On real-time ray tracing

Ville Rahikainen

University of Tampere
School of Information Sciences
Computer Science
M.Sc. Thesis
Supervisor: Martti Juhola
May 2013

University of Tampere
School of Information Sciences
Computer Science
Ville Rahikainen: On real-time ray tracing
M.Sc. Thesis, 67 pages, 3 appendix pages
May 2013

Rendering of increasingly complex and detailed objects and scenes, with physically correct light simulation, is an important problem for many fields ranging from medical imaging to computer games. While even the latest graphics processing units are unable to render truly massive models consisting of hundreds of millions of primitives, an algorithm known as ray tracing – which by its very nature approximates light transport – can be used to solve such problems. Ray tracing is a simple but powerful method known to produce high image quality, but it is also known for its slow execution speed. This thesis examines parts of the research made to bring ray tracing into the interactive sphere. Specifically, it explores ray-triangle intersections, ray coherency, as well as kd-tree building and traversal. Even though these issues are delved into in the context of interactive graphics, the insights provided by the analyzed literature will also translate to other domains.

Key words and terms: ray tracing, kd-tree

Contents

1. Introduction

Interactive 3D computer graphics have seen a tremendous improvement in quality since their inception. The visual realism and image complexity of today's state-of-the-art applications are superior even to anything done just six years ago. With the ongoing demand for higher detail from different industries this trend is certain to continue. Figure 1.1 presents a screenshot from an early video game and Figure 1.2 presents a screenshot from a game from 2007.

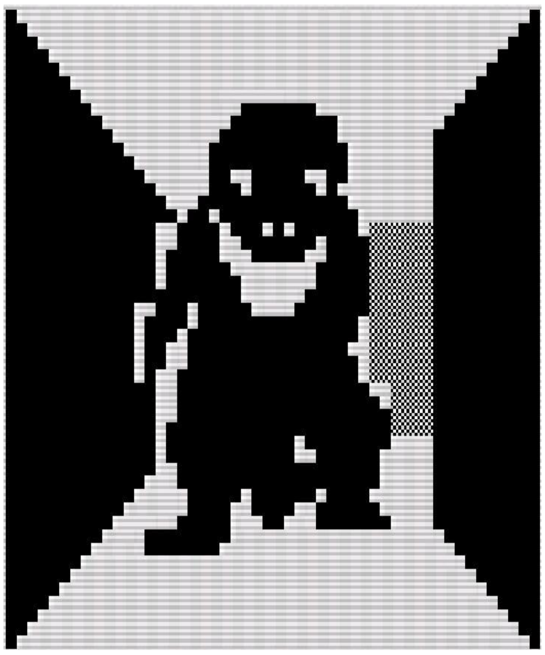
Figure 1.1 Screenshot from 3D Monster Maze

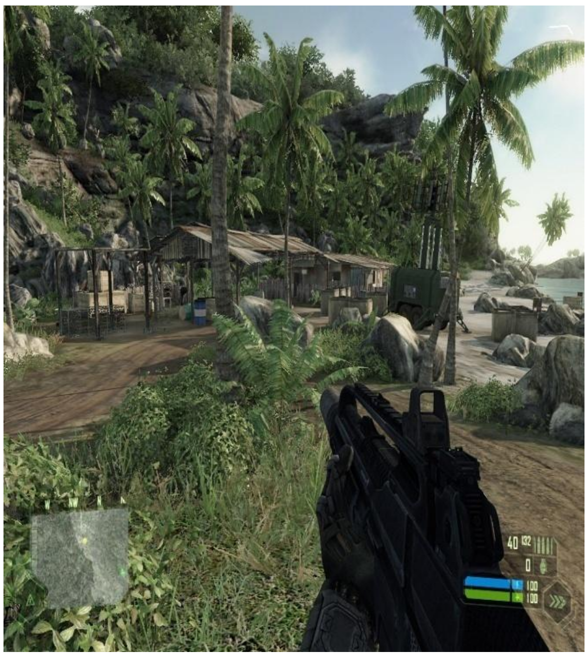
Figure 1.2 Screenshot from Crysis

The z-buffer algorithm – first introduced by Wolfgang Strasser, but independently discovered and generally attributed to Catmull [1974] – is used virtually exclusively to solve the visibility problem in interactive computer graphics. It has proprietary hardware known as GPU (Graphics Processing Unit) which makes the present day graphical quality possible and which has been specifically designed to execute the z-buffer algorithm fast. Z-buffer is capable of producing spectacularly realistic images, but it is not without its problems. Without culling its runtime is $O(N)$ or asymptotically linear compared to the number of primitives, which means that the quest for higher visual realism and more complex scenes will continue to make its execution slower. Furthermore, z-buffer has problems with many basic effects which are essential for a truly realistic image, such as shadows, reflections and refractions. To be sure, these effects can be done, but with limitations and much work. In addition, z-buffer does not handle well objects which do not purely consist of triangles [Shirley *et al.*, 2008]. Despite these shortcomings z-buffer has been without a viable alternative for decades.

However, the same factors – continued rise of processing power and memory combined with ever better GPUs – which have made it possible for z-buffer to handle increasingly detailed and complex scenes, now promise to make another algorithm capable of real-time execution in the foreseeable future. Building on decades of off-line rendering work, the last decade or so has seen a flurry of research into interactive or real-time ray tracing, an algorithm which has long been used in the creation of photorealistic images. The basic idea of ray tracing is to simulate nature, light arrives from some source(s) – be it the sun, light bulb, TV or a bioluminescent growth, known as esca or illicium, of an anglerfish – and it is reflected, refracted and shadows are formed to where it does not shine directly. There are two ways to do ray tracing – forward and backward. In forward ray tracing each light source shoots out rays in all directions and all of the rays are traced to see whether or not they reach the image plane and continue on to the eye – eye is the point from which the scene is viewed. This approach is computationally very inefficient since even the rays that do not contribute anything to the image are traced. Due to forward ray tracings' infeasibility in image production, ray tracing refers nearly exclusively to backward ray tracing [Glassner, 1989a].

Backward ray tracing starts by turning the approach around. Since we know that the rays which reach the eye certainly contribute to the image, we can trace them backwards from the eye. In different studies these rays have been called with a variety of names, such as eye rays, pixel rays, primary rays and incident rays. In this study they will be referred to as eye rays. Once a ray reaches an object in the scene a few things can happen. In the most basic form of ray tracing, known as ray casting, new rays are spawned towards each light source from the point of intersection between a ray and an object. These rays are known as shadow rays, shadow feelers or illumination rays, in this study they will be referred to as shadow rays. If the shadow ray reaches the light source then this object is being illuminated by this light source. On the other hand, if the ray does not reach the intended light source and instead reaches another – occluding – object, then the object from which the shadow ray starts is not illuminated by this light source but is in the shadow of the other object [Appel, 1968; Glassner, 1989a; Foley *et al.*, 1990]. This is not exactly how illumination would work in nature. Further realism can be achieved by using more computational power to simulate indirect light [Foley *et al.*, 1990]. Figure 1.3 illustrates backward ray tracing.

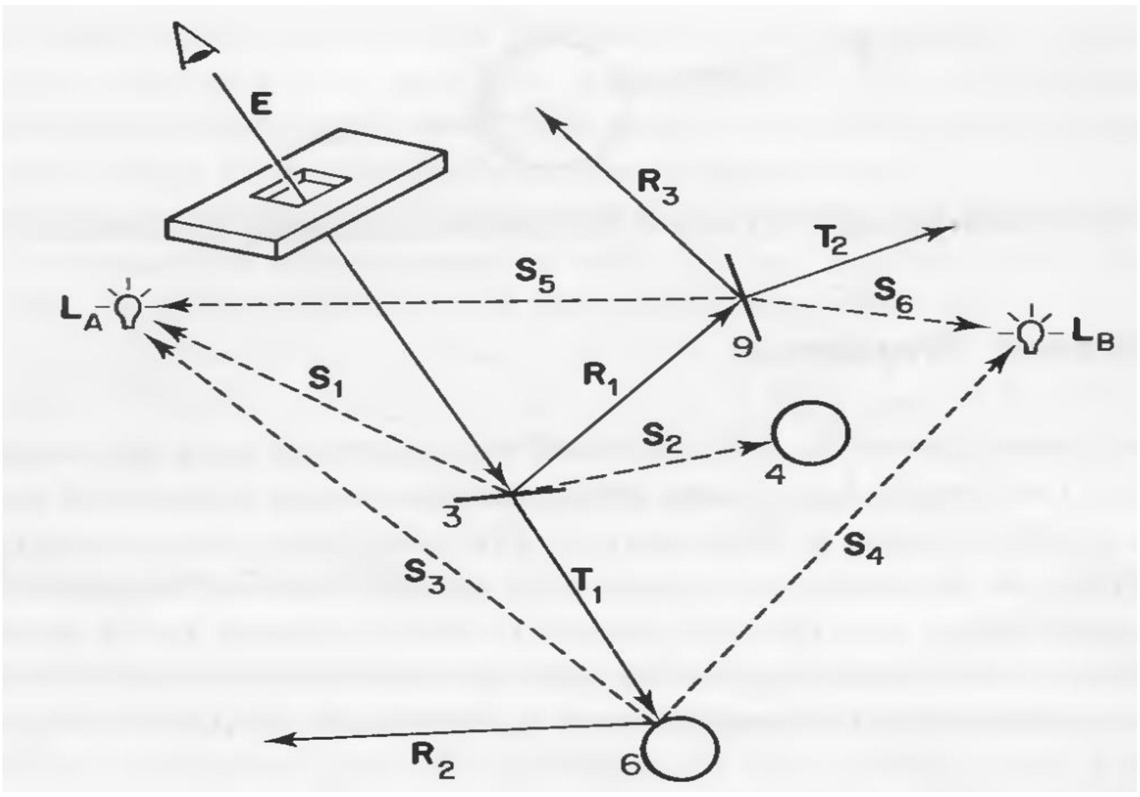

Figure 1.3 A ray intersecting, reflecting, refracting and forming shadow rays (figure from Glassner [1989a])

Whitted [1980] expanded the basic model by continuing tracing after the initial intersection. To make the continuation of tracing possible, reflections and refractions needed to be incorporated into the model which also means that techniques which eliminate parts of the scene that are not directly visible cannot be used. Rays which model refraction are called refraction rays in this study, but they are also known as transmission rays and transparency rays, reflection is modeled by reflection rays. Together reflection and refraction rays are known as secondary rays. Depending on the type of surface material, reflection can be described in two ideal ways. When light is reflected to a single direction the reflection is known as specular reflection, which works on shiny surfaces like mirrors, and which is described in its ideal form by the law of reflection. On the other hand, when the arriving light is reflected to all directions with equal intensity the reflection is known as diffuse reflection, which works on rougher surfaces. This ideal form is described by Lambert’s cosine law. In refraction light bends as it passes the boundary between two media, this bending is described by Snell’s law. As with reflection, there are two types of refraction – specular and diffuse. In ideal specular refraction light would pass through the medium without any scattering, and in ideal diffuse refraction it would be equally scattered to all directions. Beer’s Law can be used to calculate how light intensity is affected when it travels through a medium. For refraction there is one additional consideration. If light is trying to travel from a medium with higher refractive index to one with a lower refractive index at an angle greater than the critical angle, then instead of being refracted, the light is specularly reflected. This phenomenon is known as the total internal reflection. There are no materials that are perfectly specularly or diffusely reflective or refractive. So, it follows that the outcome produced by using the ideal models is not very realistic. Real materials are better represented with a combination of specular and diffuse reflection or refraction. Fresnel equations, or Schlick approximation thereof, can be used to determine how much light is being reflected and refracted. There are models such as the microfacet model which simulate nature to a greater extent, but at the expense of computational power [Glassner, 1989b; Schlick, 1994; Shirley *et al.*, 2009; Whitted, 1980].

Aliasing arises from the fact that computers handle continuous phenomena with discrete samples. Spatial aliasing is a product of the uniform pixel grid. What normally looks like an arc on a screen, is in fact an jumble of square protruding edges when zoomed in close enough. Another aspect of spatial aliasing is that there are objects so small that they are missed by rays, but they are not so insignificant that their absence goes unnoticed. Small objects can also be connected to temporal aliasing when they are, for example, moving across a screen. An object not intersected by a ray at one location can be intersected by a different ray in another location. This causes a distracting phenomenon where an object flickers in and out of existence for seemingly no reason. Temporal aliasing is also at work when a tire, which is spinning forward fast, seems to slow down and reverse direction. Another incarnation of temporal aliasing is when an otherwise constantly moving line suddenly appears to skip forward along its trajectory while missing a line. These effects are corrected for with anti-aliasing [Glassner, 1989a].

