DOI: 10.1111/cgf.70041
EUROGRAPHICS 2025 / A. Bousseau and A. Dai
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 44 (2025), Number 2

Axis-Normalized Ray-Box Intersection

F. Friederichs[1] and C. Benthin[2] and S. Grogorick[1] and E. Eisemann[3] and M. Magnor[1] and M. Eisemann[1]

[1]TU Braunschweig, Germany [2]Advanced Micro Devices, Inc. [3]Delft University of Technology, The Netherlands

Abstract

Ray-axis aligned bounding box intersection tests play a crucial role in the runtime performance of many rendering applications, driven not by complexity but mainly by the volume of tests required. While existing solutions were believed to be pretty much optimal in terms of runtime on current hardware, our paper introduces a new intersection test requiring fewer arithmetic operations compared to all previous methods. By transforming the ray we eliminate the need for one third of the traditional bounding-slab tests and achieve a speed enhancement of approximately 13.8% *or* 10.9%*, depending on the compiler. We present detailed runtime analyses in various scenarios.*

CCS Concepts
• ***Computing methodologies*** → ***Ray tracing;*** • ***Theory of computation*** → *Computational geometry;*

1. Introduction

Intersecting a ray with axis-aligned bounding boxes (AABB) is at the core of many computer graphics algorithms, such as ray tracing of Bounding Volume Hierarchies (BVH), collision detection, visibility determination, picking and selection, and potentially many others. These intersection tests contribute significantly to ray traversal time and, therefore, to the computation time of many of the mentioned applications, not due to the inherent complexity of the tests but because of the sheer number of tests required, which can easily be in the range of billions per rendered frame.

The seminal bounding slab test [KK86] still forms the basis of most approaches in this area with its basic idea of testing a ray against the six planes defining an AABB and using interval arithmetic to determine the potential intersection. Due to many processor optimizations, such as optimizing for cache-line size and cache architectures, SIMD (single instruction multiple data) processing, branch prediction and others, it is not just about a reduction of the arithmetic operations but also about memory access patterns to reduce the number of cache misses and about how these tests fit onto modern CPU architectures in general. This is one of the reasons why the widely adopted procedure by Smits [Smi98], a robust variant of the slabs test, remains popular to the present due to its compact implementation and notable speed. Branchless variants especially outperform more sophisticated counterparts on current hardware architectures [WWB*14] by a factor of two or more.

In this paper, we introduce a modification to Smits' ray-AABB intersection test and its variants. The main idea is to shift computations from the intersection to the ray generation, as in most applications each ray has to be tested against several AABBs. Transforming the original ray before traversal, we reduce the amount of required arithmetic operations per intersection test *and* shrink the minimally necessary size of the ray data structure during ray traversal.

2. Related Work

Ray tracing is today's standard to render arbitrary geometry, from simple triangles to implicit surfaces [KB89], and to simulate light interactions [PJH16]. To improve efficiency, ray intersection with geometry is minimized by early pruning using bounding volumes (BVs), such as spheres or boxes. These BVs are tested first; only if a ray intersects the BV, the more complex geometry inside is tested. The choice of BV shape affects traversal performance [WHG84] and numeric stability in the intersection tests is crucial [Mit90]. Hierarchical BV arrangements, like BVHs [Cla76], grids [FTI86], and even 5D octrees [AK87], have been proposed. Today, almost all high-performance ray tracers use BVHs with AABBs [WWB*14, Khr].

The most basic test procedure of a ray-AABB intersection test introduced by Kay and Kajiya [KK86] intersects the ray with the slabs formed by two bounding planes for each axis of the coordinate system and compares the resulting ray parameter intervals. If the intersection of all intervals is non-empty, the test result is positive. The original version handles axis-parallel rays separately. Special handling of these edge cases can be avoided by relying on guarantees made in the IEEE-754 floating-point standard [Smi98], but a naïve implementation of this shows a failure case when the interval bounds are ordered based on the sign of the ray direction, and the direction contains -0 floating-point values. To correctly order the interval bounds one should not rely on the sign of the direction components but instead on the sign of the reciprocal of the direction

© 2025 The Author(s). Computer Graphics Forum published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

components, which solves this issue [WBMS05]. Also, computing the reciprocal of the ray direction upfront replaces a costly division with a faster multiplication when calculating the intersection distances to the slabs.

The retrieval of only the closest intersection point with an AABB can theoretically be sped up by culling the back-facing planes bounding it [Woo90]. While this reduces the number of bounding slabs tests from six to three it requires an additional inside-out test for the furthest intersected face and a special handling for rays originating from inside the box, which makes it less suitable for modern processor architectures.

Apart from variants of the classical slab test, methods that use alternative ray representations emerged. Using Plücker coordinates the relative orientation of the ray to the silhouette edges of the box can be efficiently used for intersection testing [MW04]. Comparing slopes of the ray and the vectors from the ray origin to the extremal points of the AABB in their 2D projections can further reduce the required arithmetic operations [EMGM07]. Both, the Plücker and slope test produce only a binary intersection result. If the actual distance is required, it must be computed additionally as in [Woo90]. Both support early exits if a ray misses the box, but due to branch prediction on current hardware this no longer gives a measurable computational advantage. This indicates that only a computational reduction of the worst case will lead to computational benefits, which is why the slab test is again strongly preferred in practice.

The slab test can be implemented very compactly and without branches by removing any early exit checks [SWM21], [MCSM18]. Although, theoretically, more work is done, the absence of branches leads to a significant increase in performance on modern hardware architectures. Another reason is that it trivially supports SIMD to process multiple bounding boxes in parallel at the cost of a single test [WWB*14]. This works particularly well in BVH traversal with branching factors of four or eight, corresponding to the available SIMD width. Additional research on intersecting a ray with oriented bounding boxes exists but will not be covered in this paper focusing solely on AABB, but it is interesting to note that for them, the branchless slab method also plays an important role for efficiency [MCSM18].

The optimized bounding slab intersection test seems to be firmly in place today and high-performance applications, incorporating a decade of research and profiling, use it in their innermost core [WWB*14], which makes any new changes to this test challenging to integrate and challenging to achieve comparable performance right from the start.

Transforming the domain to simplify a given problem is a common theme in the field and is applied to numerous problems, e.g., ray-triangle [MT05] or -oriented bounding box intersections [SVCNM21], as well instancing, which is commonly used for larger scenes and by default in hardware accelerated raytracing APIs [Khr, Sec. 39]. In this paper, we introduce another such approach, which reduces the arithmetic complexity of the ray-AABB intersection test, and we highlight the obstacles that need to be overcome to integrate it into a high-performance system.

