Martin Eisemann
eisemann@cg.tu-bs.de
Computer Graphics Lab, TU Braunschweig

Thorsten Grosch
tgrosch@mpi-inf.mpg.de
Max-Planck-Institut fuer Informatik, Saarbruecken

Stefan Mueller
stefanm@uni-koblenz.de
University of Koblenz-Landau

Marcus Magnor
magnor@cg.tu-bs.de
Computer Graphics Lab, TU Braunschweig

Fast Ray/Axis-Aligned Bounding Box Overlap Tests using Ray Slopes

Preprint

January 8, 2008

Computer Graphics Lab, TU Braunschweig

Abstract

The original paper [EGMM07] is available at http://jgt.akpeters.com.

This paper proposes a new method for fast ray/axis-aligned bounding box overlap tests. This method tests the slope of a ray against an axis-aligned bounding box by projecting itself and the box onto the three planes orthogonal to the world-coordinate axes and performs the tests on them separately. The method is division-free and successive calculations are independent of each other. No intersection distance is computed, but can be added easily. Testresults show the technique is up to 18% faster than any other method known to us and 14% faster on average for a wide variety of different test scenes and different processor architectures. Source code is available online.

Contents

Chapter 1

Introduction

Intersecting rays with axis-aligned bounding boxes (AABB) is a quite common task in computer graphics, e.g. in ray tracing systems using Bounding Volume Hierarchies (BVH)[GS87]. The simplest ray/AABB test computes intersection distances to each of the six planes defining the AABB , see [KK86] or [Hai89]. Three intervals are formed and they must overlap for the ray to overlap the AABB. Smits proposed a simplified, and therefore also faster, version of this "standard" test, which avoided some of numerical problems [Smi98]. An even more robust version of this algorithm was presented by Williams *et al.* [WBMS05]. Woo [Woo90] introduced a ray/AABB intersection test, which reduces the set of candidate planes from six to three, by ommitting the backfacing planes of the AABB. But, as stated by several authors, the test is not very suitable for modern processor architectures. Recently, Mahovsky and Wyvill [MW04, Mah05] used Plücker coordinates to test a ray against the silhouette of the AABB, whereas a projection onto a two-dimensional plane was not necessary. Their method is division-free, consists only of dot products and achieved impressive results when compared with different implementations of the standard test. The line segment/AABB overlap test by Gregory *et al.* [GLGT99] uses the Separating Axis Theorem to define whether a line segment and a box overlap. One could adapt this test for ray-BVH traversal, but in order to be efficient, the line segment has to be clipped at every AABB, which makes it inefficient for ray tracing. Therefore, we did not consider it in our tests.

In our method, instead of testing the ray in 3D against the AABB, we test it three times in 2D, once for each plane perpendicular to one of the world coordinate axes. Using the ray's slope, we can precompute most of the information needed for the intersection test. Our method is division-free and successive computations are independent of each other. It requires fewer operations than any other method tested and is easy to understand, since it reduces the problem to separate 2D cases. Here only slopes are compared, instead of a 3D problem, which may involve complex information,

like relative orientations of lines and 5D Pücker coordinates. Testresults have shown that our method is faster than any other method known to us, with a speed-up of up to 18%. The basic implementation does not compute an intersection distance, but this can be easily added and usually does not pose a problem, as pointed out in [Smi98].

Chapter 2

Ray-AABB Intersection

Recall that usually all six faces of an AABB are tested for intersection with a parametrically defined ray

