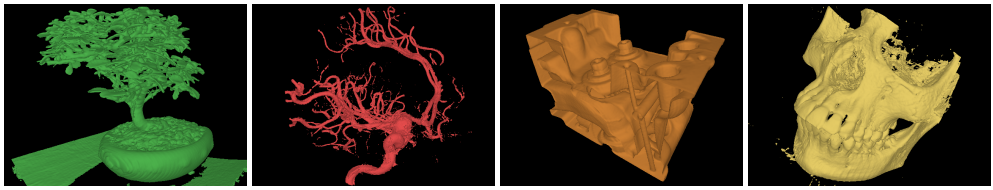
Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing

Gerd Marmitt*, Andreas Kleer*, Ingo Wald†, Heiko Friedrich*, and Philipp Slusallek*

*Computer Graphics Group, Saarland University †MPII Informatik



A new intersection algorithm for iso-surface volume ray tracing increases performance by roughly a factor of three compared to previous accurate intersection algorithms while offering similar performance as previous algorithms that only approximated the solution. The image shows some of the test data sets: Bonsai tree, aneurism, engine, and skull.

Abstract

Visualizing iso-surfaces of volumetric data sets is becoming increasingly important for many practical applications. One crucial task in iso-surface ray tracing is to find the correct intersection of a ray with the trilinear-interpolated implicit surface defined by the data values at the vertices of a given voxel. Currently available solutions are either accurate but slow or they provide fast but only approximate solutions.

In this paper, we analyze the available techniques and present a new intersection algorithm. We compare and evaluate the new algorithm against previous approaches using both synthetic test cases and real world data sets.

The new algorithm is roughly three times faster but provides the same image quality and better numerical stability as previous accurate solutions.

1 Introduction

Volume rendering is an important addition to the more common surface rendering methods. It visualizes data values that are provided at samples distributed in 3D space. While *direct volume rendering* [Lev90] computes the interaction of light with the volume along the entire length of a viewing ray, *iso-surface rendering* first computes one or more surfaces through an implicit function defined on the volume data values. This surface is then shaded using normal surface shading techniques. In this paper we consider only the latter case of isosurface rendering.

Isosurface rendering allows for transparently integrating volume primitives into a surface rendering system. In particular we aim at interactively rendering these iso-surfaces by full ray tracing. Ray tracing offers the advantage that the surface must never be computed and represented explicitly [PSL+98] as commonly done with the marching cube algorithm [LC87] for rendering on rasterization devices. With ray tracing the necessary ray-surface intersections are computed on the fly at all relevant voxels, which allows for instantly changing the iso-value without the need for a costly re-computation of an iso-surface representation.

Furthermore, we target full ray tracing for the ability to fully integrate volume rendering with standard surface rendering, including advanced effects such as shadows, reflections, and many others.

Because data values are only provided at dicrete locations in space, values for in-between points are usually derived through trilinear interpolation of the data values provided at the vertices of each voxel. As a result the function is a cubic polynomial along

* {marmitt,kleer,heiko,slusallek}@graphics.cs.uni-sb.de
† wald@mpi-sb.mpg.de

any straight line or ray passing through the voxel. The required ray-surface intersection is then given as the first non-negative root of this polynomial along the ray.

More accurate and smoother interpolation methods do exist, e.g. [RZNS04], but they are significantly more expensive to compute, which makes them less suited for the interactive applications targeted here.

In this paper we concentrate on the intersection computation between rays and the voxel data and ignore the issue of efficiently locating those voxels in the volume data set that may contain the surface. For efficient traversal of isosurface data sets see e.g. [PSL$^+$98].

The following presentation is organized as follows: We first present previous work on accurate intersection methods in Section 2 before looking at faster but approximate solution in Section 3. Our new algorithm is then presented in Section 4. We compare the quality and evaluate the performance of the different intersection methods in Section 5 before concluding and suggesting future work in Section 6.