3. Overview

The remainder of the paper is organized as follows: For simplicity of explanation, we introduce the notation and the original robust bounding slab test [Smi98] (Sec. 4) first. Based on this, we present the main idea of our axis-normalized ray-AABB intersection procedure (Sec. 5). As efficiency is crucial, we provide implementation details and optimizations in the following section (Sec. 6). We then evaluate the performance of our implementation (Sec. 7) and give a potential outlook on how to integrate it into the high-performance ray tracer *Embree* [WWB*14] (Sec. 8). Even though we evaluate our approach within a CPU-based ray tracer all results directly transfer to a GPU ray-tracer.

While the general idea nicely extends to less or more than three dimensions, we will focus the explanation on the most common case of a three-dimensional setting.

4. Background

4.1. Notation

Let $\boldsymbol{r}: \mathbb{R} \mapsto \mathbb{R}^3$ be a parametric ray, with $t \in I_r = [t^{\min}, t^{\max}]$ being a scalar that parameterizes the position on the ray within a valid ray intersection interval as

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}. \tag{1}$$

Let $\boldsymbol{o} = (o_x, o_y, o_z)^T \in \mathbb{R}^3$ and $\boldsymbol{d} = (d_x, d_y, d_z)^T \in \mathbb{R}^3$ be the origin and the direction vector, respectively. A subscript j denotes the index of the j-th component of a vector valued quantity. In this paper, we mainly consider the procedure in $\mathbb{R}^3$, so $j \in \{x, y, z\}$. We write the element-wise reciprocal of the ray direction as $\boldsymbol{d}^{-1} = \left(d_x^{-1} \quad d_y^{-1} \quad d_z^{-1}\right)^T$. We define an axis-aligned bounding box (AABB) as a tuple of two vectors, representing the two extremal points spanning the box:

$$B = \left(\boldsymbol{b}^{\min}, \boldsymbol{b}^{\max}\right),$$

with $b_j^{\min} \leq b_j^{\max}$ for all axes.

4.2. Robust Bounding Slab Test [Smi98]

The bounding slab test computes the intersection of a ray with each of the axis-parallel bounding planes of the AABB. The ray parameter t of the point of intersection for each of these planes is given by:

$$t_j^k = \left(b_j^k - o_j\right) d_j^{-1} \quad , \text{with } k \in \{\min, \max\}. \tag{2}$$

The interval I_j corresponding to the j-th axis where the ray intersects the respective bounding planes is:

$$\begin{aligned} I_j &= \left[t_j^{\text{near}}, t_j^{\text{far}}\right] \\ &= \left[\min\left(t_j^{\min}, t_j^{\max}\right), \max\left(t_j^{\min}, t_j^{\max}\right)\right] \\ &= \left\{t \,\middle|\, b_j^{\min} \leq r_j(t) \leq b_j^{\max}\right\} \end{aligned}$$

Let $I_{\text{total}} = \left[t^{\text{near}}, t^{\text{far}}\right] = \bigcap_j I_j \cap I_r$ denote the intersection of all three intervals with the valid ray intersection interval I_r. Whenever

© 2025 The Author(s).

I_{total} is non-empty, there is an intersection:

$$\text{intersects}(\boldsymbol{r}, B) = \begin{cases} 1 & \text{if } I_{\text{total}} \neq \emptyset \iff t^{\text{near}} \leq t^{\text{far}} \\ 0 & \text{otherwise.} \end{cases}$$

Most arithmetic operations are required for the ray-plane intersection distances (Eq. (2)). Transforming the ray simplifies the computation for its *dominant axis*, as we show below. We define the dominant axis as the axis j, where d_j has the largest absolute value.

5. Axis-Normalized Ray/AABB Intersection

Consider the 2D-example in Fig. 1, left. The dominant axis is x. We first translate the ray along its direction vector such that the transformed ray's origin $\hat{\boldsymbol{o}}$ coincides with the subspace spanned by the other, non-dominant axes (only y in this 2D-example). Scaling the direction vector by $1/d_x$ the x-component of the transformed ray's direction $\hat{\boldsymbol{d}}$ becomes 1. Now the intersection ray parameter with the near and far plane along the x-axis directly corresponds to the bounding box coordinates b_x^{min} and b_x^{max} without any further computations required.

5.1. Ray Transformation Scheme

The transformation scheme to achieve this configuration is detailed in the following. The scheme is applied once during ray generation.

We begin by finding the index i of the dominant axis of $\boldsymbol{d}$, i.e., the axis with the largest absolute coordinate value:

$$i = \arg\max_j |d_j|$$

Defining the components of the *dominant-axis-normalized* direction vector $\hat{\boldsymbol{d}}$ as

$$\hat{d}_j = d_j d_i^{-1}$$

always results in $\hat{d}_i = 1$, so only $\hat{d}_j$ with $j \neq i$ have to be computed explicitly. By choosing the dominant direction component as normalization, this is well-defined and numerically stable for all non-degenerate (non-zero) direction vectors.

To get the transformed ray origin $\hat{\boldsymbol{o}}$, we intersect the original ray with the plane through the origin spanned by the non-dominant coordinate axes by setting $r_i(t) = 0$ and solving for t:

$$t_o = -o_i d_i^{-1} \tag{3}$$

Substituting t_o into the ray equation, we get for the j-th component:

$$\hat{o}_j = o_j + t_o d_j \tag{4}$$

The component $\hat{o}_i$ is equal to 0 by definition. Only the components $\hat{o}_j$ with $j \neq i$ have to be computed.

Computation of the transformed ray bounds requires conversion between parameters t on the original ray and $\hat{t}$ on the transformed ray. Evaluating the ray equation (Eq. (1)) at t for the dominant component i, we get the following transformation and its inverse:

$$\hat{T}(t) = \hat{t} = t d_i + o_i \tag{5}$$

$$T(\hat{t}) = t = \hat{t} d_i^{-1} - o_i d_i^{-1} \tag{6}$$

As in [WWB*14] and hardware-accelerated GPU raytracing APIs [Khr, Sec. 39.], we keep the ray intersection distance bounds $\left[t^{\text{min}}, t^{\text{max}}\right]$, which is also a requirement for our algorithm to work properly. The described scheme works for all rays with a positive dominant direction component d_i. For a negative d_i the bounds have to be swapped, as the original direction vector is negated in this case. Consider the negative case in Fig. 1, right, as an example. The original ray's bounds are $t^{\text{min}} = 0$ and $t^{\text{max}} = \infty$. To cover the same range with the transformed ray, we must set $\hat{t}^{\text{min}} = -\infty$ and $\hat{t}^{\text{max}} = \hat{T}(t^{\text{min}})$ and look for the *farthest* intersection point instead of the closest.

