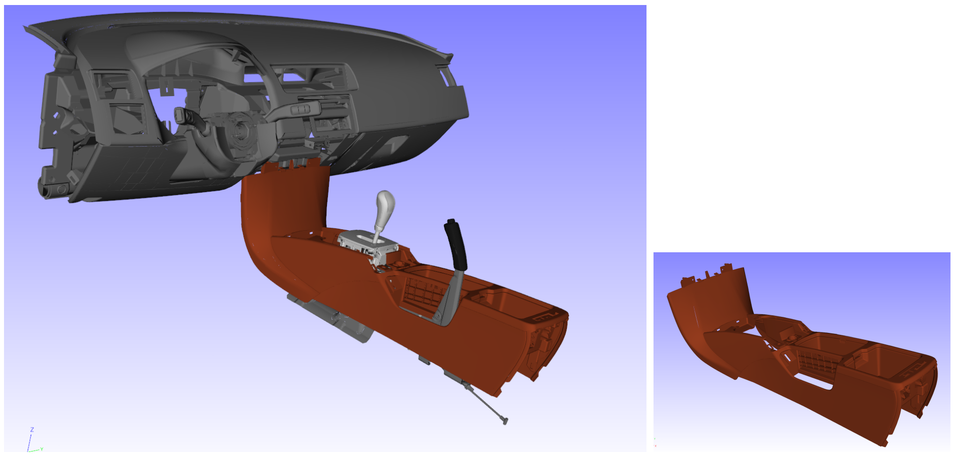
Figure 5. TunnelConsole surroundings with the object to be removed in brown. Photo courtesy of Volvo Cars.

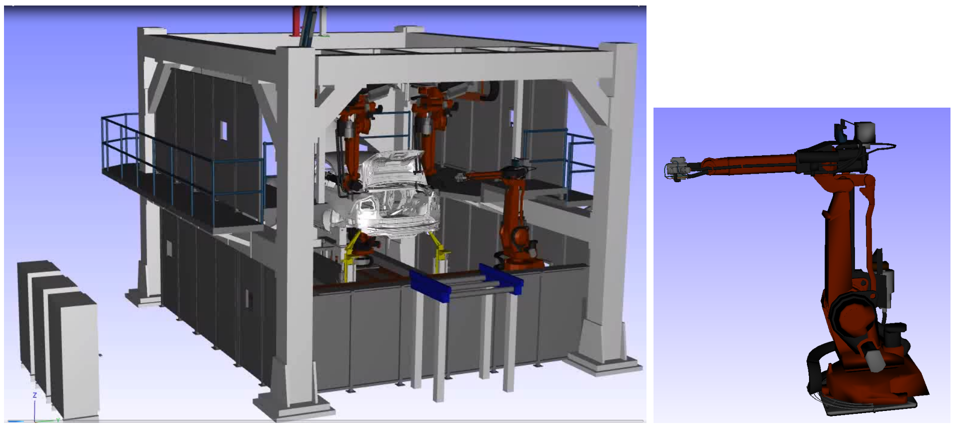
Figure 6. LargeEnvironment with the moving object in orange. Photo courtesy of Volvo Cars.

5.2. Results

We present the results of all initial tests in Tables 2, 3, and 4. In each table we present the build time for both objects' BVH (in seconds), memory usage of both final BVHs (in kB) and the number of collision and distance queries per second (CQ/s and DQ/s respectively). We show the results of MeanSplit and MeanSSE, as well as SSE and non-SSE versions of AxisSplit with the four different cost formulas: C_{SA} for surface area, C_V for volume, C_{IM} for intersection measure, and C_{RT} for ray tracing. The best results in each column are highlighted in bold.