2 Accurate Intersection Method

Given a cell with the data values ρ_{ijk} $(i,j,k \in \{0,1\})$ at its eight vertices, the density ρ at any point $(u,v,w) \in [0,1]^3$ can be computed by trilinear interpolation, i.e.

$$\rho(u,v,w) = \sum_{i,j,k \in \{0,1\}} u_i v_j w_k \rho_{ijk},$$

where $u_0 = u$, $u_1 = 1-u$, $v_0 = v$, etc (see [Shi02]). If the spatial location of this cell is $V = [x_0..x_1] \times [y_0..y_1] \times [z_0..z_1]$, then $\rho(p)$ of any three-dimensional point $p \in V$ can be computed by first transforming p to the unit coordinate system, yielding

$$p_0 = (u_0^p, v_0^p, w_0^p) = (\frac{x_1 - p_x}{x_1 - x_0}, ...).$$

Using this notation, for a ray $R(t) = a + tb$ with origin a and direction b, which overlaps the voxel V in the interval t_{in} and t_{out}, the density $\rho(t) = \rho(R(t))$ for each point on the interval is defined as

$$\rho(t) = \sum_{i,j,k \in \{0,1\}} (u_i^a + t u_i^b)(v_j^a + t v_j^b)(w_k^a + t w_k^b).$$

Expanding then yields a cubic polynomial

$$\rho(t) = At^3 + Bt^2 + Ct + D$$

whose coefficients (see [PSL$^+$98, Shi02]) are

$$A = \sum_{i,j,k} u_i^b v_j^b w_k^b \rho_{ijk},$$

$$B = \sum_{i,j,k} \left(u_i^a v_j^b w_k^b + u_i^b v_j^a w_k^b + u_i^b v_j^b w_k^a \right) \rho_{ijk},$$

$$C = \sum_{i,j,k} \left(u_i^b v_j^a w_k^a + u_i^a v_j^b w_k^a + u_i^a v_j^a w_k^b \right) \rho_{ijk},$$

$$D = \sum_{i,j,k} u_i^a v_j^a w_k^a \rho_{ijk.}$$

Finding the intersection of the ray with the implicitly defined iso-surface $\rho(t) = \rho_{iso}$ then amounts to determining the smallest $t \in [t_{in}, t_{out}]$ for which the polynomial

$$f(t) := \rho(t) - \rho_{iso}$$

is zero, i.e. we are looking for the smallest root of f in the interval $[t_{in}, t_{out}]$.

2.1 Schwarze's Analytic Inversion

Given the cubic polynomial in the ray parameter the most commonly used method for analytically computing the intersection for iso-surface rendering is the Schwarze approach [Sch90, PSL$^+$98]. This method first checks for special cases (small coefficients A and/or B) where the polynomial does not have full degree and solves those more directly. But even the special case of a quadratic polynomial already involves a costly square root operation.

The most common case of a full cubic polynomial is solved using Cardano's formula involving several cosines and even more costly inverse cosine operations. Furthermore, we need to compute *all* roots to locate the first one along the ray that is within the current voxel.

As usual, this algebraic solution is prone to numerical problems. Our target of interactive volume rendering furthermore suggests the use of single precision floating point values, which makes these issues even worse with respect to numerical stability. Consequently, it is difficult to tune this approach in order to completely avoid incorrectly computed intersections.

While the Schwarze approach is commonly applied and is mathematically the most accurate solution for computing intersection in volume ray tracing, in practice it has too many drawbacks.

3 Approximate Methods

In contrast to this correct analytic methods approximative methods trade quality for speed by not accurately computing the intersection with the cubic polynomial.

3.1 Simple Midpoint Algorithm

The most simplistic algorithm consists of assuming an intersection for every voxel where the iso-value is within the range of data values at the vertices. In this case the intersection point is usually set to the mid point between the ray's entry and exit point.

While this method is fast it obviously leads to blocky artifacts of the size of voxels. We thus use this method only as a reference for performance comparisons.

3.2 Linear Interpolation

A still simplistic, but already much more useful approach is to assume a linear function within a voxel and to simply interpolate the intersection point from the iso-values at the respective entry and exit points of the ray, $\rho(t_{in})$ and $\rho(t_{out})$.