Given the original ray bounds $\left[t^{\text{min}}, t^{\text{max}}\right]$ and using Eq. (5) we get:

$$\hat{I}_r = \left[\hat{t}^{\text{min}}, \hat{t}^{\text{max}}\right] = \begin{cases} \left[\hat{T}\left(t^{\text{min}}\right), \hat{T}\left(t^{\text{max}}\right)\right] & \text{if } d_i \geq 0 \\ \left[\hat{T}\left(t^{\text{max}}\right), \hat{T}\left(t^{\text{min}}\right)\right] & \text{otherwise.} \end{cases} \tag{7}$$

5.2. Computing Intersections

The transformed ray quantities now allow us to simplify the slab test. As $\hat{o}_i = 0$ and $\hat{d}_i^{-1} = 1$, the intersection test simplifies to

$$\hat{t}_j^k = \begin{cases} \left(b_j^k - \hat{o}_j\right) \hat{d}_j^{-1} & \text{if } j \neq i \\ b_j^k & \text{otherwise.} \end{cases} \tag{8}$$

In the positive case we return the lower bound $\hat{t}^{\text{near}}$ of the resulting interval, as in the original slab test. In the negative case, we must return $\hat{t}^{\text{far}}$ instead (see Fig. 1, right).

In cases, where the true physical intersection distance is required, e.g., for volume tracking, it can be recovered by inversely transforming the resulting intersection distance using Eq. (6):

$$t^{\text{near}} = \begin{cases} T\left(\hat{t}^{\text{near}}\right) & \text{if } d_i \geq 0 \\ T\left(\hat{t}^{\text{far}}\right) & \text{otherwise.} \end{cases}$$

5.3. Invertibility

The transformation scheme is not a bijection by definition because all rays with the same direction up to sign and scale, and the origin on the same line, are mapped to the same transformed ray. Similarly to Eq. (5) and (6) however, keeping the two original dominant components o_i and d_i suffices to reconstruct the original ray's origin $o_{j \neq i} = \hat{o}_j + o_i \hat{d}_j$ and direction $d_{j \neq i} = d_i \hat{d}_j$, if needed.

6. Implementation

The Embree ray tracer [WWB*14] contains a branchless version of the Smits' slab test [Smi98], highly optimized for modern hardware. We use this as the basis for our implementation.

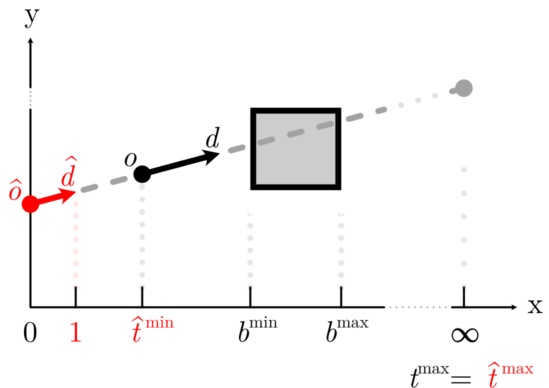
The classical branchless slab test is a direct translation of Sec. 4.2 to code and can be found, e.g., in [SWM21].

In Embree, they further optimized the algorithm by expanding the slab test expression (Eq. (2)):

$$t_j^k = \left(b_j^k - o_j\right) d_j^{-1} = b_j^k d_j^{-1} \underbrace{-o_j d_j^{-1}}_{\text{pre-compute}}$$

© 2025 The Author(s).

Negative dominant ray direction component



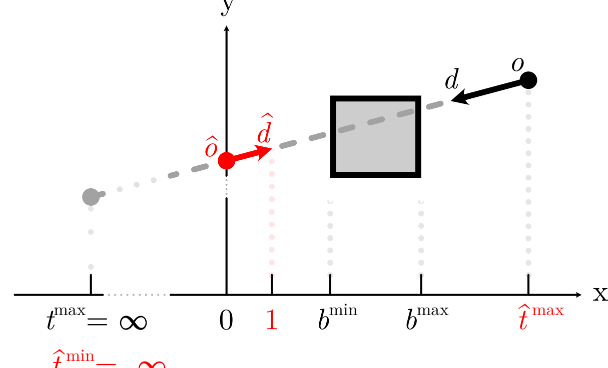


Figure 1: *Positive and negative case of the axis normalized ray-box intersection test. Black symbols correspond to the original ray, red marks the transformed quantities. The original ray bounds are assumed to be $t^{min} = 0$ and $t^{\max} = \infty$ here. In the negative case we search for the furthest intersection instead of the closest.*

The second term of the right-hand side depends only on the ray itself and can thus be pre-computed once during ray initialization. This form of the expression is likely to be translated into a single fused-multiply-add instruction by the compiler, instead of one subtraction plus one multiplication. Some extra care must be taken when implementing this using floating-point arithmetic. The pre-computed term can become NaN when both $d_j = 0$ and $o_j = 0$, which also causes the resulting t_j^k to be NaN. Embree circumvents this by bounding the absolute value of the direction components from below with a small constant (1×10^{-8}), which we also adopt in our code.

Secondly, the selection of the AABB bounds can be optimized. Given that they are properly ordered from low to high coordinate values, the bounds corresponding to t_j^{near} and t_j^{far} are fully determined by the sign of d_j. This makes six inner *min* and *max* operations in the original test unnecessary, but requires six additional *load offsets* in the ray data structure. These are used to access the respective AABB coordinates and can be pre-computed using a handful of bitwise shift- and xor-instructions. The purely scalar code is shown in listing 1. Below is another procedure `intersect_dist` that additionally returns the closest intersection distance via the output parameter `t`.

```
1  struct aabox {
2      float p[6]; // {b_j^k}
3  };
4  struct ray_opt {
5      float rd[3]; float nrdo[3];
6      float tmin; float tmax;
7      uint32_t bnx, bfx, bny, bfy, bnz, bfz;
8  };
9  bool intersect(const ray_opt &r, const aabox &b) {
10     // compute slab intersection distances
11     float tx1 = b.p[r.bnx] * r.rd[0] + r.nrdo[0];
12     float tx2 = b.p[r.bfx] * r.rd[0] + r.nrdo[0];
13     float ty1 = b.p[r.bny] * r.rd[1] + r.nrdo[1];
14     float ty2 = b.p[r.bfy] * r.rd[1] + r.nrdo[1];
15     float tz1 = b.p[r.bnz] * r.rd[2] + r.nrdo[2];
16     float tz2 = b.p[r.bfz] * r.rd[2] + r.nrdo[2];
17     // intersect above intervals
18     float tnear =
19         max(max(max(tx1, ty1), tz1), r.tmin);
20     float tfar =
21         min(min(min(tx2, ty2), tz2), r.tmax);
22     return tnear <= tfar;
23 }
24 bool intersect_dist(const ray_opt &r, const aabox &b, float &t) {
25     // [...] same as above
26     t = tnear;
27     return tnear <= tfar;
28 }
```

Listing 1: *Branchless slab test with optimizations from Embree.* `r.nrdo` *are the precomputed $-o_j d_j^{-1}$ terms. The bounding box coordinates are accessed via precomputed load offsets* `r.b**`.