Cembox				
Split Method	Time(s)	Size(kB)	CQ/s	DQ/s
MeanSplit 3.2	1.017	74670	2263	227
MeanSSE 3.4	**0.897**	44308	2286	237
AxisSplit C_{SA} 3.2	2.742	74670	4269	610
AxisSSE C_{SA} 3.4	2.311	**37670**	4347	**622**
AxisSplit C_V 3.2	2.695	74670	3990	426
AxisSSE C_V 3.4	2.293	37917	4089	454
AxisSplit C_{IM} 3.2	2.727	74670	4359	513
AxisSSE C_{IM} 3.4	2.267	37840	**4465**	540
AxisSplit C_{RT} 3.2	2.755	74670	4004	513
AxisSSE C_{RT} 3.4	2.3	37761	4009	563

Table 2. Results from the Cembox benchmark.

Tunnel Console				
Split Method	Time(s)	Size(kB)	CQ/s	DQ/s
MeanSplit 3.2	0.754	58167	1374	236
MeanSSE 3.4	**0.661**	34486	1373	245
AxisSplit C_{SA} 3.2	2.093	58167	2419	339
AxisSSE C_{SA} 3.4	1.636	**29935**	2456	367
AxisSplit C_V 3.2	1.978	58167	2058	**566**
AxisSSE C_V 3.4	1.632	30132	2107	455
AxisSplit C_{IM} 3.2	1.96	58167	2364	370
AxisSSE C_{IM} 3.4	1.663	30074	2410	402
AxisSplit C_{RT} 3.2	1.959	58167	2606	500
AxisSSE C_{RT} 3.4	1.627	30026	**2768**	495

Table 3. Results from the TunnelConsole benchmark.

Large Environment				
Split Method	Time(s)	Size(kB)	CQ/s	DQ/s
MeanSplit 3.2	4.113	284643	14523	21
MeanSSE 3.4	**3.943**	159542	4283	26
AxisSplit C_{SA} 3.2	13.045	284643	17528	46
AxisSSE C_{SA} 3.4	11.217	**137080**	16576	**90**
AxisSplit C_V 3.2	12.776	284643	18154	15
AxisSSE C_V 3.4	11.109	136092	18944	23
AxisSplit C_{IM} 3.2	12.897	284643	**20469**	32
AxisSSE C_{IM} 3.4	11.039	137494	12007	41
AxisSplit C_{RT} 3.2	13.246	284643	20065	35
AxisSSE C_{RT} 3.4	11.486	137311	11641	59

Table 4. Results from the LargeEnvironment benchmark.

Tables 2, 3 and 4 do not give a clear-cut best method for distance computations, however, they let us suspect when to use the two best split strategies. The Volume heuristic proved best only in the TunnelConsole benchmark, where the closest points are more often than not located in large, almost parallel flat surfaces. To confirm our hypothesis that the Surface Area heuristic is the best except for testing parallel surfaces, we present two more test cases. In the fist case we rotated the console in the tunnel console case by 90 degrees and retested with the same movements, see Figure 7. In the second case we created two planes with 20,000 triangles and bent them in the middle with a 45 degree angle. We then added a small amount of noise (uniformly distributed between $[0, 0.001]$) to the z coordinate of both planes, see Figure 8. The results for both cases are presented in Table 5, where we merely present the results for collision and distance queries for both cases. As can be seen, these results confirmed our hypothesis.

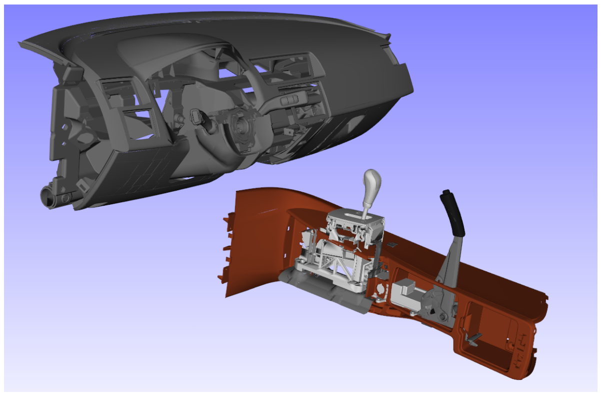

Figure 7. TunnelConsole surroundings with the rotated object to be removed in brown. Photo courtesy of Volvo Cars.