$$\mathbf{r}(t) = \mathbf{o} + t\vec{\mathbf{d}} \tag{2.1}$$

consisting of an origin $\mathbf{o} = (o_x, o_y, o_z)$ and a direction vector $\vec{\mathbf{d}} = (d_x, d_y, d_z)$. This number can be reduced to three faces, if we know the signs of the direction components. Imagine a ray with all three components negative, it may pass through one of the faces FEHG, CGHD or BFGC (see Figure 2.1). The classification of the ray would be MMM, for "minus minus minus". The other seven ray classes, as used in [MW04], are MMP, MPM, MPP, PMM, PMP, PPM and PPP, with M and P corresponding to the signs of the direction components of d_x, d_y and d_z. Even though this is sufficient for the other methods, we need to extend this approach and add another ray class, called O. This is applied to every direction component of a ray which is 0. For example the direction vector $\vec{\mathbf{d}} = (1, 0, -1)$ would be classified as POM. This leads to a total of 26 different classes (actually 27, but we do not consider OOO, because it is degenerate). This might sound like a lot, but most of the cases seldomly occur. Even if they have to be tested, the computation can be simplified, for improved execution time. The test essentially needs to be performed only once per ray, since different traversal algorithms could be used, depending on the signs, similar to [Mah05].

An AABB B can be defined by its two extremal points $\mathbf{b}_{\min}$ and $\mathbf{b}_{\max}$ with $\mathbf{b}_{\min} = (x_0, y_0, z_0)$ and $\mathbf{b}_{\max} = (x_1, y_1, z_1)$ with $x_0 \leq x_1$, $y_0 \leq y_1$ and $z_0 \leq z_1$. To intersect ray $\mathbf{r}(t)$ with AABB B in 3D-space, we reduce the problem to intersecting the projected ray $\mathbf{r}(t)$ and AABB B in 2D-space. We must consider all three planes perpendicular to the three world coordinate axes, which are the xy-, xz- and yz-planes. If one of the three intersection tests fails, the ray misses the AABB. If all three succeed, the ray intersects the AABB. In the following we will just describe the algorithm for one of the three projection-planes (the xy-plane) and a PPP ray, since all other

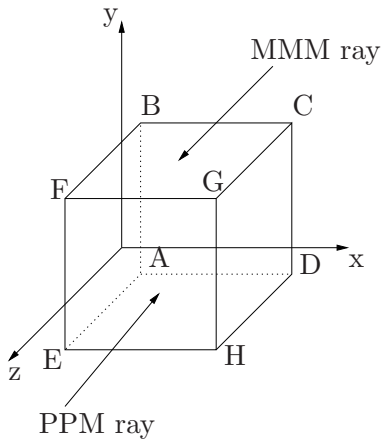

Figure 2.1: Axis-aligned bounding box pierced by an MMM ray and a PPM ray.

cases are equivalent (except for those which include a direction vector equal to 0 in one of its components, which we will describe later). For convenience the source code is available on the website listed at the end of the paper.

The slope of a parametrically defined ray $\mathbf{r}(t) = \mathbf{o} + t\vec{\mathbf{d}}$ in the xy-plane is

$$S_{\mathrm{xy}}(\mathbf{r}(t)) = d_y/d_x \quad . \tag{2.2}$$

For a PPP ray to intersect the AABB in the xy-plane, it has to intersect the linesegment passing from (x_0, y_1) to (x_1, y_0) (see Fig. 2.2). Therefore consider two rays $\mathbf{a}(t)$ and $\mathbf{b}(t)$ starting at the ray's origin and passing through these two points, with

$$\begin{aligned} \mathbf{a}(t) &= \mathbf{o} + t \begin{pmatrix} x_1 - o_{\mathrm{x}} \\ y_0 - o_{\mathrm{y}} \end{pmatrix} \quad , \\ \mathbf{b}(t) &= \mathbf{o} + t \begin{pmatrix} x_0 - o_{\mathrm{x}} \\ y_1 - o_{\mathrm{y}} \end{pmatrix} \end{aligned} \tag{2.3}$$

For a PPP ray $\mathbf{r}(t)$, $S_{\mathrm{xy}}(\mathbf{r}(t))$ needs to fulfill the following conditions, for the ray to miss the AABB:

$$S_{\mathrm{xy}}(\mathbf{r}(t)) < S_{\mathrm{xy}}(\mathbf{a}(t)) \Leftrightarrow S_{\mathrm{xy}}(\mathbf{r}(t)) < (y_0 - o_{\mathrm{y}})/(x_1 - o_{\mathrm{x}}) \quad , \tag{2.4}$$

or

$$S_{\mathrm{xy}}(\mathbf{r}(t)) > S_{\mathrm{xy}}(\mathbf{b}(t)) \Leftrightarrow S_{\mathrm{xy}}(\mathbf{r}(t)) > (y_1 - o_{\mathrm{y}})/(x_0 - o_{\mathrm{x}}) \quad , \tag{2.5}$$

This is depicted in Figure 2.2.