6.1. Dominant-Axis-Normalized Intersection Test

We can now apply the optimization described in Sec. 5 to the optimized slab test (listing 1).

Ray Initialization: First, we transform the ray into our new representation. It is a straightforward translation of the steps in Sec. 5.1 to code, which can be found in the supplemental material.

Handling per-Axis Cases by Permuting Components: Applying our optimization conditionally based on the dominant ray direction requires inefficient branching. We avoid this using load offsets. During ray initialization, we reorder (permute) the original components of $\boldsymbol{o}$ and $\boldsymbol{d}$, such that o_i and d_i are assigned to the last (third) dimension of our transformed ray. The load offsets for the bounding box coordinates are adapted accordingly. The remaining code is then the same for all cases, because the intersection test itself is invariant to permutations of the axes.

The resulting code is shown in listing 2. We have renamed the `x`, `y`, and `z` related quantities to `x0`, `x1` and `x2` to emphasize the mentioned permutation of the ray components. For the dominant axis `x2`, we now only have to read the respective bounding-box coordinates. The computations for the other two components are equivalent to the original test.

Ray Data Structure: Our transformation scheme allows us to significantly shrink the data structure for a single ray. The i-th origin and direction components $\hat{o}_i$ and $\hat{d}_i$ are always 0 and 1 by con-

© 2025 The Author(s).

```
1  struct norm_ray {
2      float rd[2]; float nrdo[2];
3      float tmin; float tmax;
4      uint32_t bnx0, bfx0, bnx1, bfx1, bnx2, bfx2;
5  };
6
7  bool intersect(const norm_ray& r, const aabox &b) {
8      // compute slab intersection distances
9      float tx0n = b.p[r.bnx0] * r.rd[0] + r.nrdo[0];
10     float tx0f = b.p[r.bfx0] * r.rd[0] + r.nrdo[0];
11     float tx1n = b.p[r.bnx1] * r.rd[1] + r.nrdo[1];
12     float tx1f = b.p[r.bfx1] * r.rd[1] + r.nrdo[1];
13     float tx2n = b.p[r.bnx2];
14     float tx2f = b.p[r.bfx2];
15     // intersect above intervals
16     float tnear =
17         max(max(max(tx0n, tx1n), tx2n), r.tmin);
18     float tfar =
19         min(min(min(tx0f, tx1f), tx2f), r.tmax);
20     return tnear <= tfar;
21 }
```

Listing 2: *Binary intersection test with dominant-axis-normalized rays.* `r.nrdo` *are the precomputed* $-\hat{o}_j\hat{d}_j^{-1}$ *terms.*

```
1  struct norm_ray_int {
2      // [...] same as norm_ray in listing 2
3      uint32_t tret;
4  };
5  float select(uint32_t mask, float f, float n) {
6      return bit_cast<float>((bit_cast<uint32_t>(f) & mask) |
7          (bit_cast<uint32_t>(n) & ~mask));
8  }
9  bool intersect_dist(const norm_ray_int& r,const aabox &b,float &t)
10 {
11     // [...] same as lines 9 to 19 in listing 2
12     t = select(r.tret, tfar, tnear);
13     return tnear <= tfar;
14 }
```

Listing 3: *Intersection test with dominant-axis-normalized rays, including closest-hit distance computation.* `r.tret` *decides which distance is returned, depending on the sign of* d_i *(see Sec. 5.2).*

struction. The same is true for the derived quantities $-\hat{o}_i\hat{d}_i^{-1}$ and $\hat{d}_i^{-1}$. Thus we can remove two 4-byte floats from our traversal ray data structure, resulting in a smaller size of 48 instead of 56 bytes.

Retrieving Intersection Distances: Listing 3 highlights the necessary changes from the binary test to acquire the intersection distance, following Sec. 5.2. Depending on the sign of d_i, we need to return either $\hat{t}^{\text{near}}$ or $\hat{t}^{\text{far}}$. This choice can be pre-computed during ray initialization and is stored as a mask in `r.tret`. We call `select` in Line 12, which returns `tfar` if `r.tret` indicates a negative case, and `tnear` otherwise. We chose to implement this as a branchless mask selection, as a ternary conditional operator at this location led to conditional branches with some compilers, even with optimizations enabled. The additional mask in the ray data structure occupies 4 additional bytes, but this ray is still 4 bytes smaller than the original ray.

7. Evaluation

We modified the existing test suite in [EMGM07], which built on the test suite in [MW04], to evaluate our approach on a synthetic benchmark. For our tests, we sample 10,000 rays with uniformly distributed ray origin and direction in the scene bounds $[-1,1]^3$. For each ray, $1k$ axis-aligned bounding boxes with uniformly distributed position and size $\boldsymbol{s} \in [0.05, 1.5]^3$ are rejection-sampled, such that we achieve a desired hit ratio for the sampled ray. After that, the boxes for each ray are randomly shuffled to rule out patterns emerging in memory access due to the box sampling procedure. We tested three different hit ratios, so that each ray hits 0%, 50% and 100% of the boxes. Every combination of algorithm and hit ratio is treated as a separate test case. In each test case, each of the $10k$ rays is intersected with its respective set of $1k$ boxes (one ray at a time). For the compared methods, we ran both the binary intersection tests, which only return true/false, and the respective variants that also return intersection distances. Each test case is repeated $5k$ times and we measure the average execution time (in CPU time) per iteration. Before every test, we run a validation pass on the generated test data to confirm that the results of all tested algorithms are the same.

We ran the benchmarks single-threaded on an *AMD Ryzen 9 7950X Processor* with *Ubuntu 22.04 LTS* installed. To reduce noise in the timings, we isolated the first physical core, pinned the benchmarking process to the corresponding logical core, and fixed the core's frequency to its base clock (4.5 GHz). The benchmark was compiled with *g++ 12.3.0* and *clang++ 15.0.7*, both with CMake's default Release build compile flags `-O3 -DNDEBUG -std=gnu++20 -MD -MT`. Both were, at the time of writing, the most recent versions available via the OS's package repositories.