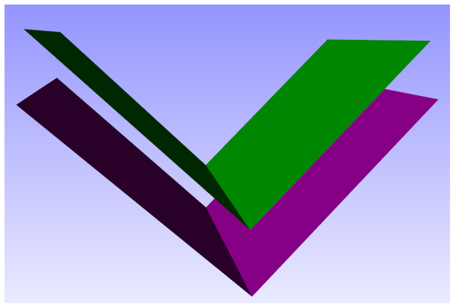

Figure 8. Two Planes case whereby the upper plane (in green) is moved upwards away from the other plane.

Final Two Tests				
Split Method	Rotated Tunnel CQ/s	Rotated Tunnel DQ/s	Two Planes ’0000 CQ/s	Two Planes DQ/s
MeanSplit 3.2	2056	380	1955	67
MeanSSE 3.4	2031	404	1680	81
AxisSplit C_{SA} 3.2	3900	690	2065	83
AxisSSE C_{SA} 3.4	4362	**806**	1673	91
AxisSplit C_V 3.2	3500	770	1950	**95**
AxisSSE C_V 3.4	3591	583	1132	84
AxisSplit C_{IM} 3.2	3823	633	2192	76
AxisSSE C_{IM} 3.4	4113	678	2195	86
AxisSplit C_{RT} 3.2	4676	738	**2340**	81
AxisSSE C_{RT} 3.4	**5000**	791	1671	92

Table 5. Results from the rotated Tunnel Console and two planes benchmarks.

5.3. Comparison of RSS and Triangle Test Execution Time

To quantify how the execution times of triangle tests (without SSE) relate to that of RSS tests and total execution time in BVHs, we measured each for all our benchmarks. The results were very similar, and we only present the results from the Cembox benchmark here, in Table 6. We can see that the query execution time consists of approximately 75% RSS tests and 25% triangle tests. We also see that a single triangle test is three times slower than a single RSS test.

Cembox distance query without SSE	
RSS tests	16030
Triangle tests	1772
Total RSS test time (s)	1.644
Total tri test time (s)	0.561
Average RSS test time (ms)	0.103
Average tri test time (ms)	0.317

Table 6. Results from one distance query in the Cembox benchmark without SSE. Total number of RSS and triangle tests and execution times.

5.4. Impact of SSE Routines on Performance

To test the effects of SSE we ran each of the cases with an SSE build strategy but did not use the SSE routines to compute the distances between triangles at the leaf level. The results in all cases were worse but to give the reader a feel for the results (without again presenting numerous tables of results) we present the results for the Cembox case here in Table 7. It can be seen that the non-SSE routines make the queries about twenty percent slower. We discuss these results more in the next section.

Cembox distance tests with BVHs optimized for SSE		
Split Method	Cembox SSE DQ/s	Cembox no SSE DQ/s
MeanSSE 3.4	237	194
AxisSSE C_{SA} 3.4	622	540
AxisSSE C_V 3.4	454	344
AxisSSE C_{IM} 3.4	540	419
AxisSSE C_{RT} 3.4	563	441

Table 7. Results from the Cembox benchmarks where all BVHs were built for SSE utilization. Average queries per second is given for SIMD-based and non-SIMD-based distance routines in those BVHs.

5.5. Discussion of Results

It is difficult to see a clear pattern for collision query results. These types of queries are not the focus of this article because collision queries are performed fastest with BVHs built out of AABBs [Larsen et al. 1999] and the results were included for interest's sake. Hence, we will instead consider differences in the distance query times.