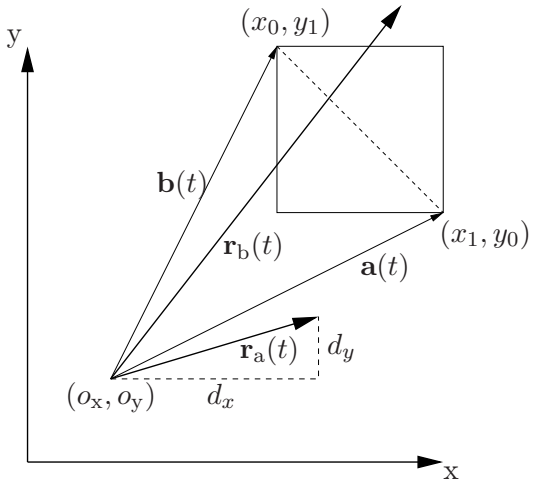

Figure 2.2: Ray $\mathbf{r}_\mathrm{a}(t)$ misses the AABB, due to its lesser slope than $\mathbf{a}(t)$, ray $\mathbf{r}_\mathrm{b}(t)$ hits the AABB in the xy-plane

Equation (2.4) holds for every ray classified as PPP with $o_x \leq x_1$ and $o_y \leq y_1$. Equation (2.5) only holds for those with $o_x < x_0$ and $o_y \leq y_1$, since otherwise the sign of the right part of equation (2.5) might change. To compensate for this we change the direction in which the slope is calculated. Instead of comparing $S_\mathrm{xy}(\mathbf{r}(t))$ we use

$$S_\mathrm{yx}(\mathbf{r}(t)) = d_x/d_y \tag{2.6}$$

for the second case, changing our condition from equation (2.5) to

$$S_\mathrm{yx}(\mathbf{r}(t)) < (x_0 - o_\mathrm{x})/(y_1 - o_\mathrm{y}) \tag{2.7}$$

If we determine that the AABB lies on the positive portion of the ray, by first testing $o_x < x_1$, $o_y < y_1$ and $o_z < z_1$, then equation (2.4) and (2.7) are sufficient to test a PPP ray $\mathbf{r}(t)$ for intersection with AABB B in its positive direction in the xy-plane.

The computation may be simplified by rearranging equations (2.4) and (2.7).

We show this exemplary for equation (2.4). Equation (2.7) is rearranged equivalently.

$$\begin{aligned} S_\mathrm{xy}(\mathbf{r}(t))(x_1 - o_\mathrm{x}) - y_0 + o_\mathrm{y} &< 0 \\ \Leftrightarrow S_\mathrm{xy}(\mathbf{r}(t))x_1 - y_0 + (o_\mathrm{y} - S_\mathrm{xy}(\mathbf{r}(t))o_\mathrm{x}) &< 0 \\ \Leftrightarrow S_\mathrm{xy}(\mathbf{r}(t))x_1 - y_0 + C_\mathrm{xy}(\mathbf{r}(t)) &< 0 \end{aligned} \tag{2.8}$$

Equation (2.8) does not look much simpler at first, but $S_\mathrm{xy}(\mathbf{r}(t))$ and $C_\mathrm{xy}(\mathbf{r}(t)) = o_\mathrm{y} - S_\mathrm{xy}(\mathbf{r}(t))o_\mathrm{x}$ can be precomputed when the ray is spawned.

Thus one simple multiplication, one subtraction, one addition and one comparison will reject a ray in the best case, if the AABB lies in the positive portion of the ray. We could even eliminate one more subtraction, by adding y_0 on both sides of the comparison, but practice showed that common C++ compilers created better code this way.