From a GPU implementation we would expect largely the same results, due to the scalar execution model on current devices. However, with current hardware raytracing APIs (*Vulkan, OptiX, DXR*), it is not possible to run ray/AABB tests in isolation or modify them. A purely compute-based comparison would be meaningless, so we decided to skip the GPU version for now and concentrate on the CPU.

7.1. Results

In this comparison, we included results for the Pluecker [MW04] and Slope [EMGM07] tests, and the optimized branchless slab test as our baseline (listing 1). For detailed numbers on an extended set of intersection procedures, please refer to the supplementary material.

Transforming the rays before traversal adds a small amount of extra cost to our method, although the other algorithms also show non-negligible pre-computation costs. However, they all are insignificant compared to time spent in the actual intersection test, especially when multiple boxes are tested against the same ray. Table 1 lists the ray-initialization times for 10k rays averaged over the 5k test runs.

	Algorithm	Clang 15.0.7	GCC 12.3.0
Binary	Plücker	0.0727	0.0699
	Slope	0.0742	0.0786
	Branchless Slab	0.0688	0.0584
	Ours	0.0868	0.0931
With Distances	Plücker	0.0724	0.0693
	Slope	0.0742	0.0783
	Branchless Slab	0.0687	0.0582
	Ours	0.0914	0.0963

Table 1: *Ray initialization times in milliseconds for 10k rays, averaged over 5k runs.*

Fig. 2 shows the benchmark results for the binary tests. There is

© 2025 The Author(s).

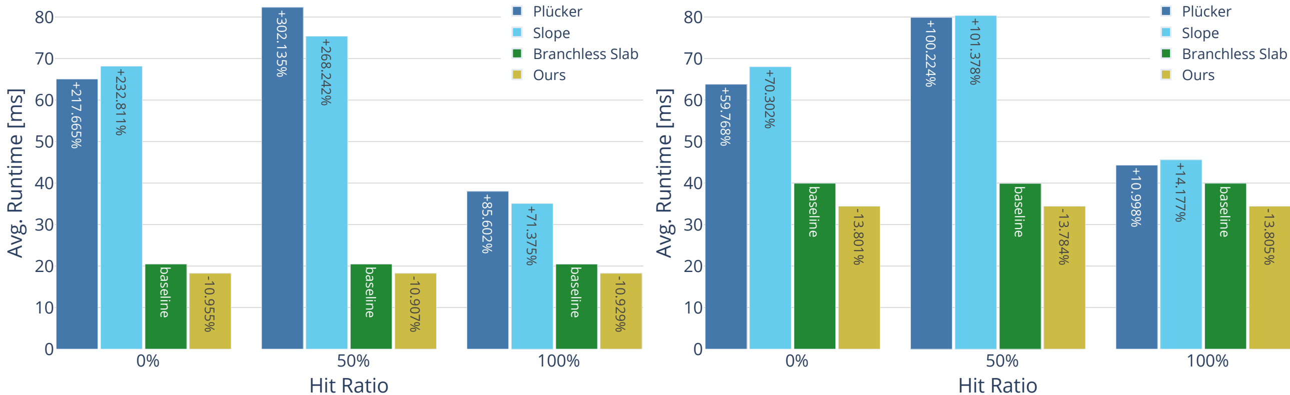


Figure 2: *Performance results of the binary intersection test for g++ 12.3.0 (left) and clang++ 15.0.7 (right). Relative performance to the optimized branchless slab test (baseline) is noted in the bars.*

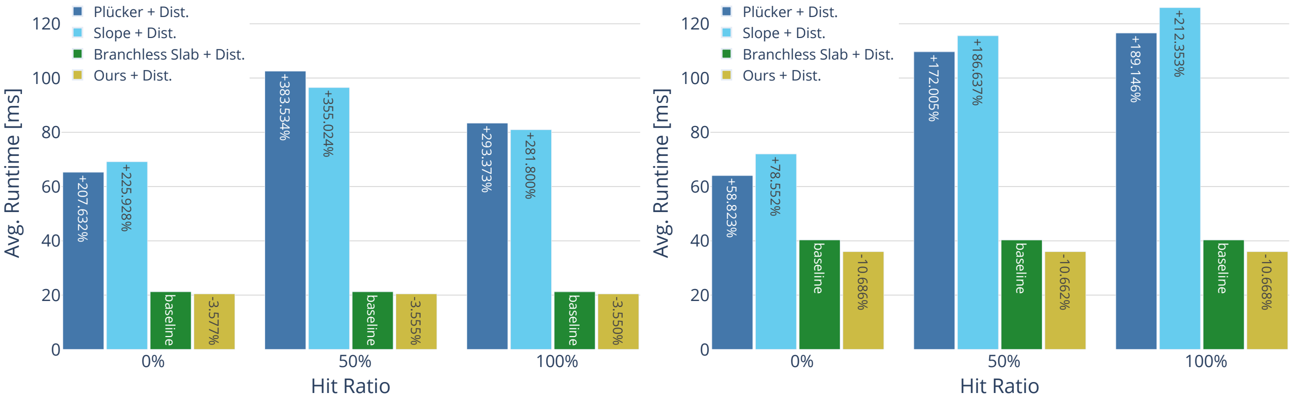


Figure 3: *Performance results for g++ 12.3.0 (left), and clang++ 15.0.7 (right) with intersection distances.*

a clear advantage of the two branchless algorithms over the other intersection tests. Our method is measurably faster than the baseline in all cases. Compiling with clang++, we achieve a roughly 13.8% faster average runtime compared to the optimized slab-test implementation. In terms of absolute runtime, the code generated by clang is much less efficient than the g++ counterpart. With g++, we achieve a performance increase of about 10.9%. As expected, the branching tests are slowest in the 50% hit ratio scenario due to unpredictable branches, whereas the branchless tests are not impacted at all.

Additionally, computing intersection distances (Fig. 3) slightly shrinks the runtime advantage of our algorithm, mostly due to an additional load of `tret` and the conditional return of either the near or far distance. The generated machine code for the conditional return is very much dependent on the compiler, and g++ seems to generate less efficient code for our approach here. In theory, we can get rid of the conditional return. As `tret` gets fixed during ray generation, we could create two traversal procedures; one for the positive and one for the negative case. However, this did not result in any performance improvement in our tests.