Anti-aliasing can be done by using supersampling. Instead of using a single ray to sample a pixel, with supersampling we could use, for instance, seven rays to get a better understanding of a pixel. This does not remove, but simply alleviates, some aliasing effects. However, there is also another version of ray tracing that does anti-aliasing on its own – stochastic or distributed ray tracing [Glassner, 1989a]. In addition to anti-aliasing, distributed ray tracing also creates a more accurate pixel color and enables many advanced effects. It is able to produce depth of field, motion blur, gloss, translucency, soft shadows and penumbras. In distributed ray tracing each pixel is sampled with multiple rays distributed in the same manner as a Poisson disk. Sixteen rays have been found to be enough for most effects, but there are situations where using 64 rays per pixel helps to reduce excess noise. Poisson disk distribution is a Fourier transform of the photoreceptor distribution in an extrafoveal region in the eye of a rhesus monkey. Humans and rhesus monkeys have a similar photoreceptor distribution and human eyes are not prone to aliasing – they have noise. An effective way to approximate Poisson disk distribution is to first distribute the rays evenly across a pixel and then displace each ray on the x- and y-axes by some sufficiently small random amount, so that the new positions of the rays do not overlap and the rays do not bunch up. This method is known as jittering and the outcome is mostly noise instead of aliasing, which is not as distracting for the human visual system. To create the advanced effects rays need to be distributed in the proper dimension, for instance, in time to create motion blur [Cook *et al.*, 1984; Cook, 1989; Foley *et al.*, 1990].

While there are a number of architectures which have been proposed as basis for proprietary hardware implementation of ray tracing, such as SaarCOR, examining these architectures is outside the scope of this thesis. Furthermore, due to space limitations the only acceleration structure to be considered in depth is kd-tree. Many of the techniques discussed for kd-trees are, however, also applicable to at least BVHs (Bounding Volume Hierarchy – a hierarchy which consists of bounding volumes, such as boxes or spheres). In addition to architectures and other acceleration structures, GPU implementations of acceleration structures and ray tracing on GPUs in general are outside the scope of this thesis.

The rest of this thesis is structured as follows: Chapter 2 discusses general efficiency strategies, known efficient ways of calculating intersections between a ray and a triangle, data layout and shadow rays, which account for majority of rays in a system [Smits, 1998]. Chapter 3 goes into memory and ray coherency, culling strategies for packets and ray tracing as a filtering problem. Chapter 4 introduces kd-trees, their different building strategies and parallelization of building. Chapter 5 examines kd-tree traversal in its various forms. Finally Chapter 6 draws some concluding remarks.

2. General considerations

The most basic optimization for ray tracing is to consider the math involved in the different computations. There are multiple ways to calculate a ray triangle intersection, but what holds true to all of them is that taking a square root is a slow operation. Similarly, multiplication is usually faster than division, while addition, subtraction and comparison are faster than any of the aforementioned operations. Thus, using a formula which minimizes the use of square root and division will in most cases speed up execution [Glassner, 1989a].

Adaptive tree-depth control cuts the tracing of a ray when the contribution of the ray to the outcome of the pixel color drops below some predetermined level. The color of a pixel is the sum of the entire ray tree (eye, reflection and refraction rays). The contribution of an individual ray decreases as the depth of the tree increases. Trees can basically have any depth, and the contribution of a ray will become increasingly miniscule [Glassner, 1989a].

Even though ray tracing is capable of supporting a multitude of primitives, supporting only a single primitive can be a benefit. Triangles can be used to approximate any other geometric primitive and thus any 3D-scene. Limiting support to triangles only reduces branching which increases execution speed. It also eliminates the need for different intersection code for different primitives, which leads to simpler code. Simpler code is easier to optimize for programmer and compiler alike and it also reduce the likelihood of mistakes [Wald *et al.*, 2001].

Because of the development in processors, ray tracing is bounded more by transfer speed between memory and cache than CPU speed. Data is moved to cache as entire cache lines, and data structures need to be designed so that they occupy entire cache lines. Doing so will minimize data transfer and increase speed, possibly at the cost of bigger than needed structures, as they are made to conform to cache line size [Wald *et al.*, 2001]. Wächter [2004] notes that the usefulness of cache line alignment ultimately depends on the hardware used, as the P4 coupled with 8-bit level 1 data cache (level 1 data cache is the smallest and fastest cache in the multi-level cache hierarchy of modern processors, it is the first place a processors looks for data after registers) does not see the increase in speed reported for the PIII by Wald *et al.* [2001].

2.1 Triangle intersection

To reach 30 FPS (Frames Per Second) for primary visibility (no shadows, reflections or refractions) on 1080p resolution without any sort of anti-aliasing, a ray tracer would need to handle 62M rays per second. Increasing rays per pixel to 16 for anti-aliasing and advanced effects results in 995M rays per second, again for only primary visibility. It is thus clear that determining whether or not a ray has an intersection in a scene, and the speed of the calculation used to resolve the intersection point, are of crucial importance.

A ray is defined by

$$R(t) = O + tD; t \in (t_{min}, t_{max}), \tag{2.1}$$

where O is the origin of the ray $R(t)$, D the normalized direction of travel and t the distance. If $t < 0$, the intersection is behind the origin of the ray and, thus, rejected. Furthermore, usually $t_{min} = \varepsilon$ as $t_{min} = 0$ could result in self-intersection. The intersection is also rejected if a closer intersection already exists or the embedding plane of the triangle and the ray are parallel. A ray is parallel with a plane when the product of the plane normal and the ray direction equals zero. The special case of the ray being parallel with the embedding plane while also being on the plane can be ignored because hitting an edge of a triangle has no effect [Badouel, 1990; Haines, 1989; Möller and Trumbore, 1997; Wald, 2004].

A triangle $\Delta\, ABC$ is an area bounded by vertices A, B and C. For the ray $R(t)$ to hit a triangle it has to satisfy the condition $t_{\text{min}} \leq t_{\text{hit}} \leq t_{\text{max}}$ and hit-point $H = R(t_{\text{hit}})$ has to be inside the triangle. There are multiple ways to solve for this problem, but methods known as Möller-Trumbore, Badouel and Plücker coordinates are currently thought of as the best [Badouel, 1990; Haines, 1989; Möller and Trumbore, 1997; Wald, 2004].

Badouel's algorithm starts by resolving whether or not a ray has an intersection with the embedding plane of a triangle. The existence of an intersection is ascertained by calculating the signed distance, t_{plane}, to the embedding plane, and by determining if t_{plane} is within the interval where intersections are sought. The signed distance can be calculated as

$$t_{plane} = -\frac{(O - A) \cdot N}{D \cdot N}, \tag{2.2}$$

where N is the normal of the plane and it is calculated by taking a cross product ($N = (B - A) \times (C - A)$) [Badouel, 1990; Haines, 1989; Wald, 2004]. If the ray reaches the embedding plane then the next step is to determine if it intersects the triangle $\Delta\, ABC$ as well. The intersection point H is calculated using Equation 2.1 where $t = t_{plane}$. The barycentric coordinates of H can be solved as follows

$$H = \alpha A + \beta B + \gamma C, \tag{2.3}$$

where $\alpha + \beta + \gamma = 1$. H is inside the triangle if α, β and γ all have a value which is ≥ 0 and ≤ 1. The amount of operations can be reduced, as it is enough to check $\beta \geq 0$, $\gamma \geq 0$ and $\beta + \gamma \leq 1$ [Badouel, 1990; Wald, 2004].

Triangle $\Delta\, ABC$ and point of intersection H can be projected to one of the primary 2D planes to increase execution speed, as long as the plane of projection is not orthogonal to the plane ABC, because projection has no effect on barycentric coordinates. To maintain numerical stability, the projection should be done to the plane where the triangle has the largest projected area, which is the dimension where *N* has its maximum absolute component. After projection to the *XY* plane, Equation 2.3 has the form

$$H' = \alpha A' + \beta B' + \gamma C', \tag{2.4}$$

which through substitution and rearranging becomes

$$\beta = \frac{b_x h_y - b_y h_x}{b_x c_y - b_y c_x}, \gamma = \frac{h_x c_y - h_y c_x}{b_x c_y - b_y c_x}, \tag{2.5}$$

where *b = C' – A'*, *c = B' – A'* and *h = H' – A'*. Additionally here and in Equation 2.7 *x* and *y* represent the two-dimensional coordinates [Badouel, 1990; Wald, 2004; Wächter, 2004]. Execution speed can be improved further by precomputing and storing values that remain the same for all intersections. The edges of a triangle and the projection dimension are such values, and so is the normal of a triangle. For normal, a further consideration is that when the projection dimension *k* is known, *N* · *k* cannot be zero. It is therefore possible to derive and store *N'* by dividing *N* with *N* · *k* which leads to

$$t = \frac{(A-O)\cdot N'}{D\cdot N'} = \frac{d - O_u \cdot N'_u - O_v \cdot N'_v - O_k \cdot N'_k}{D_u \cdot N'_u + D_v \cdot N'_v - D_k \cdot N'_k}. \tag{2.6}$$

Values $d = A \cdot N'$, $N'_u = \frac{N_u}{N_k}$ and $N'_v = \frac{N_v}{N_k}$ are constants, $N'_k = 1$ and does not need to be stored. Here *u* and *v* represent the two planes where the triangle was not projected to. Additionally, with the normal known, *u* and *v* need not be calculated using a slow modulo operation (*u* = (*k* + 1) mod 3, *v* = (*k* + 2) mod 3) since a table lookup ({0, 1, 2, 0, 1}) will suffice. The Newton-Raphson method can be used instead of division to increase execution speed. Computing β and γ can be simplified in a manner similar to Equation 2.6

$$\begin{aligned} \beta &= \frac{1}{b_x c_y - b_y c_x}(b_x H_y - b_x A_y - b_y H_x + b_y A_x \\ &= \frac{b_x}{b_x c_y - b_y c_x} H_y + \frac{-b_y}{b_x c_y - b_y c_x} H_x + \frac{b_y A_x - b_x A_y}{b_x c_y - b_y c_x} \\ &= K_{\beta y} H_y + K_{\beta x} H_x + K_{\beta d}. \end{aligned} \tag{2.7}$$

Here $K_{\beta y}$, $K_{\beta x}$ and $K_{\beta d}$ are constants. The same approach can be used for γ, and then it follows from the properties of barycentric coordinates that α = 1 – β – γ. The resulting β and γ can also be used, for example, in determining texture-coordinates [Wald, 2004; Wächter, 2004]. Algorithm 2.1.1 presents Badouel's algorithm for ray bundles.

Improved Badouel's algorithm

```
Algorithm Badouel(O, D, A, B, C)
Input: ray origin O and direction D, vertices A, B and C
Output: no hit or hit at intersection distance, barycentric coordinates beta and gamma
b = C-A; c = B-A; N = c x b;
t_plane = -((O-A) · N) / (D · N);

if (t_plane < Epsilon || t_plane > t_max)  then
   return NO_HIT;
end if
if (|N.x| > |N.y|) then
   if (|N.x| > |N.z|) then
      k = 0; /* X */
   else
      k = 2; /* Z */
   end if
else
   if (|N.y| > |N.z|) then
      k = 1; /* Y */
   else
      k = 2; /* Z */
   end if
end if

u = (k + 1) mod 3; v = (k + 2) mod 3;
H[u] = O[u] + t_plane • D[u];
H[v] = O[v] + t_plane • D[v];
beta = (b[u] • H[v] - b[v] • H[u]) / (b[u] • c[v] - b[v] • c[u]);

if (beta < 0) then
   return NO_HIT;
end if
gamma = (c[v] • H[u] - c[u] • H[v]) / (b[u] • c[v] - b[v] • c[u]);
if (gamma < 0) then
   return NO_HIT;
end if
if (beta+gamma > 1) then
   return NO_HIT;
end if
return HIT(t_plane,beta,gamma);
```

Algorithm 2.1.1 Projection method [Wald, 2004]

Wächter [2004] notes that by rearranging the distance test to:

```
if(!(t > 0.0) || (t > ray.tfar)) continue;
```

instead of testing if t is negative or larger than ray.tfar, a situation where *t = NaN* (not a number) can be avoided altogether. In addition, it is possible to further speed up this operation while also detecting and handling ±∞, *QNaN* (quiet *NaN*) and *SNaN* (signaling *NaN*) without floating-point comparisons. This is done by replacing floating point operations with integer arithmetic, where the previous formulation of the distance test is replaced by

```
        if((unsigned int&)t > (unsigned int&)ray.tfar) continue;
```

and ray.tfar is initialized to 3.3e38f. All negative values of t will pass the test because they have a sign-bit of 0x80000000, while ray.tfar always has a valid positive floating-point number. Integer arithmetic is also used in the same way to transform part of the inside test from $u + v > 1.0$f to (unsigned int&)uv > 0x3F800000. The entire test could be done in integer arithmetic, but other optimizations lead to a version where $u < 0.0$ and $v < 0.0$ are determined in a different manner.