Hence, for a PPP ray `r` and an AABB `b` the complete algorithm can be elegantly formulated as follows:

```
1: bool intersectPPP(ray* r, aabb* b)
2: {
3:  if ((r->ox > b->x1) || (r->oy > b->y1) || (r->oz > b->z1)
4:    || (r->s_xy * b->x1 - b->y0 + r->c_xy < 0)
5:    || (r->s_yx * b->y1 - b->x0 + r->c_yx < 0)
6:    || (r->s_zy * b->z1 - b->y0 + r->c_zy < 0)
7:    || (r->s_yz * b->y1 - b->z0 + r->c_yz < 0)
8:    || (r->s_xz * b->x1 - b->z0 + r->c_xz < 0)
9:    || (r->s_zx * b->z1 - b->x0 + r->c_zx < 0))
10:   return false;
11:
12: return true;
13:}
```

The needed precomputations when a ray is spawned are the following:

```
1:  void make_ray(double x, double y, double z,
                  double i, double j, double k, ray *r)
2:  {
3:    r->x = x; // ray origin
4:    r->y = y;
5:    r->z = z;
6:    r->i = i; // ray direction
7:    r->j = j;
8:    r->k = k;
9:
10:   r->ii = 1.0/i; // inverse ray direction
11:   r->ij = 1.0/j;
12:   r->ik = 1.0/k;
13:
14:   r->s_yx = r->i * r->ij; // ray slopes
15:   r->s_xy = r->j * r->ii;
16:   r->s_zy = r->j * r->ik;
17:   r->s_yz = r->k * r->ij;
18:   r->s_xz = r->i * r->ik;
19:   r->s_zx = r->k * r->ii;
20:
21:   r->c_xy = r->y - r->s_xy * r->x; // precomputation
22:   r->c_yx = r->x - r->s_yx * r->y;
23:   r->c_zy = r->y - r->s_zy * r->z;
24:   r->c_yz = r->z - r->s_yz * r->y;
25:   r->c_xz = r->z - r->s_xz * r->x;
26:   r->c_zx = r->x - r->s_zx * r->z;
27: }
```

`r->ox`, `r->oy`, `r->oz` represent the ray origins, `r->dx`, `r->dy`, `r->dz` are the ray directions. `r->idx`, `r->idy`, `r->idz` are the inverse ray directions and `s_xy`, ..., `s_zx` denote the different ray slopes.

If one of the direction components of the ray, e.g. d_x, becomes equal to 0.0 the intersection code simplifies in the following way, line 4, 5, 8 and 9 can be replaced by checking if o_x is between the extents of the AABB x_0 and x_1. If two of the direction components are equal to 0.0 the intersection can by tested by simple comparisons between the ray's origin and direction and the extents of the AABB.

For simplicity of code, it would have been nice if we could exploit IEEE arithmetic to handle these additional cases. Unfortunately, this is not possible. Here is an example with the first slope condition

```
(r->s_xy * b->x1 - b->y0 + r->c_xy < 0)
```

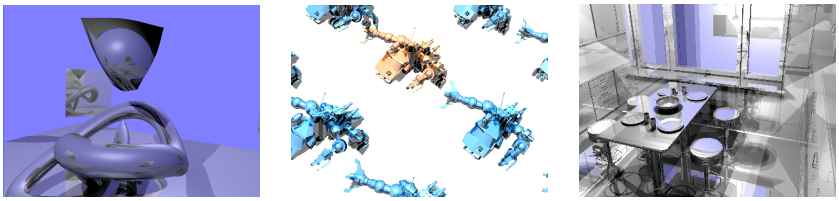

Figure 2.3: From left to right: BART Museum, RobotDance, BART Kitchen

, with

```
r->s_xy = r->j * r->ii;
r->c_xy = r->y - r->s_xy * r->x;
```

as described in the above source code. If `s_xy` $= \infty$, then the left hand side of the condition is either $+\infty$ or not a number (nan). In both cases the condition will be false, which is correct. But if one of the direction components becomes -0, we can derive a case, where the ray is assumed to miss the AABB, even though it hits it. For example if $\mathbf{r}(t) = (-1, 0, 0)^\top + t(-0, 1, 0)^\top$ and the AABB B is defined by $\mathbf{b}_{min} = (-2, -2, -2)$ and $\mathbf{b}_{max} = (2, 2, 2)$. Then `s_xy` $= -\infty$ and `c_xy` $= -\infty$. Therefore the condition will be true, and the ray is assumed to miss the AABB, even though its origin is clearly inside the AABB. Similar test cases can be found for every ray class.