Algorithm 1 Pseudo code for linear interpolating the intersection position.

```
// linear interpolation
ρ_in := ρ(R(t_in)); ρ_out := ρ(R(t_out))
if sign(ρ_in − ρ_iso) = sign(ρ_out − ρ_iso) then
    return NO_HIT
end if
return t_hit := t_in + (t_out − t_in) (ρ_iso − ρ_in)/(ρ_out − ρ_in)
```

Note that this approach is already significantly more costly than the midpoint algorithm, since it requires two costly trilinear interpolations for computing $\rho(t_{in})$ and $\rho(t_{out})$. Though these values could also be computed with bilinear interpolation on the voxel side, we compute it via full trilinear interpolation, since this code is not slower than first determining the correct input values for the bilinear interpolation. More importantly only the general trilinear interpolation is suitable for implementing in ray-parallel SIMD code, where different rays might require interpolations from different vertices.

Linear interpolation provides a good approximation in many cases where the function is close to linear. However, it will obviously fail to find the correct intersection in more complex cases, such as if the function has *two* roots, in which case the entry and exit densities are either both larger than ρ_{iso}, or both are smaller, and no intersection is found at all.

3.3 Neubauer's Method

Neubauer at al. [NMHW02] suggested a method that uses *repeated linear interpolation*. This method builds on the linear interpolation just described but refines the results by looking at the trilinearly interpolated data value at the intersection point. This intersection point splits the ray segment within the voxel into two parts. Based on the data value at the computed intersection this approach chooses to recursively apply the linear interpolation to that segment that contains the iso-value.

Typically, this approach is applied a fixed number of times (2-3), even though an adaptive termination criterion is equally possible.

Algorithm 2 Pseudo code for Neubauer's algorithm using repeated linear interpolation.

```
// Neubauer: repeated linear interpolation
t_0 := t_in; t_1 := t_out
ρ_0 := ρ(R(t_0)); ρ_1 := ρ(R(t_1))
if sign(ρ_0 − ρ_iso) = sign(ρ_1 − ρ_iso) then
    return NO_HIT
end if
for i=1..N do
    t := t_0 + (t_1 − t_0) (ρ_iso − ρ_0)/(ρ_1 − ρ_0)
    if sign(ρ(R(t)) − ρ_iso) = sign(ρ_0 − ρ_iso)
    then
        t_0 := t; ρ_0 = ρ(R(t))
    else
        t_1 := t; ρ_1 = ρ(R(t))
    end if
end for
return t_hit := t_0 + (t_1 − t_0) (ρ_iso − ρ_0)/(ρ_1 − ρ_0)
```

Unfortunately this approach suffers from similar problems as the previous technique in that it some-

times fails to locate valid intersections. Nonetheless, in those cases where it does correctly identify them at all it computes them more accurately. A new failure case appears in that the approach can falsely return the last intersection point in the case that three intersections are contained within a voxel but two of them are within the first ray segment.

4 Isolation and Iterative Root Finding

Roughly speaking, we have so far discussed methods that are either slow and correct, or fast and sometimes incorrect. Therefore, we have derived a new algorithm, that aims at being as fast as the Neubauer method, but as correct as the Schwarze method. Our new intersection method is based on two key observations:

- We are only interested in the first intersection with the implicit function and there is no need to compute all intersections as in the case of Schwarze.
- Repeated linear interpolation *does* find the correct root fast and reliably *if* the start interval for the iteration contains *exactly* one root.

In our approach we therefore first isolate the roots by computing the extrema of the polynomial. This requires solving a simple quadratic equation. These two extrema then split the ray segment into at most three parts.

We step through these segments from front to back, computing the data values at its start and end point from the cubic polynomial, which is pretty efficient once the polynomials coefficients are known. Once the interval is found, we can guarantee that it (a) contains (exactly) one root (f is continuous, contains zero, and does not have extrema in that interval), (b) that the root lies in the interval $[t_{in}, t_{out}]$, and (c) it is the first root in this interval.

After the interval is found, we locate the root using repeated linear interpolation (as in Neubauer) or simply via recursive bi-section. Again, it is usually faster to apply a fixed number of 2-3 iterations than to apply an adaptive termination criterion.

Since we already know the coefficients of the polynomial we can quickly and efficiently compute any data value along the ray. This does no longer involve any costly trilinear interpolation. This advantage comes at the cost for computing the coefficients and the extrema in the first place.