Wächter [2004] also experimented with other ideas, such as bit-sifting to remove projection case look-up table dependency, but found that the methods did not impact execution speed positively, sometimes resulting in a decrease in execution speed. The lack of improvement could, in some cases, be attributed to the features of the underlying hardware architecture used in the tests (P4). Furthermore, when converting the C-code to SIMD (Single Instruction, Multiple Data) it was noticed that some of the optimizations, like unsigned integer comparisons, were not supported by the versions available at the time (MMX, SSE(2/3)).

Benthin [2006] notes that Badouel's algorithm has an early distance test and a very late inside test, but the distance test exits only roughly 18% of the time, while the inside test exits 52 to 68% of the time. As such an algorithm with early inside test, like Plücker, should see increased performance.

Komatsu *et al.* [2008] contradict Benthin [2006] and present results showing Plücker exiting only approximately 21% of the time in the first test. This nearly exactly opposite result can, however, be explained if their pseudo code is an accurate representation of their actual code. Going by the pseudo code – which their actual text does not contradict – Komatsu *et al.* [2008] seem to have a fundamental misunderstanding of Plücker coordinates. In their pseudo code Komatsu *et al.* [2008] perform the distance test first and the inside test second, when the order should obviously be reversed.

Even though Komatsu *et al.* [2008] conduct tests without using spatial data structures, their results are still considerably out of line when compared to those reported elsewhere in the literature. For Badouel they report only 1.0027x speed increase over Möller-Trumbore, while for Plücker they get 1.0250x increase over Möller-Trumbore. Comparison of Badouel and Plücker produces a difference of 1.0222x in favor of Plücker. All of the results are for eye rays. Wald [2004], on the other hand, reports 2.1x – 2.3x for eye rays and 1.9x – 2.0x for shadow rays in favor of Badouel over Möller-Trumbore. Benthin [2006] reports 1.2x increase for Plücker over Badouel. Kensler and Shirley [2006] report 1.7x increase over Möller-Trumbore for an algorithm that is slower than either Badouel or Plücker (comparing results from Kensler and Shirley [2006] to those reported for a version of Badouel by Wald in Havel and Herout [2010], the testing machines are roughly comparable). Based on this it would appear that there is some sort of a problem with the implementations of Badouel and Plücker by Komatsu *et al.* [2008].

Unlike Badouel, Möller-Trumbore does not start by intersecting the embedding plane. Instead, the triangle to be intersected is translated to the origin and transformed so that it is aligned with the x-axis. In addition, only triangle vertices are stored and no precomputation is needed. While this results in memory savings of 25 to 50% for triangle meshes, it also means slower execution when compared to recent versions of Badouel and Plücker coordinates [Möller and Trumbore, 1997]. The other advantage Möller-Trumbore has over Badouel and Plücker is that its conditional checks are performed earlier, resulting in fewer operations when there is no intersection [Komatsu, 2008]. The following equation describes a point on a triangle in Möller-Trumbore:

$$T(u,v) = (1-u-v)A + uB + vC. \tag{2.8}$$

For an intersection to occur u and v must fulfill the same conditions as β and γ in Badouel. An intersection is calculated as $R(t) = T(u, v)$. Substituting $T(u, v)$ with $R(t)$ in Equation 2.8 and rearranging the resulting equation leads to

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2)\cdot E_1} \begin{bmatrix} (T \times E_1)\cdot E_2 \\ (D \times E_2)\cdot T \\ (T \times E_1)\cdot D \end{bmatrix}, \tag{2.9}$$

where $E_1 = B - A$, $E_2 = C - A$ and $T = O - A$ [Möller and Trumbore, 1997]. Further optimizations are possible through the use of scalar triple product rules and the commutative property of cross product:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(E_2 \times E_1)\cdot D} \begin{bmatrix} (E_1 \times E_2)\cdot T \\ (T \times D)\cdot E_2 \\ (D \times T)\cdot E_1 \end{bmatrix} = -\frac{1}{N\cdot D} \begin{bmatrix} N\cdot T \\ -(D \times T)\cdot E_2 \\ (D \times T)\cdot E_1 \end{bmatrix}. \tag{2.10}$$

Edges E_1 and E_2 and the normal N are constants, and can thus be precomputed [Komatsu *et al.*, 2008].

By applying scalar triple product rules to Equation 2.10, Möller-Trumbore can be made more efficient when used with ray frustums:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = -\frac{1}{N\cdot D} \begin{bmatrix} N\cdot T \\ D\cdot(-T \times E_2) \\ D\cdot(T \times E_1) \end{bmatrix}. \tag{2.11}$$

Outcome of $N \cdot T$ is a constant for all rays in a frustum, since they are assumed to share an origin, and can thus be precomputed. Additionally, $-T \times E_2$ and $T \times E_1$ are also constants for all rays in a frustum. The precomputed values take 40 bytes of space and they can be easily arranged so as to take advantage of cache line size. The improved version also retains the early condition checks of its predecessor [Komatsu, 2008]. Algorithm 2.1.2 describes Möller-Trumbore for ray bundles.

Möller-Trumbore

```
Algorithm Precomputation triangle(A, B, C)
Input: triangle vertices A, B and C
Output: triangle edges 1 and 2, and triangle normal
   $E_1 \leftarrow B - C$
   $E_2 \leftarrow C - A$
   $N \leftarrow E_1 \times E_2$
   return $(E_1, E_2, N)$
Algorithm Precomputation frustum(O, N, A, $E_1$, $E_2$)
Input: ray origin O, triangle normal N, triangle vertex A, triangle edges 1 and 2
Output: edges 1 and 2 translated to origin, triangle normal at origin
   $T \leftarrow O - A$
   $G_u \leftarrow -T \times E_2$
   $G_v \leftarrow T \times E_1$
   $f_2 \leftarrow N \cdot T$
   return $(G_u, G_v, f_2)$
Algorithm Intersection(D, N, $G_u$, $G_v$, $f_2$)
Input: ray direction D, triangle normal N, triangle edges 1 and 2 at origin, triangle
normal at origin
Output: no hit or hit at distance t and barycentric coordinates u’ and v’
   $f_1 \leftarrow N \cdot D$
   if $f_1 \geq 0$ then
      return(No Hit)
   end if
   $u' \leftarrow D \cdot G_u$
   if $u' < 0$ then
      return(No Hit)
   end if
   $v' \leftarrow D \cdot G_v$
   if $v' < 0$ then
      return(No Hit)
   end if
   if $u' + v' > -f_1$ then
      return(No Hit)
   end if
   $t \leftarrow -f_2 / f_1$
   return(Hit, u’, v’, t)
```

Algorithm 2.1.2 Ray bundle Möller-Trumbore [Komatsu, 2008]

Contrary to Badouel, Plücker coordinates have an early inside test and a late distance test. Plücker coordinates can be used to express a directed 3D line in 6D. A line L which goes through 3D points X and Y is formulated in Plücker space as $L = [X - Y, X \times Y]$, from which it follows that a ray R is $R = [D, D \times O]$. Two lines, $L_0 = [U_0, U_1]$ and $L_1 = [V_0, V_1]$, intersect if $L_0 \cdot L_1 = U_0 \cdot V_1 + U_1 \cdot V_0 = 0$. If $L_0 \cdot L_1 > 0$ then the lines pass each other counterclockwise, while $L_0 \cdot L_1 < 0$ means that the lines pass each other clockwise. A ray R and a triangle Δ ABC expressed in Plücker coordinates intersect, if $R \cdot A > 0$, $R \cdot B > 0$ and $R \cdot C > 0$ or $R \cdot A < 0$, $R \cdot B < 0$ and $R \cdot C < 0$, i.e., all the inner products between edges and a ray have the same sign. If all inner products equal to 0.0, then the ray is coplanar with the triangle and produces no intersection. [Benthin, 2006; Erickson, 1997; Komatsu *et al.*, 2008; Shoemake, 1998; Wächter, 2004].

Because all rays in a bundle are assumed to share an origin (if the origin is not shared, then this method is not usable), Plücker can be optimized further for ray bundles by transferring the ray origin to the origin of the coordinate system. The transfer is achieved by subtracting O from vertices A and B. This simplifies the inner product between R and E = [A – B, A x B] to

$$\begin{aligned} R \cdot E &= D \cdot \big((A-O) \times (B-0)\big) + D \times (O - O) \cdot ((A-O)-(B-O)) \\ &= D \cdot \big((A-O) \times (B-0)\big) + 0 \cdot (A-B) \\ &= D \cdot ((A-O) \times (B-0)). \end{aligned} \tag{2.12}$$

The computations are sped up because $(A-O) \times (B-0)$ and similar computations for the other two edges are constant for all rays in a bundle. As such, each ray only needs to take three dot products to solve for intersection. After the intersection has been verified (using either ray bundle or single ray method) t can be calculated using Equation 2.2. The result is constant for all rays in a bundle and, hence, it can also be precomputed. If the triangles are, however, dynamic, computing on the fly might be more efficient [Benthin, 2006; Komatsu, 2008]. Algorithm 2.1.3 presents ray bundle Plücker test.

Plücker test for ray bundles

```
Algorithm Precompute triangle(A, B, C)
Input: triangle vertices A, B and C
Output: triangle normal N
   $E_1 \leftarrow B - C$
   $E_2 \leftarrow C - A$
   $N \leftarrow E_1 \times E_2$
   return (N)

Algorithm Precompute frustum(O, N, A, B, C)
Input: ray origin O, triangle normal N, triangle vertices A, B and C
Output: triangle edges BA, CB and AC translated to origin, triangle normal at origin
   $T_0 \leftarrow (B - O) \times (A - O)$
   $T_1 \leftarrow (C - O) \times (B - O)$
   $T_2 \leftarrow (A - O) \times (C - O)$
   $f_2 \leftarrow N \cdot (O - A)$
   return($T_0, T_1, T_2, f_2$)

Algorithm Plücker(D, N, $T_0, T_1, T_2, f_2$)
Input: ray direction D, triangle normal N, triangle edges BA, CB and AC at origin, triangle normal at origin
Output: no hit or hit at distance t
   $\alpha \leftarrow D \cdot T_0$
   $\beta \leftarrow D \cdot T_1$
   $\gamma \leftarrow D \cdot T_2$
   if $\alpha$, $\beta$ and $\gamma$ don't have the same sign then
      return(No Hit)
   end if
   $f_1 \leftarrow N \cdot D$
   if $f_1 \geq 0$ then
      return(No Hit)
   end if
   $t \leftarrow -f_2 / f_1$
   return(Hit, t)
```

Algorithm 2.1.3 Plücker test modified from [Komatsu, 2008], see Benthin [2006, p.78] for SSE

Another variation of Plücker precomputes scaled normals of p and q (the selected axes) coordinates: N_p and N_q, scaled p and q coordinates for two edges: $E0_p$, $E0_q$, $E1_p$ and $E1_q$, p and q coordinates for a vertex: A_p and A_q, and the dot product of the vertex and scaled triangle normal: d. Index of the discarded axis, r, is also stored, and it is used to restore the indexing of coordinate components during intersection tests. Using these values an intersection is calculated as follows

$$\begin{aligned}
\det &= D_p N_p + D_q N_q + D_r \\
t' &= d - (O_p N_p + O_q N_q + O_r) \\
T_p &= t' D_p - \det \bullet (A_p - O_p) \\
T_q &= t' D_q - \det \bullet (A_q - O_q) \qquad (2.13) \\
u' &= E1_q T_p - E1_p T_q
\end{aligned}$$

$$v' = E0_p T_q - E0_q T_p$$
$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{det} \begin{bmatrix} t' \\ u' \\ v' \end{bmatrix}.$$

The first six lines are used to determine whether or not an intersection happens and the last line calculates the position of the intersection. Therefore, the slow division operation is done only if there is an actual intersection. Further improvement is possible in the case of axis aligned triangles, where two dimensions have a normal which evaluates to zero (here N_p and N_q) [Havel and Herout, 2010; Shevtsov *et al.*, 2007b].