The reader should note that no intersection distance is calculated in this code. If an intersection distance is needed, it can be added by computing the distances to the three faces of the AABB and using the largest of these, if an intersection was confirmed before. The three faces that need to be tested are already known due to the classification of the ray.

Even though the memory requirements in the ray structure increases due to the precomputation of slopes, using ray slopes to test for ray/AABB intersection has several advantages. The method is division-free if no intersection distance is needed (and even if it is needed, the inverses of the direction components can be used to avoid the divisions), and successive computations are independent of each other, allowing them to be computed in parallel.

2.1 Testresults and Conclusion

The ray slope method was compared to four other methods. The "standard" method [KK86], where essentially every face of the AABB was tested, the improved version [Smi98], its more robust version [WBMS05], and the Plücker method by Mahovsky and Wyvill [MW04]. For the last one we used precomputed ray classification and explicit saving of the needed Plücker coordinates, which was the fastest of their different versions in both their and

our tests. As a test bed we used the same framework as Mahovsky and Wyvill did in [MW04] and added ours and Williams' test.

To test the algorithm and the speed of the arithmetic instructions, we created 500,000 random ray/AABB pairs, which were saved in an array, so every method could use the same database. Hit ratios varied between 0%, 50% and 100%, to ensure that we have comparable times for low, medium and high hit frequencies. This is important since some algorithms contain certain termination criterias, allowing them to skip parts of the computation if the ray misses the AABB. All ray/AABB pairs in the array were tested and the whole process was repeated 100 times. This way prefetching techniques of modern compilers can be used to minimize the cache misses, which could otherwise influence the results. So every method performed 50,000,000 ray/AABB intersections in total. Tests were performed with both single- and double-precision arithmethic. Every ray/AABB intersection result for each method was tested for correctness before being used for timing, by comparing it with the result of the "standard" method. The test was performed on a Pentium4 2.4GHz processor. Table 2.1 shows the testresults. It is important that the reader only compares methods which compute the same results. E.g. a method which does not compute an intersection distance should only be compared to another method which does not compute an intersection distance, or methods using division for calculating the intersection distance should not be compared to methods using multiplication and so on.

The different types of tests are:

- slope: The method presented in this article. Returns `true` if the ray intersected the AABB, otherwise `false`. It does not compute an intersection distance.
- pluecker: The Plücker method determines whether the ray hits the AABB. It returns no intersection distance [MW04].
- standard: All six faces are tested for intersection and an intersection distance is computed. Explicitly tests for zero direction components [KK86].
- smits: The same as standard, but exploits IEEE arithmetic to handle direction components equal to zero, instead of explicit testing [Smi98].
- williams: The improved smits test, as presented in [WBMS05]
- int: If the suffix int is added, a separate test for computing the intersection distance is added. (This comes for free for the standard, smits and williams test)
- div: Divisions are used for the calculation of the intersection distance.

	double precision			single precision		
Hit-ratio	0.0	0.5	1.0	0.0	0.5	1.0
slope	8.58s	8.92s	9.22s	4.58s	4.69s	4.81s
slope_int_div	8.67s	9.42s	10.17s	4.49s	5.44s	6.38s
slope_int_mul	8.69s	9.27s	9.88s	4.48s	4.81s	5.13s
pluecker	8.99s	9.28s	9.63s	4.61s	5.14s	5.69s
pluecker_int_div	9.06s	9.92s	10.80s	4.52s	5.56s	6.63s
pluecker_int_mul	9.05s	9.53s	10.00s	4.50s	4.81s	5.11s
smits_div	13.97s	16.64s	19.28s	9.25s	11.28s	13.31s
smits_div_cls	12.70s	13.77s	14.81s	8.81s	10.45s	12.08s
smits_mul	11.38s	12.61s	13.86s	5.84s	6.86s	7.88s
smits_mul_cls	10.14s	10.53s	10.91s	6.09s	6.88s	7.66s
standard_div	14.03s	16.78s	19.55s	9.67s	11.72s	13.80s
standard_mul	11.58s	12.98s	14.42s	6.03s	7.17s	8.30s
williams	11.34s	13.45s	14.43s	5.59s	6.76s	7.74s
Speed-up	5%	4%	4%	0%	3%	6%