As shown below this approach is roughly a factor of three faster than the Schwarze code while providing the same guarantees on correctness and even better numerical stability. It is also well suited for a data parallel SIMD implementation.

Algorithm 3 Pseudo code for the new intersection algorithm.

$t_0 = t_{in};\ t_1 = t_{out};\ f_0 = f(t_0);\ f_1 = f(t_1)$
// Find extrema by looking at $f'(t) = 3At^2 + 2Bt + C$
if $f'\ has\ real\ roots$ **then**
  $e_0 = smaller\ root\ of\ f'$
  if $e_0 \in [t_0, t_1]$ **then**
    if $sign(f(e_0)) = sign(f_0)$ **then**
      // Advance the ray to the second segment
      $t_0 := e_0;\ f_0 := f(e_0)$
    else
      $t_1 := e_0;\ f_1 := f(e_0)$
    end if
  end if
  $e_1 = second\ root\ of\ f'$
  if $e_1 \in [t_0, t_1]$ **then**
    if $sign(f(e_1)) = sign(f_0)$ **then**
      // Advance the ray to the third segment
      $t_0 := e_1;\ f_0 := f(e_1)$
    else
      $t_1 := e_1;\ f_1 := f(e_1)$
    end if
  end if
end if
if $sign(f_0) = sign(f_1)$ **then**
  return NO_HIT;
end if
// now, know we've got a root in t_0, t_1
// find it via repeated linear interpolation
for i=1..N **do**
  $t := t_0 + (t_1 - t_0)\frac{-f_0}{f_1 - f_0}$
  if $sign(f(R(t))) = sign(f_0)$ **then**
    $t_0 := t;\ f_0 = f(R(t))$
  else
    $t_1 := t;\ f_1 = f(R(t))$
  end if
end for
return $t_{hit} := t_0 + (t_1 - t_0)\frac{-f_0}{f_1 - f_0}$

4.1 Parallel SIMD Implementation

Wald et al. [WSBW01] already demonstrated the performance advantage of exploiting the SIMD instructions on today's processors using a data parallel approach, where the computations are performed on multiple rays in parallel. Modern SIMD instruction sets allow for operating on up to four floating values in a single instruction, which can improve performance significantly if the algorithm are SIMD friendly.

However, this is not the case for the Schwarze intersector due to its complex control flow for handling special cases and the evaluation of complex trigonometric functions. The repeated linear interpolation approach from Neubauer is better suited for a SIMD implementation but still needs to handle the case of inconsistent decisions of which ray segment needs to be handled in the next iteration.

Our new intersection technique is based on Neubauer's algorithm but adds a SIMD friendly computation of the polynomial's coefficients at the beginning. Stepping through the segments is less suited for SIMD computation because it needs to evaluate both possible cases for each of the tests (see Algorithm 3) and use conditional assignement to record the results. Even though, this adds only little SIMD overhead and still results in fast SIMD compuations as shown below.

5 Experiments and Results

In the following we compare the different volume intersection algorithms in terms of performance and visual quality.

For all experiments, we have used an AMD Opteron Processor running at 1.8 GHz. Performance data has been collected by timing the call to the intersection functions. As the timing overhead may dominate the total execution time, the intersection kernel is being called several thousand times between for better accuracy.

In order to measure the performance of the different methods, we performed two different experiments. In the first setup, we used purely synthetic data rendered at 256^2 resolution in which we fed the different algorithms with seven typical marching cubes configurations, with the voxel corners being either 0 or 1, and the iso-value used for intersection as $\rho_{iso} = 0.5$. For each of these configurations, we computed the minimum, maximum, and average execution times.