For the distance queries we can see two strategies being consistently good, AxisSSE C_{SA} and AxisSplit C_V. There are two cases where AxisSplit C_V is best and for all other cases AxisSSE C_{SA} is the best strategy. After testing the first three benchmark cases, we noticed that the TunnelConsole had a significantly different performance profile to the two other cases. As briefly mentioned in section 5.2, we suspected that this had something to do with the fact that the console object that is moved is more like a surface without a real volume than a solid object. This meant that in most of the movements large, flat parts of the console are close to the surroundings. Intuitively, the heuristic minimizing volume should produce flatter RSSs that better approximate flat surfaces, while the heuristic minimizing surface area produces RSSs similar to spheres, cubes and other general, fat shapes, which are more efficient for distance tests in general. To test our hypothesis that using the volume measure is better for parallel flat shapes, we created the test case with two planes. As can be seen, the results from this case confirmed our hypothesis. To be sure we also decided to rotate the tunnel console and run the simulation again, this time conjecturing that the surface area measure would be better. Again, the results confirmed our suspicion. Hence, the optimal choice of measure seems to be based on the type of queries that the user will be doing. If one has a scenario where the objects are thin and surface like (i.e. two dimensional manifolds embedded in three dimensions) and the objects have large areas equally close to each other, then one should use the volume measure for distance queries. For the other cases presented here, where one has a two objects and not all parts are close to other objects, the better choice seems to be the surface area measure.

Using a SSE-optimized splitting strategy for nodes with four or less triangles allows us to increase the performance of distance queries in all benchmarks except in some cases for the C_V measure. For collision queries, however, the performance benefit of SSE varies greatly. In some cases, such as the LargeEnvironment benchmark, we even see a significant slowdown of the collision queries when using the SSE-optimized BVHs for most measures. This result highlights the problems of balancing the pruning power of BVHs and the number of triangles that can be packed in each leaf for better SSE utilization as mentioned in Section 3.

The results for memory reduction when using the SSE heuristics are positive. Since the SSE-optimized BVHs can pack several triangles into each leaf node, they require less nodes and consequently less memory. We see that the memory footprints of the SSE-optimized BVHs are roughly 40–50% smaller than those of the regular BVHs. Although the memory requirements are lower, as can be seen in Tables 2, 3, and 4, switching from MeanSplit to AxisSplit doubles or triples the BVH build time in all benchmarks depending on the leaf split strategy used.

Our SSE implementation of the triangle-triangle tests (see Algorithm 1) are most efficient when exactly four triangle tests are performed at once. If less than four tests are made, we waste computation power, and if more than four are made, we need to call the triangle distance routine multiple times. This has a large impact on which kind of leaf splitting strategy is the most efficient. As can be seen by memory usage in Tables 2, 3, and 4 AxisSSE in combination with ChainSplit produces BVHs of roughly half the size of those produced by MeanSplit, which means that the leaf nodes contain an average of two triangles. This means that tests with two triangles in each leaf become especially common, and the SSE routines can therefore be utilized efficiently.

We notice that although the SSE triangle-triangle distance routines used here are significantly faster than their serial versions, [Shellshear and Ytterlid 2014], in combination with an SSE-optimized splitting strategy they result in inconsistent performance improvements apart from LargeEnvironment where they result in queries that are twice as fast (e.g. comparing AxisSplit C_{SA} with AxisSSE C_{SA}). This seems to indicate that it is often best to do more very cheap RSS-RSS distance tests to be able to do less triangle-triangle distance tests (as shown in Table 6 RSS distance tests are about three times as fast as triangle distance tests). Table 7 is an example that shows that it does not appear to be a good idea to have more than one triangle in a leaf (at least with the splitting strategies presented here) unless SSE triangle distance routines are used.

6. Conclusion and Future Work

We present a construction strategy for BVHs based on sampling different ways of splitting nodes and creating BVs that minimize certain cost functions. We recommend using BV volume as the cost function for objects that are expected to face each other with large surfaces, and BV surface area in all other cases. This strategy is well suited for BVHs for proximity queries, and enables collision tests that are up to twice as fast; and distance tests up to five times as fast as the one presented in Larsen et al. [1999]. We also present a heuristic for packing multiple triangles into BVH leaf nodes in such a way that the triangles in a pair of leaves can be tested against each other efficiently using SSE-based collision- and distance routines. Using this heuristic together with SSE-based triangle-triangle tests yields up to five times as fast distance queries over standard splitting strategies, while slightly reducing BVH build time and decreasing memory footprints of BVHs by up to 50%.