Table 2.1: Testresults for a Pentium 4 2.4GHz. 50,000,000 ray/AABB intersections are performed by every method. Times are measured in seconds. The speed-up shows the gain of using the fastest of our methods to using the fastest non ray slope method.

- mul: Multiplications are used for the calculation of the intersection distance.
- cls: If the suffix cls is added, a precomputed ray classification (MMM, etc.) is used, that is computed once when the ray is spawned, and carried in the ray, so that only three faces of the AABB need to be tested.

Notice that the ray slope method (slope) was faster than the fastest of the other methods (pluecker) in 6 out of 6 tests. In general our presented method should always be faster than the Plücker test [MW04], since both test the same edges but our method uses less arithmetic operations.

Since the first test is relatively susceptible to different processor features like branch prediction, we implemented the methods in our ray tracing system, to see if it would work as well in more realistic environments. The models in our test scenes are taken from [LAM01] and the complexities of the scenes range from 2,328 to 624,100 triangles. A recursion depth of two was used. Since we need an intersection distance for primary and secondary rays, we only compare the methods computing an intersection distance. Note that the code added to the slope and Plücker test for calculating the intersection distance is exactly the same. All other methods compute the distance implicitly, so no additional code was added. We used two different PCs for out tests. A Pentium Mobile 2GHz processor with 512MB RAM and an AMD Athlon 64 X2 Dual Core Processor 4800+ with 3GB of RAM, only

Method/Scene	Dragon	Robot	Museum	RobotDance	Kitchen
NumTris	2.328	6.241	75.546	624.100	110.540
Resolution	800x600	800x600	800x600	800x600	400x300
slope_int_div	3.96s	3.32s	13.02s	7.45s	27.46s
slope_int_mul	3.74s	3.11s	12.25s	6.98s	26.56s
pluecker_int_div	4.42s	3.70s	14.69s	8.45s	34.79s
pluecker_int_mul	4.19s	3.49s	13.87s	8.05s	30.72s
smits_div	5.76s	4.73s	20.02s	11.27s	43.68s
smits_div_cls	5.26s	4.33s	18.35s	10.27s	39.99s
smits_mul	4.49s	3.70s	15.08s	8.53s	32.58s
smits_mul_cls	4.27s	3.53s	14.43s	9.33s	32.10s
standard_div	6.03s	4.92s	20.82s	11.77s	45.37s
standard_mul	4.80s	3.97s	16.42s	10.70s	38.77s
williams	4.64s	3.78s	15.94s	8.81s	36.00s
Speed-up	12%	12%	13%	15%	16%

Table 2.2: Rendering results for several test scenes, rendered on a Pentium Mobile 2GHz, 512MB RAM. Times are measured in seconds. The speed-up shows the gain of using the fastest of our methods (slope_int_mul) to using the fastest non ray slope method.

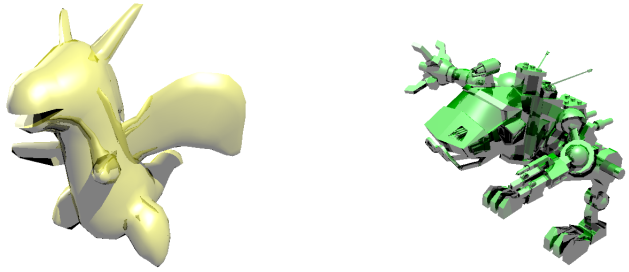

Figure 2.4: Left: Dragon, right: SingleRobot

one core was used for the tests. Testresults are shown in Table 2.2 and 2.3. The last row shows the average, percentage speed-up of the slope_int_mul compared to the fastest non ray slope method. Both ray slope methods achieved best rendering results in 5 out of 5 tests with a speed-up of up to 18%. A minimum speed-up of 12% was achieved in all tests, with an average speed-up of about 14% on the Pentium Mobile and 15% on the AMD Athlon compared to the second fastest non ray slope method. This also shows the high consistency of our method.