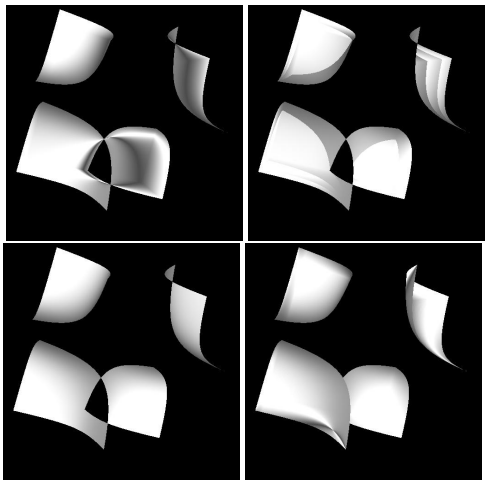


Figure 1: Test scene for testing how the various algorithms handle the special case where a ray has more than one intersection with the iso-surface. top left: linear, top right: Neubauer (2 iterations), bottom left: Neubauer (10 iterations), bottom right: accurate cubic intersection (Schwarze and the new algorithm provide the same results). As can be seen, only the cubic algorithms correctly handle all cases. The shading artifacts are due to incorrect normals in this test case and have no influence on the results.

In the second setup, we have integrated all intersection kernels into a realtime iso-surface ray tracing system [Wal04] rendering images at 640×480 resolution. As before, each kernel is called several times in order to factor out any influence of the measurement procedure. This procedure is then applied to several real-world data sets. These results should be general enough to also apply to other data sets and application areas.

5.1 Comparison of Accuracy

Figure 1 shows the artifacts produced by the different approximative algorithms. It clearly demonstrates that the approximate algorithms produce significant errors for some of the more complex voxel configurations. While these cases are not too frequent, we would like to avoid these problems using a fully accurate but still fast algorithm that can be used in an interactive context.

5.2 Comparison of Performance

The performance for the different synthetic voxel configurations are almost identical so that we only present the aggregate results over all seven measured configurations (see Table 1). On average we can compute roughly three million intersection per second for these test cases. The min and max values are due to the different execution times for different ray and voxel configurations.

For the SIMD versions we perform the same measurements but have to account for the fact that multiple rays are processed in parallel for each intersection. The results (see Table 2) show that the best performance improvements can be realized for the low-quality midpoint and linear algorithms (by a factor of roughly 3 and 2, respectively). The only other algorithm that benefits significantly is the new intersection algorithm, which clearly shows that the complex control flow conflicts strongly with the performance gain through SIMD computations. For the new algorithm we still see an increase in performance by about 50%.

Performance in Real-World Datasets

For the synthetic test cases we selected some of the more complex cases, which are not representative. More interesting and practically relevant are the performance measurements when using commonly used data sets. For this we have selected the Bonsai Tree, the Aneurism and the Engine data set (see Table 3).

For these realistic data sets we see that the new algorithm shows a speedup between roughly a factor 1.7 and 1.9 compared to the Schwarze algorithm that generates images of the same high quality. The new algorithm has similar performance than the approximate Neubauer method but always generated accurate images without artifacts.

Again, we were also interested in checking our SIMD implementation with real data sets (see Table 4). For the Neubauer code we see a strong performace improvement by a factor between 2.3 and 2.6 compared to the sequential code. The Neubauer code seems particularly well suited to some configurations that were not in our synthetic test suite.

For our new algorithm the improvement is smaller due to the more complex control flow but we still achieve about 60% higher performance through the use of SSE. With the SSE implementation of the new accurate algorithm we consistently achieve about 60% of the performance of the approximate Neubauer algorithm for these realistic data sets. However, we still have the advantage that we compute the correct intersection in all cases.

intersector	correct	min	max	avg
midpoint	-	1.28	33.3	26.88
linear	-	0.46	7.54	6.84
Neubauer	-	0.56	3.18	3.04
Schwarze	+	0.44	3.26	3.03
New	+	0.57	3.23	2.92

Table 1: Performance comparison of the different ray-voxel intersection techniques, for synthetic experimental setups, measured in million ray-voxel intersections on a 1.8GHz AMD Opteron CPU. Data for real-world data set are given in Table 3.

intersector	correct	min	max	avg
midpoint	-	2.59	98.41	85.48
linear	-	1.17	13.93	13.35
Neubauer	-	0.19	8.01	3.04
Schwarze	+	0.24	2.20	1.98
New	+	0.85	4.80	4.35

Table 2: SIMD performance of the different ray-cell intersection techniques for synthetic experimental setups, measured in million ray-cell intersections. We see a strong improvement for the new code while the Schwarze code actually gets slower.