An algorithm by Havel and Herout [2010] is a combination of Badouel by Wald [2004] sans the projection to a single plane and the version of Plücker by Shevtsov *et al.* [2007b]. While N is calculated as in Badouel, normals for the two other planes are calculated as

$$N_1 = \frac{C - A \times N}{|N|^2}, \quad d_1 = -N_1 \cdot A$$
$$N_2 = \frac{N \times B - A}{|N|^2}, \quad d_2 = -N_2 \cdot A. \tag{2.14}$$

The following equations describe how the barycentric coordinates, which are expressed as scaled distance from their planes, are determined, and used to resolve the inside test. The resulting values are inserted to the last line of Equation 2.13 to resolve the distance test and calculate the intersection point:

$$\begin{aligned} P &= O + tD \\ u &= N_1 \cdot P + d_1 \\ v &= N_2 \cdot P + d_2 \end{aligned} \tag{2.15}$$

$$\begin{aligned} \det &= D \cdot N \\ t' &= d - (O \cdot N) \\ P' &= \det \bullet O + t' \bullet D \\ u' &= P' \cdot N_1 + \det \bullet d_1 \\ v' &= P' \cdot N_2 + \det \bullet d_2. \end{aligned} \tag{2.16}$$

Kensler and Shirley [2006] use a genetic algorithm to search all known unique ways to compute an intersection with a volume defined by four points. They then hand tune the resulting algorithm to produce another algorithm which outperforms Möller-Trumbore on average by 1.72x to 2.16x (depending on whether or not rays in a bundle share origin).

Of the approaches presented here Wald [2004], Shevtsov *et al.* [2007b] and Havel and Herout [2010] report the highest amount of tests per second. While the algorithm by Havel and Herout [2010] is the fastest in general, it is comparable or slightly slower than Wald's [2004] and Shevtsov *et al.*'s [2007b] in the worst case scenario.

All the previous methods compute values for each triangle separately. Typical scenes, however, consist of meshes where triangles share vertices. Avoiding repeating unnecessary calculations by applying known values to neighbouring triangles should increase execution speed.

Triangle fan is a structure with vertices $\{p_0, \ldots, p_n\}$, where p_0 is the center vertex [Galin and Akkouche, 2005]. The vertices form a set of triangles, as presented in Figure 2.1.

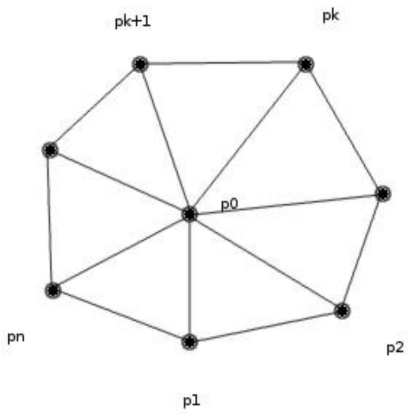

Figure 2.1 Closed triangle fan

Triangle values for the fan are calculated using a modification of Möller-Trumbore. Instead of solving Equation 2.8 to Equation 2.9, we can derive

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(e_k \times e_{k+1}) \cdot D} \begin{bmatrix} -(e_k \times e_{k+1}) \cdot s \\ (s \times D) \cdot e_{k+1} \\ -(s \times D) \cdot e_k \end{bmatrix}. \qquad (2.17)$$

Here $s = O - p_0$, which is the translation vector to ray origin. Edge vectors e_k and e_{k+1} are calculated as $p_k - p_0$ and $p_{k+1} - p_0$, respectively. The first step in the algorithm is to solve the translation vector, s, and the constant cross product, $n = s \times D$. In the second step all edge vectors are calculated and for each vector also a dot product is determined. These values are calculated in pairs: $a = p_k - p_0$, $b = p_{k+1} - p_0$, $x_a = n \cdot a$ and $x_b = n \cdot b$. If $x_a$and x_b have the same sign, then there is no intersection, because both of the vertices are on the same side of a plane that contains the ray and the center vertex. In this case the next two vertices are evaluated. Because the next triangle shares a vertex with the first triangle, computations can be shared. The value of vector b is assigned to vector a and x_a becomes x_b. If, on the other hand, there is an intersection, then the triangle normal, $n_k = a \times b$, and determinant, det $= n_k \cdot D$, are calculated. A determinant value in an interval close to zero (for instance, $\varepsilon = 10^{-5}$) indicates that the ray is on the triangle plane and, thus, there is no intersection. Otherwise, the barycentric coordinates and the distance can be calculated as

$$u_k = -\alpha x_b \quad v_k = \alpha x_a \qquad t_k = -\alpha(n_k \cdot s), \tag{2.18}$$

where $\alpha = \frac{1}{det}$. If vertex locations are not needed and are not shared between triangles, then edges can be precomputed. Similarly, triangle normals can also be precomputed [Galin and Akkouche, 2005].

Triangle fan building can start from either a closed or an open mesh, and the goal is to build as large fans as possible while avoiding forming fans with only a single triangle – also known as degenerate fans. If the mesh is closed, then the first fan is created at a given candidate vertex c. Afterwards, and in the case of an open mesh, c is selected by using a fitness function

$$f(c) = \alpha v(c) + \beta r(c) - \gamma t(c). \tag{2.19}$$

Here $v(c)$ is used to denote the connectivity of c, $r(c)$ expresses the number of open edges in the one ring neighbourhood of c, and $t(c)$ indicates the number of degenerate triangle fans remaining, if a fan were to be created centered on c. Coefficients α, β and γ are used to adjust the relative weight of each variable of the function and values $\alpha = 2$, $\beta = 7$ and $\gamma = 11$ have been suggested to work well in general. If the starting mesh is closed, then an open mesh is formed by removing the triangles in the fan, centered at c, from the mesh. From the now open mesh, the triangles with open edges are placed in a list and the new candidate is selected by the fitness function from the one ring neighbourhood of the triangles in the list. If the mesh is not closed, then the algorithm starts from creating the list of triangles with open edges. The process is repeated until there are no more triangles left. Using triangle fans speeds up intersection calculations by nearly 40% [Galin and Akkouche, 2005]. Replacing Möller-Trumbore with the algorithm by Havel and Herout [2010] should provide a further speedup, as there is no reason it would be slower for triangle fans when it is faster for single triangles.

Triangle fans can be formed from convex and star polygons. Storing models made from triangle fans instead of individual triangles decreases the required storage space, as fans take $M + 2$ (where M is the number of triangles in the fan) vertices to describe, while separate triangles require $3M$ vertices. Loading speed is also increased, since there are fewer lines to load.

2.2 Data layout in main memory

Efficient data layout not only reduces the needed amount of main memory – henceforth memory – but can also increase traversal performance due to fewer cache misses. Cache misses can be reduced by storing data elements so that those elements close to each other in meshes and hierarchies are also close in memory layouts [Moon *et al.,* 2010]. Mapping kd-trees to memory efficiently is complicated, because kd-trees are multidimensional while memory is primarily one-dimensional. Common strategies, such as B-trees, cannot be used to describe a kd-tree in memory. There are two principal ways to manage dynamic variables in a memory pool (a continuous block of empty memory), general memory allocator and fixed-size memory allocator. Of the two, general memory allocator is more wasteful as it requires two additional pointers for deallocation, but it is also more flexible. Fixed-size memory allocator does not need redundant pointers, but it can only be used to allocate memory for variables with same type and size [Havran, 2001].

A common, but inefficient, way to store kd-trees is known as random representation. It uses general memory allocation and each node allocates memory for four pointers and node information. More importantly, node memory address and node location in a kd-tree have no connection. Another approach is the depth first search (DFS) representation which produces a linear order for nodes in the memory pool. Due to the use of fixed-size memory allocation, it has lower memory consumption than random representation. Memory is allocated for two pointers and node information. Subtree representation also uses fixed-size memory allocation and generates a linear order for nodes in a memory pool. Unlike DFS representation, however, each variable allocates cache line size of memory. The allocated memory is used to store nodes which form subtrees. Subtrees can be represented in two possible ways: ordinary subtrees and compact subtrees. An ordinary subtree holds equal sized nodes that have pointers to their two children. A compact subtree representation only has pointers to subtrees, as node addressing is provided by the traversal algorithm. The lack of pointers between nodes causes a need for a specialized traversal algorithm, which is not justified by the limited theoretical efficiency increase over ordinary subtree representation. While the theoretical speedup for DFS and ordinary subtree over random representation are 1.62x and 2.5x respectively, the actual reductions in the cost of a traversal step were, on average, 1.25x and 1.39x, respectively [Havran, 2001].

2.3 Shadow rays

Unlike other rays, shadow rays need only know if there is an intersection or not. So, calculating rest of the normal intersection information would be redundant. Because shadow rays can account for more than 90% of all rays in a system, avoiding unnecessary work when tracing them obviously produces a major boost in efficiency. Furthermore, because finding an intersection is sufficient for shadow rays, traversal can be terminated as soon as an intersection is found [Smits, 1998].

Light Buffer is a method where a point light source can be thought of as being enclosed by a uniformly subdivided direction cube. The cube is constructed in a preprocessing step. Each direction cell of the cube is associated with a list that holds all the opaque objects that can be ‘seen’ from the cell. The lists are sorted into ascending order based on depth. When determining whether or not a point is in shadow, the shadow ray can be thought of as starting from the light source, piercing a direction cell and continuing on for the defined length. The list associated with the cell holds all the objects which can cause the point to be in shadow. The algorithm then makes use of the fact that as long as there is an occluding object, a shadow is confirmed. As there is no need to find the closest occluding object, the execution can stop as soon as an intersection with an occluding object is confirmed. The Light Buffer can be improved by culling back facing polygons from the candidate list because they are known to be in shadow. Lists that contain only a single polygon can be deleted, as a polygon cannot occlude itself unless it is back facing. Also, if there is an object which covers the whole direction cell, all objects which come after the covering object in the list can be culled because the covering object shades everything in its direction pyramid. A list with a covering object is specially marked, and all objects with depth greater than the covering object can be determined to be in shadow without further tests [Arvo and Kirk, 1989]. This approach obviously only works for stationary light sources and static objects. For dynamic scenes, all the directional cubes need to be rebuilt for every frame or the cells where there has been change need to be identified and their lists updated.

Using a single large frustum to trace shadow rays (also applies to eye rays) means that the frustum cannot be tight. Instead, utilizing multiple smaller frusta, say one frustum per SIMD, would allow tighter bounds, and also enable individual termination as soon as all rays have a confirmed intersection. Benthin and Wald [2009] develop such a system with the now cancelled Larrabee architecture in mind. Their traversal algorithm is based on common origin of rays inside each frustum, which makes it possible to use only intervals over ray directions, when interval arithmetic (see Section 3.3) is the culling method. Using frustum traversal also means that individual rays are only needed for actual intersection tests, and they can thus be generated on demand. Culling efficiency of interval arithmetic is dramatically reduced if a direction interval has a zero in one of the three dimensions (ray directions differ for this dimension). Such a case is rare, and it can be corrected by intersecting the AABB (Axis-Aligned Bounding Box – a bounding box is an area defined by six planes surrounding an object) of the ray direction and the node AABB – assuming the dimension with zero is tight. The increase in triangle intersection tests caused by frustum traversal can be counteracted by using further culling – such as back face culling and corner rays – at the leaves.

3. Ray aggregation

3.1 Memory coherency

Real-time execution of ray tracing algorithms has for a long time been impossible due to insufficient computational power of the available processors. However, after the introduction of the multicore processors sufficient processing power has come within reach. Developers of interactive ray tracers now face a new obstacle in the form of limited bandwidth between memory and processor cache [Navrátil *et al.*, 2007; Shirley *et al.*, 2008; Wald *et al.*, 2001]. Especially inefficient are ray tracers which utilize depth-first traversal, as they must trace each ray to completion before starting on a new ray. This leads to incoherent access to scene geometry when possibly multiple primary rays intersect the same geometry one after another at different points of the rendering [Navrátil *et al.*, 2007; Pharr *et al.*, 1997]. All geometry and textures can be loaded to memory provided that the scene is small enough, but this is a slow not to mention wasteful process as it is possible that not even half of the loaded data is used in the rendering process. Furthermore, with ever increasing amounts of scene geometry and textures, it is likely that the scene geometry and textures simply do not fit in memory [Pharr *et al.*, 1997].

A technique known as TOD (Texture On Demand) offers a way to solve this problem. Minimizing traffic between disk and memory can be achieved by loading the textures to memory lazily, i.e., only when needed. Some predetermined amount of memory is set aside for textures and a texture is loaded when needed. Retaining knowledge of the last used texture is advisable as the next texture to be requested is often the same as the previous. Loaded textures remain in memory which is searched first when a texture is requested. New textures are loaded only if the requested texture is not found. In case the memory is full the texture to be removed can be determined by utilizing, for instance, LRU (Least-Recently Used) algorithm [Peachey, 1990; Pharr *et al.*, 1997]. Similarly, geometry traffic between disk and memory can be minimized by loading geometry only when it is needed for intersection tests. However, steps to counteract heap fragmentation caused by reallocation of variable memory block sizes need to be taken. Spatial locality in 3D-space must be tied with spatial locality in memory to ensure coherency in both 3D-space and memory. While lazy loading of textures and geometry reduces the memory needed to render a scene, it can also slightly reduces the time the rendering takes [Pharr *et al.*, 1997].