Acknowledgments. We would like to thank Jeffrey Mahovsky and Brian Wyvill for providing us with their testing framework from [MW04]. And we would also like to thank Jonas Lext, Ulf Assarsson and Tomas Akenine-Möller for providing

Method/Scene	Dragon	Robot	Museum	RobotDance	Kitchen
NumTris	2.328	6.241	75.546	624.100	110.540
Resolution	800x600	800x600	800x600	800x600	400x300
slope_int_div	2.63s	2.42s	8.62s	5.66s	19.58s
slope_int_mul	2.49s	2.35s	8.38s	5.61s	19.44s
pluecker_int_div	3.05s	2.79s	10.30s	6.62s	24.23s
pluecker_int_mul	2.92s	2.75s	10.02s	6.46s	23.49s
smits_div	3.15s	2.82s	10.81s	6.80s	25.41s
smits_div_cls	3.19s	2.84s	11.19s	6.74s	26.30s
smits_mul	2.92s	2.64s	9.89s	6.28s	23.01s
smits_mul_cls	2.90s	2.76s	10.03s	6.37s	22.82s
standard_div	3.86s	3.39s	12.87s	8.26s	30.83s
standard_mul	3.28s	2.92s	10.95s	7.09s	25.54s
williams	3.12s	3.06s	11.88s	7.32s	26.62s
Speed-up	16%	12%	18%	12%	17%

Table 2.3: Rendering results for several test scenes, rendered on an AMD Athlon 64 X2 Dual Core Processor 4800+ (only one core used). Times are measured in seconds. The speed-up shows the gain of using the fastest of our methods (slope_int_mul) to using the fastest non ray slope method.

us with the 3D-Models.

Chapter 3

Web Information:

Sample C++ source code is available online at
http://www.cg.cs.tu-bs.de/people/eisemann/rayslope/rayslope.zip.

Bibliography

[EGMM07] Martin Eisemann, Thorsten Grosch, Marcus Magnor, and Stefan Mueller. Fast Ray/Axis-Aligned Bounding Box Overlap Tests using Ray Slopes. *journal of graphic tools*, 12(4), 12 2007.

[GLGT99] A. Gregory, M. Lin, S. Gottschalk, and R. Taylor. H-COLLIDE: A Framework for Fast and Accurate Collision Detection for Haptic Interaction. In *Proceedings of Virtual Reality Conference*, pages 38–45, 1999.

[GS87] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics & Applications*, 7(5):14–20, May 1987.

[Hai89] E. Haines. *An Introduction to Ray Tracing*, chapter Essential Ray Tracing Algorithms, pages 33–77. Academic Press Ltd., London, UK, UK, 1989.

[KK86] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM Press.

[LAM01] J. Lext, U. Assarsson, and T. Möller. A Benchmark for Animated Ray Tracing. *IEEE Computer Graphics and Applications*, 21(2):22–31, March 2001.

[Mah05] J. Mahovsky. *Ray Tracing With Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, 2005.

[MW04] J. Mahovsky and B. Wyvill. Fast Ray-Axis Aligned Bounding Box Overlap Tests With Plücker Coordinates. *Journal of Graphics Tools*, 9(1):35–46, 2004.

[Smi98] B. Smits. Efficiency Issues for Ray Tracing. *Journal of Graphic Tools*, 3(2):1–14, 1998.

[WBMS05] A. Williams, S. Barrus, R. Keith Morley, and P. Shirley. An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of Graphic Tools*, 10(1):49–54, 2005.

[Woo90] Andrew Woo. *Fast Ray-Box Intersection*, pages 395–396. Academic Press Professional, Inc., San Diego, CA, USA, 1990.