Method	correct	bonsai	aneurism	engine
midpoint	-	26.21	26.18	26.22
linear	-	6.65	6.65	6.68
neubauer	-	2.93	2.94	2.94
Schwarze	+	1.60	1.56	1.48
New	+	2.76	2.80	2.73

Table 3: Single ray intersection performance for real-world scenes. As for the synthetic datasets, the new method provides significantly better performance than approximative techniques while being as accurate as the much slower Schwarze method.

Method	correct	bonsai	aneurism	engine
midpoint	-	87.71	87.12	87.13
linear	-	13.68	13.66	13.67
Neubauer	-	7.65	7.60	6.94
Schwarze	+	1.49	1.39	1.44
New	+	4.37	4.56	4.39

Table 4: SIMD intersection performance for real-world scenes. Again, our new algorithm is significantly faster than the Schwarze method, but computes the same correct results.

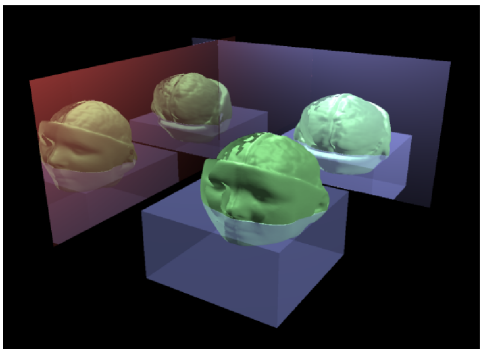


Figure 2: A test scene showing mixed surface and iso-surface volume rendering interactively with roughly 2 frames per second at 640×480 pixels on a dual Intel Pentium-4 2.2 GHz system.

6 Conclusions and Future Work

In this paper we have investigated the use of different ray-voxel intersection algorithms for iso-surface volume rendering using ray tracing. These algorithms differ in their achieved accuracy and performance, as well as their suitability for a SIMD implementation.

In particular we have compared the accurate but slow and numerically unstable Schwarze code against the fast approach by Neubauer, which is prone to approximation artifacts. As an alternative we developed a new approach that integrates the advantages of both methods, providing the most accurate results with the best performance.

In contrast to the Schwarze code both the new and the Neubauer code benefit significantly from the use of SIMD processing. Due to its simpler control flow, the Neubauer algorithm achieves the best performance for SIMD operations but is prone to image artifacts. Our new algorithm clearly outperforms the Schwarze algorithm while providing the same correct image.

All discussed algorithms have been integrated into the OpenRT realtime ray tracing system [Wal04], where they provide fully integrated volume and surface rendering. In this system an iso-surface volume is just another surface primitive and all applications and advanced rendering techniques can deal with them without any issues. Figure 2) shows a volume data set rendered with reflections and transparency via surface primitives. Even the recently developed techniques for realtime computation of global illumination [BWS03] could now be applied also to volume objects.

In the near future, we will investigate how to accelerate higher-order interpolation techniques such as the one proposed by Rössl et al. [RZNS04]. Finally, it seems interesting to investigate special optimizations for time-varying datasets.

References

[BWS03] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum*, 22(3):621–630, 2003. (Proceedings of Eurographics).

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21(4):163–169, July 1987.

[Lev90] Marc Levoy. Efficient Ray Tracing for Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[NMHW02] A. Neubauer, L. Mroz, H. Hauser, and R. Wegenkittl. Cell-based first-hit ray casting. In *Proceedings of the Symposium on Data Visualisation 2002*, pages 77–ff, 2002.

[PSL+98] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, pages 233–238, October 1998.

[RZNS04] Christian Rössl, Frank Zeilfelder, Günther Nürnberger, and Hans-Peter Seidel. Reconstruction of volume data with quadratic super splines. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):397–409, 2004.

[Sch90] Jochen Schwarze. Cubic and Quartic Roots. In Andres Glassner, editor, *Graphics Gems*, pages 404–407. Academic Press, 1990. ISBN: 0122861663.

[Shi02] Peter Shirley. *Fundamentals of Computer Graphics*. A K Peters, 2002. ISBN 1-56881-124-1.

[Wal04] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).