Ray coherence cannot be exploited while all rays are being traced recursively – independently and in a fixed order. To make use of ray coherence, rays need to be traced in groups and when needed. However, if rays are no longer traced from start to finish, the color of a pixel also needs to be determined in a different way. The color can be determined by retaining all of the necessary information in the ray and by computing the outgoing color as a weighted sum of incoming colors. To make use of coherency, a scene is divided into a voxel grid and each voxel is associated with a queue of rays waiting for an intersection test with the geometry it contains. Each voxel also contains information about overlapping voxels. Rays are tested for intersection against geometry in all overlapping voxels. In case an intersection is found, shading is calculated and the new rays produced by the intersection are added to the queue. On the other hand, if an intersection is not found the ray is moved to the next voxel which contains geometry and is on the route of the ray. Because loading new geometry to memory is slow, voxels to be processed are selected based on a cost (a lot of geometry which currently is not in memory is costly) benefit (many rays in a single queue move the rendering process more than just a few rays) approximation [Pharr *et al.*, 1997]. Algorithm 3.1.1 presents memory coherent traversal.

BFS (Breadth First Search) traversal

```
Algorithm BFS_traversal
Generate eye rays and place them in queues
while there are queued rays
   Choose a voxel to process
   foreach ray in voxel
      Intersect the ray with the voxel's geometry
      if there is an intersection then
         Run the surface shader and compute the BRDF
         Insert spawned rays into the voxel's queue
         if the surface is emissive then
            Store radiance contribution to the image
            Terminate the ray
         else
            Advance the ray to the next voxel queue
         end if
      end if
   end foreach
end while
```

Algorithm 3.1.1 Memory coherent traversal [Pharr *et al.*, 1997]

In Algorithm 3.1.1 bidirectional reflectance distribution function (BRDF) is a function that describes the angles of incident and reflectance by using cross products of two hemispheres [Rusinkiewicz, 1997]. The reordering scheme comes with a small overhead, but as the available memory gets smaller the algorithm is able to render scenes faster than depth-first ray tracing. However, due to the new formulation of the rendering equation, adaptive sampling cannot be used. Furthermore, the chosen acceleration structure – uniform grid – is not able to adapt to different levels of geometry density which can lead to cache trashes. Moreover, the amount of active rays in the system is not limited in any way and it is in fact a feature of the algorithm to increase the amount of active rays fast. Uncontrolled growth in the amount of active rays can lead to cache trashing [Navrátil, 2010; Navrátil *et al.*, 2007; Pharr *et al.*, 1997].

Rectifying the problem with adapting to geometric density can be done easily by replacing uniform grid with another acceleration structure. The acceleration structure used in this study is discussed in Chapters 4 and 5. Cache problems can be solved with an algorithm which can employ memory-to-cache bandwidth efficiently and take cache size into consideration [Navrátil, 2010; Navrátil *et al.*, 2007].

Instead of queuing rays at all voxels with geometry, the new algorithm queues rays in the acceleration structure so that the size of the geometry contained in the subtree which starts from the queue node is not larger than the size of L2 (Level 2) cache. Level 2 cache is a larger version of level 1 cache. It holds recently used data, and it is accessed if the data the processor needs is not found in level 1 cache. In case there is a leaf with geometry whose size is larger than L2 cache size, the leaf is treated by loading blocks of rays and geometry to prevent cache trashing. The algorithm proceeds in generations. First all eye rays are queued. When a queue is processed, rays traverse the remaining subtree. Rays are traced until an intersection is found in a leaf or the ray exits the bounds of the subtree. A ray that has exited a subtree traverses the acceleration structure until it encounters another queue point – in which case it is queued and another ray from the previous queue starts traversal – or it exits the scene. After eye rays, shadow rays are traced in similar manner. Then, starting from the queue point that contains the intersection point of the eye ray which produced them, reflection and refraction rays are traced in identical manner. These rays, in turn, produce intersection points from which new shadow and secondary rays are spawned. This process limits the amount of active rays in the system. It is continued until there are no more rays to process (be it through all rays leaving the scene, all rays reaching their source of emission, some predetermined termination criteria, or a combination of causes). Unlike recursive ray tracing which seeks to minimize ray traffic, this algorithm seeks to minimize geometry traffic, because ray traffic is faster as long as all threads remain busy. In simulations the described algorithm was able to reduce bandwidth consumption – depending on the scene – by up to 7.8 times compared to packet ray tracing (see Section 3.2) [Navrátil, 2010; Navrátil *et al.*, 2007].

3.2 Ray packets

Many rays traverse a given scene in nearly identical manner, they start from the eye and intersect the same primitive at nearly the same point. Thus it would seem like a good idea to reduce repeating the same traversal steps, memory accesses, etc. by somehow taking advantage of this coherency between rays. Systems such as cone, beam and pencil tracing were the first to utilize ray coherence, but real renaissance was brought about by introduction of SIMD architectures and packets [Arvo and Kirk, 1989; Wald *et al.*, 2001]. SIMD makes it possible to perform the same operation on multiple inputs simultaneously and the width of a SIMD unit tells how many operations it can perform simultaneously. First SIMD units were 4-wide, but, for instance, Intel Larrabee was supposed to have 16-wide SIMD. The idea of a packet – sometimes also known as bundle – is simple; instead of intersecting rays one at a time with primitives, intersect multiple rays with the same primitive. With 4-wide SIMD intersection calculations of four rays with a primitive can be done at the same time resulting in a theoretical speedup of 4x [Wald *et al.*, 2001].

Packets were first introduced to kd-trees by Wald *et al*. [2001]. Implementation for grids was presented by Wald *et al*. [2006] while Mahovsky and Wyvill [2006] and Wald *et al*. [2007b] demonstrated the same for BVH. Culling methods (see Section 3.3) have allowed the size of the packets to expand beyond SIMD size, even up to cache size. Cadet and Lécussan [2007] suggested combined use of kd-trees and BHVs to take advantage of the respective better performances of kd-trees on smaller packets and BVHs on wider packets.

As the packets traverse an acceleration structure some of the rays become inactive (they no longer intersect with the structure). A situation where all but one ray in the packet is inactive is possible, which could in turn lead to slower than single ray traversal due to the overhead introduced by packet traversal. Eye and shadow rays usually exhibit good coherence, but even with packets which contain only eye rays coherence can decline rapidly if the scene contains objects that are small enough. Furthermore, reflection and refraction rays can become totally random in mere three bounces. Because of the aforementioned reasons, packets were initially researched for eye and shadow rays exclusively. While the increases in execution speed have been encouraging, neglecting reflections and refractions misses the whole point of ray tracing. Without reflections and refractions ray tracing is only a more limited and slower version of rasterization and thus not worth using. It is therefore clear that to challenge rasterization reflections and refractions also need to be implemented interactively [Boulos *et al*., 2007].

Månsson *et al.* [2007] tested various methods of sorting secondary rays to coherent packets before tracing them. They concluded that all of the used schemes resulted in comparable or worse execution speed than not sorting at all. Boulos *et al.* [2008] note that determining which rays are coherent before actually tracing them is a difficult task.

Boulos *et al*. [2007] examined four reordering schemes to determine how well they worked when used with secondary rays. The first method – named blind – where all shadow and secondary rays were placed in a single packet was deemed untenable without testing. The reason, as described in Section 2.3, is that a faster intersection method for shadow rays exists when they are traced on their own. In addition, reflected and refracted rays tend to head to wildly different directions and, thus, their coherency is poor. Another method – named group – in which rays are placed in packets based on some shared property – like intersected material – was similarly left unexplored. Group was deemed unfit because depending on the way chosen to divide rays to packets, it could lead to as many packets as there are rays. Of the remaining two methods runs is very similar to group. Like group, rays are placed in packets if they share a property, but in addition their eye rays need to be numerically adjacent. The last method to be evaluated empirically is known as types. In this method rays are placed in packets based on type. Of the two evaluated methods types was found to be faster, typically by 10 – 20%. The better outcome of types is attributed to fewer intersection tests as bounce depth increases and significantly fewer box tests at all bounce depths.

Boulos *et al.* [2008] remove inactive rays from packets to maximize SIMD utilization. Because this reordering of a packet is a costly operation, they perform it only when packet utilization (active rays in a packet divided by total rays in a packet) drops below 50%. To improve tightness after reordering, ray origins are moved to their intersection point with the current AABB and the ray parameter is clipped to the exit point of the AABB. The scheme produces results that are comparable or slower than packet tracing on low bounce depths, but become faster than packet tracing between 5th and 10th bounce. An increase in SIMD width would have a positive effect on the speed of box tests, but triangle tests would see only minor speedups because on average only 2 rays reach a leaf node.

3.3 Culling methods

SIMD size packets increase execution speed considerably, but using even bigger packets would amortize more operations while saving bandwidth. With culling methods operations can be applied to an entire packet once, instead of performing them to each individual ray [Reshetov, 2007].

While interval arithmetic was developed for analyzing tolerances, uncertainties and rounding errors, it can also be used as a culling method. An interval is a set of points between two endpoints. Using intervals, a box B in 2D is a cross product of the intervals on x- and y-axes, and a ray is an interval from origin to ray direction. Intersection with a box is calculated as

$$\begin{aligned} O_x + t_x D_x &= B_x \\ t_x &= (B_x - O_x)\frac{1}{D_x}. \end{aligned} \tag{3.1}$$

The solution for y-axis is obtained similarly. There is an intersection with a box if $t_x \cap t_y \cap t \neq \emptyset$. This is easily expanded to packets by simply using intersection ($\cap$) operation on all results. In interval arithmetic less than and greater than comparisons can, in addition to true and false, also result in undecided. This can happen when intervals of two variables overlap. Generally if a test evaluates to true or false, a single operation is needed for all the rays. Undecided, on the other hand, usually means that rays need to be processed individually [Boulos *et al.*, 2006].

A set of four rays can be used to form a convex shaft which bounds all the other rays inside a packet. These rays can be actual rays in a packet or purely virtual rays. A single SIMD operation can be used to trace the rays. In leaves with triangles, each triangle is tested against the four rays and if a single test fails for all four rays then there can be no intersection with any of the rays in the packet. On the other hand, if no tests fail for all four rays, then the possibility of an intersection exists. The indexes of these triangles can be grouped into a “relevant triangle list” to be tested against later [Boulos *et al.*, 2006; Dmitriev *et al.*, 2004]

Reshetov *et al.* [2005] have demonstrated that the previous approach is flawed in some cases, and can lead to incorrect traversal choices. Another method would be to utilize frustum culling from raster graphics. In this case a convex hull of rays would be used to test for an intersection with axis-aligned boxes. When applied to ray tracing this approach succumbs to ever increasing amounts of trivial rejects as their amount increases with AABB size. To counteract the increase of failures, roles of the frustum and AABBs can be reversed: AABB planes are used to separate it from a frustum. While this does not eliminate failed trivial rejects, they become rarer because an AABB cross section is usually much larger than a frustum cross section. An added advantage of this process is that the rays in a packet do not have to have a shared origin, because rectangular bounds of each axis-aligned plane are used instead of frustum plane normal. While the algorithm does indicate intersections with AABBs where there are none, most of the resulting unnecessary intersection tests can be handled by using two simple tests at leaf nodes:

1. **(**minimum of y-entry values) > (maximum of x-exit values)
2. **(**minimum of x-entry values) > (maximum of y-exit values).

Even though not all redundant intersections are avoided, the remaining calculations do not produce a large negative impact on performance [Reshetov *et al.*, 2005].