7.2. Profiling and Assembly Analysis

A smaller test run (100 rays, 1000 boxes per ray and 100 repetitions) using *valgrind-callgrind* reveals some more details about the runtime performance behavior. However keep in mind, that the total numbers, especially the cycle estimate, are not directly comparable to execution times on a real CPU. In Tables 2 and 3, we show results for the algorithms tested. The most obvious difference between the branching and branchless algorithms is the absence of branch mispredictions (Bm) in the latter case. This also leads to drastically reduced L1 instruction cache misses (I1mr), although they fetch a similar amount of instructions on average (Ir).

Comparing the baseline (branchless slab) and our algorithm, we consistently fetch fewer instructions (Ir) in both the binary test and the one that additionally returns intersection distances. Our test also reads less data (Dr) than the baseline in all cases. The Slope test fetches the least amount of instructions on average when compiled with clang++, but it noticeably suffers from branch mispredictions. In total, the numbers for g++ more closely align with the ones we have measured on the real CPU.

© 2025 The Author(s).

Algorithm	Ir	Dr	Dw	I1mr	D1mr	Bm	CEst
				Binary			
Plücker	3.29	**3.44**	0.00	0.39	4.32	4.30	3.41
Slope	**2.72**	5.27	0.00	0.44	4.32	4.80	**2.94**
Branchless Slab	3.76	4.85	0.00	**0.04**	**4.31**	**0.00**	3.51
Ours	3.28	4.31	0.00	**0.04**	**4.31**	**0.00**	3.09
				With Distances			
Plücker	3.66	**3.85**	**2.17**	0.42	4.32	6.44	3.89
Slope	**3.23**	6.22	**2.17**	0.46	4.32	6.32	3.50
Branchless Slab	3.82	4.85	2.33	**0.04**	4.32	**0.00**	3.57
Ours	3.70	4.58	2.33	0.05	**4.31**	**0.00**	**3.46**

Table 2: *Callgrind results for clang++ 15.0.7. Lower is better. Column names are the captured event types:* Ir*: Instructions fetched,* Dr*: Data reads,* Dw*: Data writes,* I1mr*: L1 instruction read misses,* D1mr*: L1 data read misses,* Bm*: Mispredicted branches,* CEst*: Total cycle estimation.*

Algorithm	Ir	Dr	Dw	I1mr	D1mr	Bm	CEst
				Binary			
Plücker	3.71	**2.98**	0.00	0.36	4.32	4.52	3.80
Slope	3.11	4.91	0.00	0.32	4.32	4.47	3.27
Branchless Slab	2.82	5.42	0.00	0.03	**4.31**	**0.00**	2.69
Ours	**2.22**	4.90	0.00	**0.02**	4.31	**0.00**	**2.16**
				With Distances			
Plücker	4.26	**3.37**	2.99	0.41	4.32	6.15	4.40
Slope	3.80	5.82	2.99	0.39	4.32	5.99	3.99
Branchless Slab	**2.89**	5.42	**2.53**	**0.03**	**4.31**	**0.00**	**2.75**
Ours	**2.89**	5.16	**2.53**	**0.03**	**4.31**	**0.00**	**2.75**

Table 3: *Callgrind results for g++ 12.3.0. Lower is better. Column names are the captured event types (see Table 2).*

8. Towards High-Performance BVH Traversal

BVH traversal is one of the key applications for ray-AABB intersection. Unfortunately, it is nontrivial to implement a new ray-AABB intersection test such as ours into existing applications, as the intersection test is often at the core of frameworks that have undergone a decade of code optimization, where every register load and order of arithmetic operation is carefully optimized. Exchanging it leads to a host of unforeseeable problems.

To point out the different challenges that are to be overcome, if the frameworks have not been built from scratch around the intersection test, we have implemented our algorithm in the *Embree raytracing kernels* library [WWB*14] in the *SSE2* code path, which employs a BVH with a branching factor of four and computes ray-AABB intersections with the child nodes in parallel. Our goal is to achieve the same traversal sequence as with the original ray parameterization. The necessary changes to the code as well as their runtime implications are detailed in the following.

BVH traversal: Embree originally uses an additional SIMD-*traversal ray* for intersecting bounding boxes and their children, which stores 4-wide SIMD values for parallelizing the intersection calculations. The original scalar ray is used for primitive intersection. In our implementation, we replaced the traversal ray with the transformed version.

Traversal with the transformed ray is similar to the usual algorithm, although there is one key difference: When the direction is inverted compared to the original ray, we search for the *largest* intersection distance and visit the *farthest* child node first (with respect to the transformed ray). This ensures the same traversal sequence as when using the standard traversal algorithm.

Stack culling and traversal ray bounds update: Embree maintains a stack of BVH nodes. Newly-found intersecting child nodes get pushed onto this stack, far nodes first, together with the intersection distance to the corresponding AABB. When the next node is popped for traversal, the currently closest distance to any primitive is compared to the node's intersection distance. If the node's distance is larger, the node can safely be skipped and the next node is popped from the stack.

Typically, the lower ray bound $t^{\min}$ is updated during traversal if intersections with primitives in leaf nodes are encountered. With our modified algorithm, we have to update $\hat{t}^{\min}$ or $\hat{t}^{\max}$, if $d_i < 0$. Because in our implementation, the distances of the transformed traversal ray and the primitive ray are not directly comparable, we must inversely transform (Eq. (6)) the distance returned by the primitive intersection to our representation when updating the bounds of the traversal ray, every time a closer primitive is found in one of the BVH leaves. That requires an additional multiplication and addition, or fused multiply-add depending on the hardware, and two additional reads. However, as this update can only occur at the leaves of the BVH, the performance impact is lower for large trees. One could also use the axis-normalized ray for primitive intersection, but this could require special handling for negative ray directions, depending on the primitive type.

Sorting negative intersection distances: In case of $d_i < 0$, we have to proceed with the far intersection distance and, consequently, push the intersected children in order of *ascending* distance instead. This creates two different code paths; one for positive d_i and one for negative ones. For each BVH traversal, this causes an additional branch misprediction. For large BVH trees, it is less of a problem than for small ones with shorter traversals, as the branch only occurs once right after the initialization of the traversal ray.

A far more severe consequence of our ray parameterization is that the code that sorts hit child nodes and pushes them onto the traversal stack in the right order (closest or farthest node first, depending on the sign of d_i), now has to handle negative distances. The original, highly optimized code simply reinterprets floats as unsigned integer keys, which can be compared more cheaply than floats directly, but as a consequence gives wrong results for negative distances. To create keys that compare more efficiently than floats but still correctly in the positive and negative case, we map the floating-point range to signed 32-bit integers instead, for all four child-intersection distances at once in SIMD-fashion (listing 4). This is efficient but still more work than a simple reinterpretation, which is a no-op by definition.