Possible future work includes investigating the use of the AVX and AVX2 instruction sets. A straightforward approach to utilize AVX and similar instruction sets with larger SIMD registers would be to increase the number of triangles allowed in each leaf node. This would most likely lead to an expected reduction in memory consumption but it would be interesting to see whether the intended performance gains materialize or not (as in the case with SSE). Since AVX would be able to test eight triangle pairs at once, other leaf-splitting strategies that pack triangles more tightly and produce leaves with an average of slightly less than three triangles would likely perform better than ChainSplit. It might even be the case that a trivial leaf-splitting strategy that never splits leaf nodes (NoSplit in Section 3.4) would be the optimal choice.

Another avenue for future work is to combine the construction strategies presented here with the recent developments in multi-threaded [Min et al. 2009] and GPU based collision and distance queries [Lauterbach et al. 2010].

We would also like to investigate the use of our probabilistic cost function (Equation 5) for BVs for which we can compute the optimal shape (e.g. axis-aligned bounding boxes, spheres, etc). This would allow a more correct analysis of the quality of such a cost function because as mentioned earlier, there exists no known way to optimally compute RSSs which can lead to the cost function not performing as well as it should.

7. Acknowledgements

The authors of this work gratefully acknowledge Volvo Cars Corporation for the generous allowance for publication of the geometries presented here. This work was carried out at the Wingquist Laboratory VINN Excellence Centre, and is part of the Sustainable Production Initiative and the Production Area of Advance at Chalmers University of Technology. It was supported by the Swedish Governmental Agency for Innovation Systems. The authors are also greatly indebted to the suggestions and comments provided by the reviewers.

References

BENGTSSON, C. 2011. *Approximating optimal swept volumes in two and three dimensions*. Master's thesis, Chalmers University of Technology. 5, 11

HELD, M., KLOSOWSKI, J., AND MITCHELL, J., 1995. Speed comparison of generalized bounding box hierarchies. Technical report, Dept. of Applied Math, State Univ. of New York at Stony Brook. 3

HERMANSSON, T., BOHLIN, R., CARLSON, J. S., AND SÖDERBERG, R. 2013. Automatic assembly path planning for wiring harness installations. *Journal of Manufacturing Systems*. URL: `http://www.sciencedirect.com/science/article/pii/S0278612513000393`, doi:10.1016/j.jmsy.2013.04.006. 2

JEFFREY, A. M. 2005. *Ray tracing with reduced-precision bounding volume hierarchies*. PhD thesis, University of Calgary, Calgary, Alta., Canada. 3

KARRAS, T., AND AILA, T. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, 89–99. URL: `https://research.nvidia.com/sites/default/files/publications/karras2013hpg_paper.pdf`. 3

KLOSOWSKI, J. T., HELD, M., MITCHELL, J. S., SOWIZRAL, H., AND ZIKAN, K. 1998. Efficient collision detection using bounding volume hierarchies of k-DOPs. *Visualization and Computer Graphics, IEEE Transactions on 4*, 1, 21–36. doi:10.1109/2945.675649. 3, 7, 8

LARSEN, E., GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. 1999. Fast proximity queries with swept sphere volumes. Tech. rep., Department of Computer Science, UNC Chapel Hill. URL: `http://gamma.cs.unc.edu/SSV/ssv.pdf`. 2, 4, 7, 11, 12, 19, 21

LARSEN, E., GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D., 2014. PQP. http://gamma.cs.unc.edu/SSV/. 12

LAUTERBACH, C., MO, Q., AND MANOCHA, D. 2010. gProximity: Hierarchical GPU-based operations for collision and distance queries. In *Computer Graphics Forum*, vol. 29, Wiley Online Library, 419–428. doi:10.1177/0278364911429335. 2, 22