Reshetov [2007] presents further improvements in frustum culling. When a packet arrives at a leaf, a frustum – named transient frustum – is computed for the active rays. Thus the frusta are specially made for each situation, which allows them to be very tight. For frustum building, the biggest prevalent axis of rays is used, since using it increases performance by 30% but does not affect the accuracy of computations. Rays in a packet are intersected with two planes on the chosen axis of a leaf. Two axis-aligned rectangles, which contain the intersections for each plane, are computed on the planes, and a frustum is formed between them. The frustum contains intersections between rays and a leaf AABB. If x is the prevalent axis, then the first rectangle is defined by values x_{00}, y_{00}, y_{01}, z_{00}, z_{01} and the second rectangle by values x_{10}, y_{10}, y_{11}, z_{10}, z_{11}. It then follows that the bottom plane normal is

$$\begin{aligned} n_b = ([x_{10}, y_{10}, z_{10}] - [x_{00}, y_{00}, z_{00}]) \times ([x_{00}, y_{01}, z_{00}] - [x_{00}, y_{00}, z_{00}]) = \\ [(y_{01} - y_{00})(z_{00} - z_{10}), 0, (x_{00} - x_{10})(y_{00} - y_{01})]. \end{aligned} \tag{3.2}$$

Determining whether or not a vertex $v = [v_x, v_y, v_z]$ is on the outside of the bottom plane can be done by solving the following equation, because $[x_{00}, y_{00}, z_{00}]$ lies in the bottom plane. The equation can be simplified further by removing the first multiplier $(y_{01} - y_{00})$, because only the sign of the calculation is needed. A negative result means that v is on the outside. Values for the left, right and top plane can be computed similarly. If the value is negative for one or more planes, then the vertex is on the outside of the plane(s) as well as the frustum. If all three vertices of a triangle are on the outside of a frustum then there is no intersection between the frustum and the triangle [Reshetov, 2007]:

$$n_b \cdot \left([v_x, v_y, v_z] - [x_{00}, y_{00}, z_{00}]\right) = (y_{01} - y_{00})(v_z(x_{10} - x_{00}) + x_{00}\, z_{10} - x_{10}\, z_{00} + v_x(z_{00} - z_{10})). \quad (3.3)$$

If a triangle passes the first test, then it can be tested against a near plane to find out if it is behind all ray origins and a far plane to determine if it is further away than the previous intersection. It can also be evaluated for an intersection against corner rays to identify if the frustum is separated from the triangle by any of the triangle edges. The remaining triangles are tested for an intersection with rays in a packet using a ray triangle intersection test. This approach makes it possible to reduce the amount of nodes in a kd-tree by more than 10x and consequently increase the amount of triangles in leaf nodes to hundreds, while still culling more than 90% of all potential ray triangle intersection tests. The reduction in node count is significant, because building a shallower tree should be faster and shallower trees should also increase SIMD utilization. In addition, with hundreds of triangles at leaves it is possible to form meshes at leaves and thus share computations between triangles. For secondary rays, frustum size increases due to diverging rays, which leads to decrease in efficiency as the larger frustum is more likely to intersect with AABBs [Reshetov, 2007].

Overbeck *et al.* [2008] extend frustum culling to secondary rays, by using two planes to bind all rays in a packet. A far plane is chosen from the scene AABB in the positive direction of the dominant ray direction axis. A near plane is chosen from the AABB bounding ray origins, in the negative direction of the same axis. Barycentric coordinates are used to compute intervals for both planes. Corner ray directions are computed as the difference between extremal intersection points with the far plane and extremal intersection points with the near plane. The approach increases performance by 1.2 to 1.3x.

3.4 Ray tracing with stream filtering

RayStream is actually a method and an architectural design, but here the design will be ignored. In RayStream ray tracing is considered as a filtering problem. A stream of some size is passed on to a filter. The filter operates by utilizing SIMD to process *N* amount of rays in parallel. Each ray in a stream is independent, so the order in which the rays are processed has no bearing on the outcome. Each ray is tested against a condition set by the filter. The rays that pass the condition are active and are therefore appended to an output stream which contains only active rays. Rays that do not pass the condition are inactive and they are thus removed from further processing. In case of traversal the filter would test whether or not rays in a stream intersect the current node [Gribble and Ramani, 2008; Ramani *et al*., 2009; Wald *et al*., 2007a].

New output streams are produced until rays intersect geometry in a leaf or the stream runs out of rays that pass the filter condition. In the same fashion the intersection test outputs a stream of rays which intersect with the geometry. For shading, input streams are passed through a filter stack to generate output streams of rays requiring similar actions – for example not intersecting geometry or intersecting a light source. Stacks are used because they produce higher SIMD efficiency when used with long streams and shading has long streams. On the other hand, intersection streams are short and consequently would not benefit from using stacks [Gribble and Ramani, 2008; Ramani *et al*., 2009; Wald *et al*., 2007a].

While the results obtained from this system are only simulations and not based on actual tests, they still give a rough idea of what could eventually be expected. A scene consisting of 2124001 primitives, ideal diffuse reflection and being lighted with 2 light sources was rendered at 15.65 FPS when using 16 wide SIMD and 64x64 ray streams. Furthermore, the authors state that the approach could possibly be combined with culling strategies and memory coherent ray tracing [Ramani *et al*., 2009].

4. Kd-tree building

A kd-tree – also known as k-d tree, k-dimensional binary search tree, multidimensional binary search tree and bin tree – is a form of BSP-tree (Binary Space Partitioning tree, a method of space subdivision) with 0 to $k - 1$ dimensions. Each node has two pointers, a discriminator and a key. The discriminator is the splitting dimension of the node. In 3D scenes $k = 3$ – one dimension for each of the x, y and z axes. Subtrees are built such that the left subtree has values which are lower than the roots along the splitting dimension and the right subtree contains values which are higher [Bentley, 1975; Fussell and Subramanian, 1988]. Keys hold the splitting plane of a node. Splitting planes are axis-aligned because an intersection calculation with an arbitrary plane is more complex and, thus, also slower to execute [Fussell and Subramanian, 1988; Havran, 2001].

Building acceleration structures manually is not only prohibitively slow, but the resulting trees are grossly suboptimal for use in ray tracing [Goldsmith and Salmon, 1987]. There are a number of ways to guide construction of a kd-tree, such as spatial median or object median using round robin or maximum extent, empty space maximizing and cost models or heuristics. Where the spatial median method divides a node in two halves of equal size, object median divides a node so that both resulting subnodes have an equal amount of triangles [Havran, 2001; Wald, 2004]. Round robin selects the next splitting dimension

$$Dim = l \text{ modulo } k, \tag{4.1}$$

where l is any level in the tree [Fussell and Subramanian, 1988]. In maximum extent the dimension with the largest extent is chosen as the splitting dimension. While maximum extent has generally been thought of as an improvement over round robin, Wald [2004] experimentally demonstrated the preconception as wrong. Early research opined that a balanced tree, produced by the object median method, was essential for efficient traversal [Fussell and Subramanian, 1988]. This view has been shown to be false both empirically as well as theoretically. Theoretically the reason is twofold: firstly, probability of accessing each leaf is not equal and, secondly, the search does not necessarily end when the first leaf is reached [Havran, 2001; Wald, 2004].

Of all the currently known methods the most efficient to traverse trees are produced by using a cost model known as surface area heuristic (SAH) [Goldsmith and Salmon, 1987; Wald, 2004]. Thus, this discussion will be limited to the different ways of building kd-trees with SAH and how the building process can be parallelized effectively.

4.1 Conventional method

SAH is an automatic way to determine how to construct a tree. It is based on the knowledge that while three assumptions hold – all rays intersect the scene, the rays are uniformly distributed, and none of the rays intersect an object – the probability of an arbitrary ray intersecting a convex object is proportional to the surface area of said object. Additionally, if convex object X contains another convex object Y ($X \cap Y = Y$) then the probability of an arbitrary ray intersecting Y is the surface area of Y divided by the surface area of X. For trees this means that the probability of an arbitrary ray, which fulfills these assumptions, intersecting a node can be calculated by dividing the surface area of a node with the surface area of the root. Intersection estimates for an arbitrary ray can then be calculated as follows

$$\sum_{i=1}^{N_i} \frac{SA(i)}{SA(root)} \tag{4.2}$$

$$\sum_{l=1}^{N_l} \frac{SA(l)}{SA(root)}, \text{ and} \tag{4.3}$$

$$\sum_{l=1}^{N_l} SA(l) \bullet N(l)/SA(root)\,, \tag{4.4}$$

where N_i = number of interior nodes, N_l = number of leaves, $N(l)$ = number of objects stored in leaf l, $SA(i)$ = surface area of interior node i and SA(l) = surface area of leaf node l [Goldsmith and Salmon, 1987; MacDonald and Booth, 1990; Havran, 2001]. These equations can be combined to a single equation, which can be used to determine the total cost of an arbitrary ray if the costs of the operations involved are known:

$$\frac{C_i \bullet \sum_{i=1}^{N_i} SA(i) + C_l \bullet \sum_{l=1}^{C_l} SA(l) + C_o \bullet \sum_{l=1}^{N_l} SA(l) \bullet N(l)}{SA(root)}, \tag{4.5}$$

where C_i = cost of traversing an interior node, C_l = cost of traversing a leaf and C_o = cost of testing an object for intersection. Because rays are assumed to not intersect any objects, the derived estimate is an upper bound [MacDonald and Booth, 1990; Harvan, 2001]. Due to the prohibitively expensive cost of determining a globally optimal tree for anything but the most minimal of scenes, a local greedy approximation – which assumes that both children produced by the subdivision become leaves – is usually used. The local version is described as

$$C_t + C_o\left(\frac{SA_L}{SA(root)}|T_L| + \frac{SA_R}{SA(root)}|T_R|\right), \tag{4.6}$$

where C_t = cost of a traversal step, SA_L = surface area of the left node, SA_R = surface area of the right node, T_L = triangles in the left node and T_R = triangles in the right node. An inefficient tree is built if two special cases are not accounted for. Firstly, a triangle may overlap a voxel in only a point or a line and it should thus get culled. Secondly, triangles lying in a plane should not end up in both new voxels. These cases can be accounted for by dividing triangles into three groups: those which are to the left of the plane (T_L) , those which are to the right of the plane (T_R), and those which are on the plane (T_P). Equation 4.6 is then separately evaluated with $T_L + T_P$ and $T_R + T_P$ and the lowest cost is selected [Wald and Havran, 2006]. Mailboxing (see third paragraph down from this one) can be accounted for by subtracting the number of objects that overlap the splitting plane times the probability that a ray traverses both child nodes from Equation 4.6. This simple optimization reduces ray-primitive intersections by about 30% on average, improves tracing time by few percent while moderately increasing the number of traversal steps [Hunt, 2008a].

The original equations, while accepting the practical importance of the matter, assume that no object is placed so as to be split by a plane and thus ending up in both of the new nodes. While this assumption holds it can be shown that the split position with the lowest cost estimate can be found between object and spatial medians [MacDonald and Booth, 1990; Harvan, 2001]. Havran [2001] demonstrated that if the previous assumption is discarded for a more realistic one where some objects are assumed to straddle the splitting plane, then the lowest cost estimate can be found either at the boundaries of the real objects or at the boundaries of the objects' bounding box. These are known as events or split candidates. When using this cost estimate, searching for the minimum value between spatial and object medians produces a 3% decrease in traversal time.

The assumption that rays do not intersect any object is clearly impractical and means that the algorithm is based on considering the worst case scenario. Thus equations which describe the situation more realistically might lead to better traversal performance. A more accurate cost model was developed by Havran [2001]. The tests conducted with three different configurations of the new model showed that at best it could achieve a reduction of 1% in traversal time and at worst it considerably slowed down traversal. The fastest configuration of the new method involved sampling in addition to more complex calculations during construction leading to an increase in building time.