Wider SIMD: All of the above can similarly be applied to the AVX2 / AVX512 code path, which utilizes eight instead of four-wide SIMD instructions. One can expect more or less the same results up to a constant factor.

8.1. Performance Analysis

Using the basic path-tracer example included in the Embree repository, we rendered three scenes from [McG17] with 1024 samples

© 2025 The Author(s).
Computer Graphics Forum published by Eurographics and John Wiley & Sons Ltd.

```
1  __forceinline __m128i float4_to_ikey(__m128 f)
2  {
3      __m128i key = _mm_castps_si128(f);
4      __m128i mod = _mm_and_si128(
5          _mm_srai_epi32(key, 31),
6          _mm_set1_epi32(~(1 << 31))
7      );
8      key = _mm_xor_si128(key, mod);
9      return key;
10 }
```

Listing 4: *Conversion of possibly negative float values to correctly comparable integer keys.*

per pixel and a resolution of 1024×1024 and compared the total runtime and differences between the original implementation and our modified one. Numerical results are given in Table 4. Our implementation is slightly slower, due to the reasons discussed above. The resulting images are visually identical (Fig. 4). A small remaining error stems from the fact that even the slightest numerical differences e.g., caused by our ray transformation procedure can change the outcome of an intersection test. Nevertheless, the error is negligible and proves that our method can be applied successfully to real demanding applications, and is not limited to purely synthetic experiments.

Scene	Time Orig. in s	Time Ours in s	PSNR	MAE	Max. Diff.
Cornell Box	32.877	34.472	104.077	2.543e-6	1.0
Hairball	140.473	145.787	∞	0.0	0.0
San Miguel	246.108	252.726	94.657	2.225e-5	1.0

Table 4: *Runtime and error results for a 1024 sample render of three scenes with 1024x1024 resolution. The error metrics were computed on the 8-bit sRGB output images [clang++ 15.0.7 with compile flags `-O3 -DNDEBUG -msse2`].*

8.2. Numerical Error Analysis

To gain further insight into the potential numerical error introduced by the ray transformation, we conducted another experiment. We placed an icosphere mesh consisting of 81,920 triangles centered at distances of 2^j around the scene's origin. 100 of these center positions were randomly sampled for each j on the corresponding spherical shell. We also varied the radius of the sphere mesh according to 2^k, with $k \in -16, -15, \ldots 20$ and $j \in -16, -15, \ldots 20$. The sizes of the leaf bounding boxes are obviously much smaller than the radii. Their mean diagonal length spans a range of $\approx [3.8 \cdot 10^{-7}, 2.6 \cdot 10^4]$ for the selected scales. For each pair (j,k) and the 100 position samples, we transformed the vertices accordingly and rebuilt the BVH to avoid intersection tests in (possibly better conditioned) local geometry coordinate systems skewing the results. For each (j,k) we then traced 10,000 rays from the respective sphere's center to randomly sampled points on the triangle edges, as these are the most critical points, and counted the number of rays m that missed the scene entirely. This should be zero for a perfect traversal and triangle intersection routine, as the surrounding sphere mesh is water-tight. For each pair (j,k) we computed the ratio of missing rays to total number of rays $\frac{m}{100 \cdot 10{,}000}$. As the triangle intersector we chose the Plücker variant implemented in Embree that guarantees water-tightness at the edges under reasonable scene bounds. A ray leaving the scene would most likely indicate a numerical problem in the BVH-traversal routine. The same random seed is used for both variants and, as before, the original ray is used for primitive intersections. Thus, any difference in the number of misses can only be caused by the BVH-traversal procedure itself.

The results are shown in Fig. 5. Both tests fail at some point due to large distances from the scene origin combined with small bounding box sizes. The boundary of the domain where almost all tests fail is the same for both methods. Our test shows a small band before the critical threshold, where it starts producing spurious traversal errors. This was to be expected due to the additional ray transformation, and can be largely attributed to multiplication of large floating-point values with small ones, and the addition/subtraction of large values in our ray transformation (Eq. (3) and (4)).

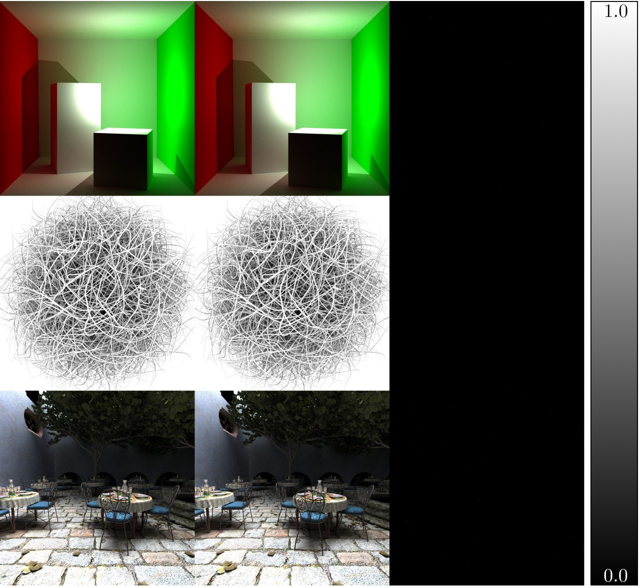


Figure 4: *Visual comparison and absolute error for the three test scenes. From top to bottom:* Cornell Box, Hairball, San Miguel. *From left to right: reference, ours, abs. difference. Brighter pixels in the error image mean larger error. Fully white pixels correspond to the max. absolute difference (*1.0*) in the* $[0,255]$ *sRGB colorspace found in any of the image pairs (see last column of Table 4).*

The bounds of common scenes are often rather small; the scenes found in [Bit16] for example are bounded by $((-111 \quad -16 \quad -120), (111 \quad 86 \quad 111))$ at maximum. For these smaller bounds the (signed) difference in traversal errors compared to the reference is in the low $\pm 10^{-6}$ range, and overall negligible. For larger bounds, e.g. origin distances in the range $[0, 5000]$ and sphere sizes in $[0.001, 5000]$ (marked using white boxes in Fig. 5), our test causes some traversal errors at the extreme corner of the range. A common strategy, which Embree also makes use of in its *robust* code path, is to pad the per-axis interval bounds by a few ULPs ("Unit in Last Position") to make the traversal conservative. Adopting this for our method reduces the remaining error significantly and makes our method applicable for larger scene bounds. Scenes can also often be scaled down appropriately, and for extremely large scenes a double precision implementation is usually required, even with the reference test.