LIN, M. C., AND MANOCHA, D. 2003. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, Citeseer. URL: `http://gamma.cs.unc.edu/collision.pdf`. 2

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3, 153–166. doi:10.1007/BF01911006. 2, 5

MIN, T., MANOCHA, D., AND RUO-FENG, T. 2009. Parallel collision detection between deformable objects using SIMD instructions. *Chinese Journal of Computers 32*, 10, 2042–2051. URL: `http://cjc.ict.ac.cn/quanwenjiansuo/2009-10/tm.pdf`. 3, 22

MÜLLER, G., AND FELLNER, D. W. 1999. Hybrid scene structuring with application to ray tracing. In *Proceedings of the International Conference on Visual Computing (ICVC 99)*, 19–26. URL: `http://www.cgv.tugraz.at/V3D2/pubs.collection/ModNav3D/modnav3d-icvc99.pdf`. 3

OSHITA, M., AND MAKINOUCHI, A. 2001. Real-time cloth simulation with sparse particles and curved faces. In *Computer Animation, 2001. The Fourteenth Conference on Computer Animation. Proceedings*, IEEE, 220–227. doi:10.1109/CA.2001.982396. 2

SANTALÓ, L. A. 2004. *Integral geometry and geometric probability*. Cambridge University Press. 4

SEGEBORN, J., SEGERDAHL, D., CARLSON, J. S., CARLSSON, A., AND SÖDERBERG, R. 2010. Load balancing of welds in multi station sheet metal assembly lines. In *Proceedings of the ASME 2010 International Mechanical Engineering Congress & Exposition, Vancouver, British Columbia, Canada, November 12-18, 2010*. doi:10.1115/IMECE2010-40396. 2

SHELLSHEAR, E., AND YTTERLID, R. 2014. Fast distance queries for triangles, lines, and points using SSE instructions. *Journal of Computer Graphics Techniques (JCGT) 3*, 4 (December), 86–110. URL: http://jcgt.org/published/0003/04/05/. 2, 3, 9, 12, 13, 21

SPENSIERI, D., CARLSON, J. S., BOHLIN, R., AND SÖDERBERG, R. 2008. Integrating assembly design, sequence optimization, and advanced path planning. In *ASME Conference Proceedings*, ASME, 73–81. URL: http://link.aip.org/link/abstract/ASMECP/v2008/i43253/p73/s1, doi:10.1115/DETC2008-49760. 2

SPENSIERI, D., EKSTEDT, F., TORSTENSSON, J., BOHLIN, R., AND CARLSON, J. S. 2010. Throughput maximization by balancing, sequencing and coordinating motions of operations in multi-robot stations. In *Proceedings of the 8th International NordDesign Conference 2010*, 455–465. 2

THAKKUR, S., AND HUFF, T. 1999. Internet streaming SIMD extensions. *Computer 32*, 12, 26–34. doi:10.1109/2.809248. 2

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG) 26*, 1, 6. URL: http://doi.acm.org/10.1145/1189762.1206075, doi:10.1145/1189762.1206075. 3, 6

ZACHMANN, G. 2001. *Virtual Reality in Assembly Simulation*. PhD thesis, TU Darmstadt. 3

Author Contact Information

Robin Ytterlid
FCC
Sven Hultins Gata 9D, Gothenburg, Västra Götaland,
41288 Sweden
robin.ytterlid@fcc.chalmers.se

Evan Shellshear
FCC
Sven Hultins Gata 9D, Gothenburg, Västra Götaland,
41288 Sweden
evan.shellshear@fcc.chalmers.se

Robin Ytterlid and Evan Shellshear, BVH Split Strategies for Fast Distance Queries, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 1, 1–25, 2015
http://jcgt.org/published/0004/01/01/

Received:	2014-10-08		
Recommended:	2015-01-15	Corresponding Editor:	Marc Olano
Published:	2015-01-27	Editor-in-Chief:	Morgan McGuire