An object does not necessarily reside inside a single voxel. When an object has parts of it inside two or more voxels, it will also have multiple, unnecessary, intersection tests with the same ray. This can be prevented by using a mailbox. Each object has a mailbox which stores the unique ID of a ray and the result of the intersection test with the ray. Before executing an intersection test with a ray, the ID of the ray and the ID stored in the mailbox are compared. If they match then the stored result can be used as is. Otherwise, an intersection calculation needs to be performed [Arvo and Kirk, 1989; Wald, 2004]. This simple approach is not valid for multiple threads, as changing a single value in the triangle data would invalidate an entire cache line in the other processors. Instead of storing the mailbox with the triangle data, storing it (it holds both the ray and triangle ID) as a hash table with the thread has been suggested. While this does solve the problem, the solution is not as fast as standard mailboxing [Shevtsov *et al.,* 2007b; Wald *et al.,* 2001; Wald, 2004]. Inverse mailboxing seeks to rectify the speed problem by storing the IDs of the last eight triangles visited by a ray packet in a ring buffer. The data is thread-local and allows traversal by multiple threads simultaneously. The results show an increase in execution speed of 9.7 to 23.4% (tested on three models) [Shevtsov *et al.*, 2007b]

Most scenes will have some measure of empty space in them. BVHs do not process this space by virtue of only encompassing objects. On the other hand, space subdivision measures such as kd-tree process the entire scene. Empty space, however, does not need to be traversed. Guiding the construction of a kd-tree so that empty space is collected to leaf nodes on the upper levels of a kd-tree should make it possible to skip large parts of the scene during traversal [Havran, 2001]. Havran [2001] experimented with three ways to cut of empty space and concluded that when automatic termination as described in the paper is used, the tested methods are not likely to provide increased performance. An easy way to cut of empty space is to encourage SAH to choose splits where the right or left subnode is empty, by reducing the cost of such splits by some constant amount. The following rule is a suggestion for setting the constant [Hurley *et al.*, 2002; Wald and Havran, 2006]:

$$\lambda(p) = \begin{cases} 80\% & ; |T_L| = 0 \; OR \; |T_R| = 0 \\ 1 & ; otherwise. \end{cases} \tag{4.7}$$

While Hurley *et al.* [2002] report a 5% improvement in traversal using this method, Bikker [2007] observes only a minimal improvement. Algorithm 4.1.1 describes the SAH evaluation process with empty space cutting.

```
SAH evaluation
Algorithm  Cost(probability_L, probability_R, trianglesLeft, trianglesRight)
Input: probability of left subvoxel, probability of right subvoxel, number of triangles left subvoxel, number of triangles in right subvoxe
Output: cost of plane position
   return λ(p)( $C_t$ + $C_o$(probability_L • trianglesLeft + probability_R • trianglesRight))

Algorithm SAH(plane,V,trianglesLeft,trianglesRight,trianglesPlane)
Input: plane position, voxel V, triangles to the left, right and on the plane position
Output: best plane position
   ($V_L$,$V_R$) ← splitBox(V,plane)
   probability_L ← SA($V_L$)/SA(V)
   probability_R ← SA($V_R$)/SA(V)
   $C_L$ ← Cost(probability_L, probability_R, trianglesLeft + trianglesPlane, trianglesRight)
   $C_R$ ← Cost(probability_L, probability_R, trianglesLeft, trianglesPlane + trianglesRight)
   if $C_L$ < $C_R$ then
      return  ($C_L$, left)
   else
      return ($C_R$, right)
    end if
```

Algorithm 4.1.1 SAH [Wald and Havran, 2006]

A situation where the node and object bound boxes intersect without the actual object intersecting the node bound box is possible. In such a situation there will be unnecessary ray-object intersection tests. In addition, the trees that are produced are of inferior quality for traversal, as the tree building process will be skewed due to an inaccurate cost estimate. Three ways – post processing, intersection tests also in interior nodes and split clipping – have been suggested to correct this defect. Of the three, split clipping is considered to be the most viable one. In split clipping, when an object straddles a splitting plane the extent of the objects bounding box on both sides of the splitting plane is minimized. While these perfect splits increase traversal performance by 9% on average, they also correspondingly increase tree building time by 140% [Havran, 2001; Havran and Bittner, 2002; Wald and Havran, 2006]. A detailed description and code of a significantly faster split clipping method is given by Soupikov *et al.* [2008]. Their implementation also retains the increased traversal performance of the original method. Figure 4.1 depicts the idea behind split clipping.

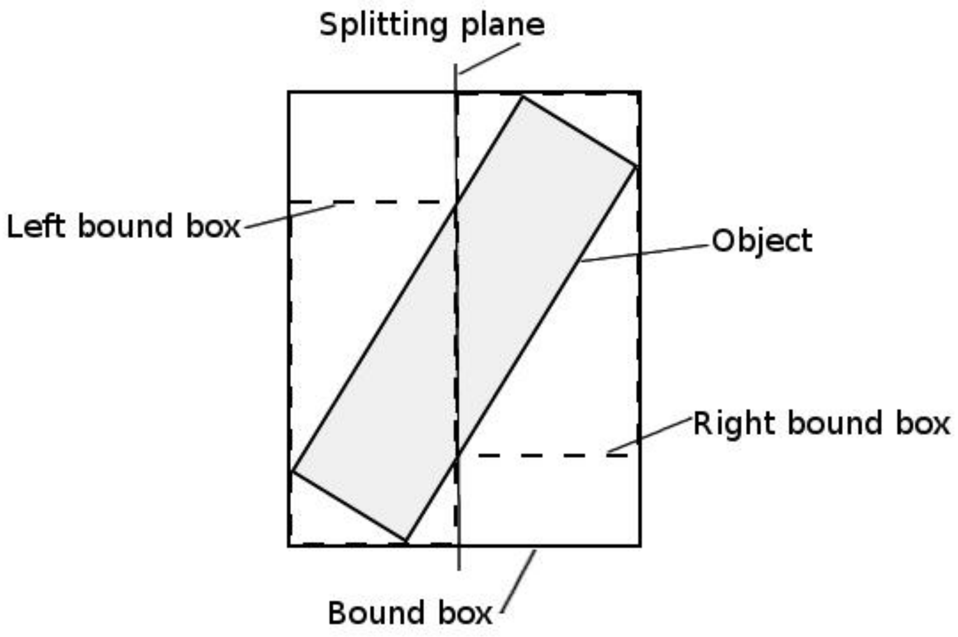

Figure 4.1 Split clipping

Determining when to stop subdivision has traditionally been done by defining a maximum depth or a maximum number of objects for a node. A node becomes a leaf when either of these user defined constants is reached. Limiting the tree depth to a predefined constant clearly cannot be optimal for every possible scene, but it does have the advantage of bounding memory use to a known value [Havran, 2001]. Havran [2001] develops an automatic termination criteria based on previous research, which has shown that every scene has a maximum tree depth after which further subdivision provides no additional benefit. This point depends on the scene. The equation of the criteria depends on two constants which are determined experimentally and it does improve traversal performance. However, for the tests the constants were determined by experimenting on the test scenes. So, it is unclear how well this approach would work on different scenes. Hurley *et al.* [2002] and Wald [2004], on the other hand, propose simply comparing the cost of traversing a split object and the cost of not splitting a voxel and subdividing if the cost of traversing is lower than the cost of not splitting. However, it is possible to get stuck in local minima if the problem is not accounted for [Wald and Havran, 2006]. Reshetov *et al.* [2005] note that even with automatic termination, creation of small cells most likely to be missed by a single ray, should be prevented.

Plane selection is a slow process as the amount of possible split positions is (in principle) infinite. Even when using only the planes which define the bounding box of a triangle, each voxel with N triangles has $6N$ split candidates which need to be considered. The most straightforward way to find the best split position is to iterate over all triangles in a voxel and compute the cost estimate for each split candidate [Wald and Havran, 2006]. Algorithm 4.1.2 describes this $O(N^2)$ approach.

$O(N^2)$ plane selection

```
Algorithm PerfectSplits(t, V)
Input: triangle t, voxel V
Output: clipped object
   B ← Clip t to V {consider “perfect” splits}
   return k ← 1..3((k,$B_{min}$,k) OR (k,$B_{max}$,k))

Algorithm Classify(T, $V_L$,$V_R$,plane)
Input: list of triangles T, voxels to the left and right of the splitting plane, the
splitting plane
Output: counts of triangles to the left, right and on the plane
   $T_L \leftarrow T_R \leftarrow T_P \leftarrow \emptyset$;
   foreach triangle t in T
      if t lies in plane AND Area(plane OR V ) > 0 then
         $T_P \leftarrow T_P$ OR t
      else
         if Area(t AND ($V_L$ NOT plane)) > 0 then
            $T_L \leftarrow T_L$ OR t
         end if
         if Area(t AND ($V_R$ NOT plane)) > 0 then
            $T_R \leftarrow T_R$ OR t
         end if
      end if
   end foreach
   return $T_L$, $T_R$, $T_P$

Algorithm NaıveSAH::Partition(T, V)
Input: list of triangles T, voxel V
Output: best splitting plane, triangles to left and right of the plane, one OR’d with
triangles on the plane
   foreach triangle t in T
      bestCost ← ∞
      $bestPlane_{side}$ ← ∅ {initialize search for best node}
      foreach plane in PerfectSplits(t, V )
         ($V_L$, $V_R$) ← split V with plane
         ($T_L$, $T_R$, $T_P$ ) ← Classify(T, $V_L$, $V_R$, plane)
         (cost, $plane_{side}$) ← SAH(V, plane, |$T_L$|, |$T_R$|, |$T_P$ |)
         if cost < bestCost then
            ( bestCost, $bestPlane_{side}$ ) ← (cost, $plane_{side}$)
         end if
      end foreach
   end foreach
   ($T_L$, $T_R$, $T_P$) ← Classify(T, $V_L$, $V_R$, plane)
   if ($bestPlane_{side}$ ← LEFT) then
      return (bestPlane, $T_L$ OR $T_P$, $T_R$)
   else
      return (bestPlane, $T_L$, $T_R$ OR $T_P$)
   end if
```

Algorithm 4.1.2 $O(N^2)$ plane selection [Wald and Havran, 2006]

Algorithm 4.1.2 is too slow to be used in any meaningful scene. The time complexity of the process can be reduced to $O(N \log^2 N)$ by changing how N_L, N_R and N_P are determined. For each dimension all triangles are iterated over and it is determined whether a triangle is axis-aligned. Axis-aligned triangles produce a "planar event" ($p^|$), and other triangles produce a "start event" (p^+) and an "end event" (p^-). Each event keeps track of its type, plane position and the triangle that generated it. The list of events (E) is then sorted to ascending order by plane position. In case of equal plane positions the order of events is end, plane, start. All that is left is to sweep over the possible plane positions and determine the best split position. At the start of the sweep N_L, N_R and N_P have values

$$N_L^{(0)} = 0 \quad N_P^{(0)} = 0 \quad N_R^{(0)} = |T|.$$

The counts are updated with the following rules:

$$\begin{aligned} N_L^{(i)} &= N_L^{(i-1)} + p_{i-1}^{|} + p_{i-1}^{+}, \\ N_R^{(i)} &= N_R^{(i-1)} - p_i^{|} - p_i^{-}, \\ N_P^{(i)} &= p_{i-1}^{|}. \end{aligned}$$

Algorithm 4.1.3 describes this plane selection process.

***O*(*N* log² *N*) plane selection**

```
Algorithm  FindPlane(T, V)
Input: list of triangles T, list of voxels V
Output: best plane position
  bestCost ← ∞, bestPlane ← ∅
  for k ← 1 to 3  do
    eventlist E ← empty
    foreach  triangle t in T
      B = AABB(t )
      if B is planar then
        E.add(event(t,$B_{min,k}$, |
      else
        E.add(event(t,$B_{min,k}$, +)
        E.add(event(t,$B_{max,k}$, −)
      end if
    end foreach
    sort(E,<E) {sort all planes according to <E}
    trianglesLeft ← 0, trianglesPlane ← 0, trianglesRight ← |T|
    for i ← 0 to i < |E| do
      plane ← $E_{i,plane}$
      trianglesStarting ← trianglesEnding ← trianglesLying ← 0
      while i < |E| && $E_{i,planePos}$ = $plane_{planePos}$ && $E_{i,type}$ = ending
        trianglesEnding++
        i++
      end while
      while i < |E| && $E_{i,planePos}$ = $plane_{planePos}$ && $E_{i,type}$ = lying
        trianglesLying++
        i++
      end while
      while i < |E| && $E_{i,planePos}$ = $plane_{planePos}$ && $E_{i,type}$ = starting
        trianglesStarting++
        i++
       end while
      trianglesPlane ← trianglesLying
      trianglesRight ← trianglesRight - trianglesLying - trianglesEnding
      (cost,$plane_{side}$) ← SAH(V, plane,$N_L$,$N_R$,$N_P$)
      if cost < $bestPlane_{cost}$ then
        (bestCost,bestPlane,$bestPlane_{side}$) = (cost, plane, $plane_{side}$)
        trianglesLeft ← trianglesLeft + trianglesLying + trianglesRight
        trianglesLying ← 0
      end if
    end for
  end for
  return (bestPlane, $bestPlane_{side}$)
```

Algorithm 4.1.3 *O*(*N* log² *N*) plane selection [Wald and Havran, 2006]

Algorithm 4.1.3 is also too slow for use in real-time applications. The lower limit *O*(*N* log *N*) of sorting based algorithms can be reached if an event list containing events for each dimension is sorted only once and the plane finding algorithm is applied to the sorted list. The events are sorted based on plane position as before and the same sort order is maintained, but events of same dimension are stored together [Wald and Havran, 2006]. Algorithm 4.1.4 describes this process.