© 2025 The Author(s).

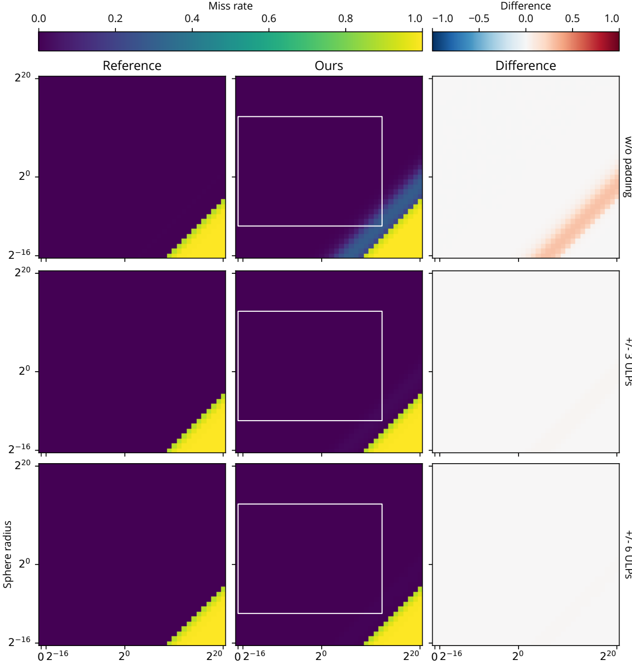


Figure 5: *Fraction of rays that miss due to traversal errors. White boxes denote distances from origin in the range* $[0, 5000]$ *and sphere radii in* $[0.001, 5000.0]$ *scene units.*

9. Conclusion and Future Work

We presented a novel ray-AABB intersection procedure that shows a measurable speed improvement over the highly-optimized branchless slab test employed in high-performance systems. We reduce the required bounding slabs tests by one third and achieve a speed up of consistently more than 10% in our synthetic benchmark, with imperceptible numerical differences when applied to rendering complex scenes. We also showed that the test can be made conservative for a large range of scene sizes by padding the respective quantities accordingly. The efficient implementation in current state-of-the-art raytracing libraries such as *Embree* remains an open, yet solvable and interesting problem.

In the future, we want to map our test to single-ray/single-AABB intersection using SIMD to effectively trace incoherent rays, as our test only needs to test four bounding slabs which would fit nicely into one SSE2 SIMD register; an improved hardware implementation or reduced power consumption in dedicated hardware could be other benefits that should be investigated. Our test might enable an optimized traversal procedure of sparse octrees or general voxel grids. Thus, CPU and GPU voxel ray-tracers are the next logical step.

Acknowledgements

This work was partially funded by the DFG project "Increasing Realism of Omnidirectional Videos in Virtual Reality" (ID 491805996) and "Immersive Tech Lab within Convergence AI at TU Delft". Open Access funding enabled and organized by Projekt DEAL.

References

[AK87] Arvo J., Kirk D.: Fast ray tracing by ray classification. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (1987), SIGGRAPH '87, p. 55–64. doi:10.1145/37401.37409. 1

[Bit16] Bitterli B.: Rendering resources, 2016. https://benedikt-bitterli.me/resources/. 8

[Cla76] Clark J. H.: Hierarchical geometric models for visible surface algorithms. *Commun. ACM 19*, 10 (1976), 547–554. doi:10.1145/360349.360354. 1

[EMGM07] Eisemann M., Magnor M., Grosch T., Müller S.: Fast ray/axis-aligned bounding box overlap tests using ray slopes. *Journal of graphics tools 12*, 4 (2007), 35–46. 2, 5

[FTI86] Fujimoto A., Tanaka T., Iwata K.: Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications 6*, 4 (1986), 16–26. doi:10.1109/MCG.1986.276715. 1

[KB89] Kalra D., Barr A. H.: Guaranteed ray intersections with implicit surfaces. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (1989), SIGGRAPH '89, p. 297–306. doi:10.1145/74333.74364. 1

[Khr] Khronos® Vulkan Working Group: *Vulkan® 1.3.273 - A Specification (with all registered extensions)*. https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#ray-traversal. 1, 2, 3

[KK86] Kay T. L., Kajiya J. T.: Ray tracing complex scenes. *ACM SIGGRAPH computer graphics 20*, 4 (1986), 269–278. 1

[McG17] McGuire M.: Computer graphics archive, 2017. URL: https://casual-effects.com/data. 7

[MCSM18] Majercik A., Crassin C., Shirley P., McGuire M.: A ray-box intersection algorithm and efficient dynamic voxel rendering. *Journal of Computer Graphics Techniques Vol 7*, 3 (2018), 66–81. 2

[Mit90] Mitchell D.: Robust ray intersection with interval arithmetic. In *Proceedings of Graphics Interface '90* (1990), pp. 68–74. doi:10.20380/GI1990.08. 1

[MT05] Möller T., Trumbore B.: Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*. 2005, pp. 7–es. 2

[MW04] Mahovsky J., Wyvill B.: Fast ray-axis aligned bounding box overlap tests with Plücker coordinates. *Journal of Graphics Tools 9* (2004), 35 – 46. 2, 5

[PJH16] Pharr M., Jakob W., Humphreys G.: *Physically Based Rendering: From Theory to Implementation (3rd ed.)*, 3rd ed. Morgan Kaufmann Publishers Inc., Oct. 2016. 1

[Smi98] Smits B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14. 1, 2, 3

[SVCNM21] Sabino R., Vidal C. A., Cavalcante-Neto J. B., Maia J. G. R.: *Fast and Robust Ray/OBB Intersection using the Lorentz Transformation, Ray Tracing Gems II*. Apress, 2021, ch. 32. 2

[SWM21] Shirley P., Wald I., Marrs A.: *Ray Axis-Aligned Bounding Box Intersection, Ray Tracing Gems II*. Apress, 2021, ch. 2. 2, 3

[WBMS05] Williams A., Barrus S., Morley R., Shirley P.: An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools 10* (01 2005), 49–54. 2

[WHG84] Weghorst H., Hooper G., Greenberg D. P.: Improved computational methods for ray tracing. *ACM Trans. Graph. 3*, 1 (1984), 52–69. doi:10.1145/357332.357335. 1

[Woo90] Woo A.: Fast ray-box intersection. In *Graphics gems*. 1990, pp. 395–396. 2

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (2014), 143:1–143:8. 1, 2, 3, 7

© 2025 The Author(s).