***O*(*N* log *N*) best plane**

```
pre: E is <E-sorted
Algorithm FindPlane(N, V , E)
Input: list of triangles T, list of voxels V, presorted list of events E
Output: best plane position
  bestCost ← ∞
  bestPlane ← none
  foreach dimensions k
    $trianglesLeft_k$ ← 0, $trianglesPlane_k$ ← 0, $trianglesRight_k$ ← |T|
  end foreach
  for i ← 0 to i < |E| do
    plane ← ($E_{i,plane}$, $E_{i,k}$)
    trianglesStarting ← trianglesEnding ← trianglesLying ← 0
    while i < |E| && $E_{i,k}$ = $plane_k$ && $E_{i, planepPos}$ = $plane_{planePos}$ && $E_{i,type}$ = ending
      trianglesEnding++
      i++
    end while
    while i < |E| && $E_{i,k}$ = $plane_k$ && $E_{i, planepPos}$ = $plane_{planePos}$ && $E_{i,type}$ = lying
      trianglesLying++
      i++
    end while
    while i < |E| && $E_{i,k}$ = $plane_k$ && $E_{i, planepPos}$ = $plane_{planePos}$ && $E_{i,type}$ = starting
      trianglesStarting++
      i++
    end while
    $trianglesPlane_k$ ← trianglesLying
    $trianglesRight_k$ ← $trianglesRight_k$ - trianglesLying - trianglesEnding
    (cost,$plane_{side}$) ← SAH(V, plane,$N_L$,$N_R$,$N_P$)
    if cost < bestCost then
      (bestCost,bestPlane,$bestPlane_{side}$) = (cost, plane, $plane_{side}$)
      $trianglesLeft_k$ ← $trianglesLeft_k$ + trianglesLying + trianglesRight
      $trianglesLying_k$ ← 0
    end if
  end for
return (bestPlane, $bestPlane_{side}$)
```

Algorithm 4.1.4 *O*(*N* log *N*) best plane selection [Wald and Havran, 2006]

To maintain *O*(*N* log *N*) time complexity classification and building of the event lists of subnodes has to be done without sorting. To do this all triangles are first marked as belonging to group 'both'. The event list is then iterated over and events which match the classification of T_L or T_R are marked as belonging to their respective groups. The event list is then split into two sorted sublists – the sublists are sorted because E is sorted – one containing left only events (E_{LO}) and the other containing right only events (E_{RO}). Triangles marked both are split clipped and they form two new unsorted lists – one for left side and one for right. These lists can be sorted in *O*(*N*) time due to the assumption that $O(\sqrt{N})$ triangles overlap the splitting plane. The sorted lists can then be merged in *O*(*N*) [Wald and Havran, 2006]. Algorithm 4.1.5 describes this process.

Triangle classification

```
Algorithm ClassifyLeftRightBoth(T,E, bPlane)
Input: list of triangles T, presorted list of events E, splitting plane bPlane
Output: -
  foreach triangle t in T
    $t_{side} \leftarrow$ both;
  end foreach
  foreach event e in E
    if $e_{type}$ = ending && $e_k = bPlane_k$ && $e_{planePos} \le bPlane_{planePos}$ then
      $t[e_t]_{side} \leftarrow$ LeftOnly
    end if
    else if $e_{type}$ = starting && $e_k = bPlane_k$ && $e_{planePos} \ge bPlane_{planePos}$ then
      $t[e_t]_{side} \leftarrow$ RightOnly
    end if
    else if $e_{type}$ = lying && $e_k = bPlane_k$ then
      if($e_{planePos} < bPlane_{planePos}$ || ($e_{planePos} = bPlane_{planePos}$ && $bPlane_{side}$ = left)) then
        t[et]side $\leftarrow$ LeftOnly
      end if
      if($e_{planePos} > bPlane_{planePos}$ || ($e_{planePos} = bPlane_{planePos}$ && $bPlane_{side}$ = right)) then
        $t[e_t]_{side} \leftarrow$ RightOnly
      end if
    end if
  end foreach
```

Algorithm 4.1.5 Triangle classification [Wald and Havran, 2006]

Even though the combined use of Algorithms 4.4 and 4.5 reaches the asymptotic lower bound of sorting based approaches, the building times are not fast enough for use in real-time applications. Just doing 30 rebuilds per second requires that each rebuilding takes 33.33 ms and to reach 60 rebuilds per second each rebuilding can take only 16.67 ms. Building a kd-tree for a model with 804 triangles with Algorithms 4.4 and 4.5 took 30 ms and for a more reasonable model of ~69k triangles the building time was 3.2 s [Wald and Havran, 2006].

4.2 Approximations and parallel building

The first attempt to build an acceleration structure in parallel was done by Benthin [2006]. He parallelized the conventional construction algorithm by dividing split candidate lists into sublists with the amount of threads. The threads then proceed to sort the sublists and the sorted sublists are merged into one. Kd-tree building phase was parallelized by splitting the first levels with one thread and then building the remaining sub kd-trees with individual threads. This scheme reached 2x speedup on a dual core system.

To further increase building speed, algorithms which approximate SAH have been developed. Even though most of the following scanning based algorithms have a time complexity of $O(N \log N)$ the constant factors of scanning based approaches are lower than those of sorting, which leads to better performance. Scanning also defers work to leaf nodes so greater amount of unneeded work is avoided if lazy building is used. Popov *at al*. [2006] developed a streaming construction algorithm which uses triangle AABBs instead of the actual triangles and utilizes both BFS and DFS building. The algorithm starts with BFS building and proceeds to some predefined depth after which a switch to the conventional DFS algorithm (as described by Wald and Havran [2006]) is performed. The switch depth is chosen such that the DFS algorithm can retain locality and, thus, overcome performance degradation caused by the random memory access pattern of triangle classification stage. In the BFS section of the algorithm memory access is sequential because triangle AABBs are stored in a continuous memory array. Nodes of the current level are associated with a partition from the array. The array is swept and, for each sample, the amount of AABBs which end between the sample and the one preceding it, as well as the ones which start between the sample and the one following it, are recorded. There are 1024 uniformly distributed samples for all three axes. Uniform distribution is used because sample indexes of events can be calculated from event positions. AABB count updates are performed as follows:

$$\begin{aligned} n_l^{i+1} &= n_l^i + S^i, \\ n_r^{i+1} &= n_r^i - E^i, \end{aligned} \tag{4.8}$$

where n_l^i = number of AABBs to the left of sample *i*, n_r^i = number of AABBs to the right of sample *i*, S^i = number of AABBs that start between samples *i* and *i* + 1 and E^i = number of AABBs that end between samples *i* and *i* + 1. Once the best split has been determined, triangles which are to the left (resp. right) of the splitting plane are copied to a partition of the left (resp. right) node in a second array, and the ones straddling the plane are clipped and copied to both. The cost function is sampled during the copy. Further refinement of the cost estimate would be possible by using adaptive resampling. The overall upper bound of the cost function would be set to be the minimum upper bound found at the sample points. Intervals with a lower bound higher than the overall upper bound would be eliminated from further processing and the remaining intervals would be resampled. The process can be repeated until only one split location remains. Because their method, as compared to exact SAH evaluation, only slowed traversal at most by 2.2%, Popov *et al*. [2006] did not implement adaptive resampling. The algorithm does not manage much better than the conventional algorithm on small presorted scenes with even sized triangles (models produced by 3D scanning). With increased size and no presorting the algorithm reaches up to 48% speedup in building time. The approach was parallelized by using single threads to build trees in the DFS part of the algorithm. The achieved speedup was at best 2.43x on 4 cores. Algorithm 4.2.1 describes streaming build.

***O*(*N* log *N*) streaming SAH build**

```
Algorithm UPDATESAMPLESTATISTICS(aabb, statistics)
Input: axis-aligned bounding boxes, counter
Output: -
  $l_{xyz} \leftarrow$ indexes of samples just below min point of aabb
  $u_{xyz} \leftarrow$ indexes of samples just above max point of aabb
  foreach dimension do
    Increase statistics.ob jStart[$l_{[dim]}$]
    Increase statistics.ob jEnd[$u_{[dim]}$]
  end foreach

Algorithm GETSPLITLOCATION(stat)
Input: number of starting and ending aabbs
Output: best split location
  stat.oLe f t[0] ← 0
  stat.oRight[0] ← #objects
  for i ← 0 to len(stat) do
    stat.oLe f t[i] ← stat.oLe f t[i − 1] + stat.ob jStart[i]
    stat.oRight[i] ← stat.oRight[i − 1] − stat.ob jEnd[i]
  end for
  Evaluate the cost function at stat
  return The best found split location at stat

Algorithm construct(root_node, boxes)
Input: root node, bounding boxes of all triangles
Output:-
aabbIn ← boxes
stat ← 0
UPDATESAMPLESTATISTICS(aabbIn, stat)
levelNodesIn ← {root_node,GETSPLITLOCATION(stat)}

while levelNodesIn != null do
  nextLevelAABB ← null
  nextLevelNodes ← null
  foreach {node, split} in levelNodesIn do
    if #objects in node < threshold then
      Run conventional build routine for subtree
```

```
        end if
        curAABBIn ← node's partition of aabbIn
        $len_r$, $len_l$ ← #objects in the subtrees of node
        Allocate $len_l$ + $len_r$ space at end of nextLevelAABB
        $stat_l$ ← 0
        $stat_r$ ← 0
        foreach aabb in curAABBIn do
          if left then
            Add aabb to left child's partition in nextLevelAABB
              if not completely to the right of the split plane then
                clip aabb if necessary
              end if
          UPDATESAMPLESTATISTICS(aabb, $stat_l$)
          else
            Add aabb to right child's partition in nextLevelAABB
              if not completely to the left of the split plane then
                clip aabb if necessary
              end if
            UPDATESAMPLESTATISTICS(aabb, $stat_r$)
          end if
        end for
        Create nodes $N_l$ ,$N_r$ for the two subtrees
        nextLevelNodes ←+ {$N_l$ ,GETSPLITLOCATION($stat_l$)}
        nextLevelNodes ←+ {$N_r$,GETSPLITLOCATION($stat_r$)}
      end for
      levelNodesIn ← nextLevelNodes
      aabbIn ← nextLevelAABB
    end while
```

Algorithm 4.2.1 *O*(*N* log *N*) streaming SAH build [Popov *et al.*, 2006]

Another approach, by Hunt *et al.* [2006], samples C_L, C_R, SA_L and SA_R at q uniformly distributed locations per axis to produce approximations of the cost function. As the cost function can have discontinuities, a second scan over the primitives is done. Additional q samples per axis are taken during the second scan from segments (segments are the range between sample points in the initial scan) with a large change in $C_L - C_R$. Hunt *et al.* [2006] suggest that q = 8 is an adequate number of samples. This means that that a total of $2q$ samples are taken per axis. For each axis $C_L - C_R$ can have n different values, for each such value a bin is created to mark a sample location. The amount of adaptive samples dedicated for a segment is determined by the amount of changes within a segment. When $C_L - C_R$ crosses a bin boundary (the value changes) within a segment the amount of samples for the segment is increased by one. The resulting samples per segment are placed at the sample locations so that they are evenly spaced within the segment. The resulting evaluations are then used to generate a piecewise quadratic approximation of the cost function and the split plane is positioned at the minimum of the approximation. This minimum does not need to reside at a previously sampled location. Memory accesses of the algorithm as well as the number of performed scans can be reduced by doing the cost evaluation in SIMD fashion and by maintaining additional containers for C_L and C_R values associated with additional sample points. Scan combining is beneficial until all temporary variables fit in registers. The algorithm switches to exact SAH evaluation when a node contains 36 or fewer AABBs. The Bunny model (see Appendix A) is built in 110 to 250 ms on a single 2.4 GHz core depending on whether the longest axis, all axes or a combination of the two is used. The increase in tracing time ranges from 0.33% to 7.90% depending on scene size and the combination of axes used during building. Hunt *et al.* [2006] assert – but have yet to demonstrate – that the algorithm can be run in parallel by dividing AABBs to threads which then evaluate the splits. A gather operation would combine the results.

Further improvement in building times can be achieved by combining a scanning based approach with a lazy building from hierarchy. Unlike the conventional building algorithm which must process all N primitives in a scene during the sort of the first split, scanning based algorithms can consider only a part of the scene. Thus, with scanning a scene can be built lazily. A lazy system builds the acceleration structure only for the visible primitives of the scene. Scene acceleration structures are generally completely rebuilt for every frame. This approach is based on the view that a scene can undergo a total transformation from frame to frame. However, using for example an acceleration structure of a previous frame as a kind of presort of scene geometry would allow a linear time building of the new acceleration structure. These two approaches produce an asymptotic time complexity of $O(v + \log N)$ where v is the number of visible primitives. Impact on trace time is small (1302 ms for the new algorithm against 1283 ms for the conventional algorithm on a scene with $N = 541023$ and $v = 9392$ on a single 2.667 GHz core) while building time is greatly improved (116ms for the new algorithm versus 12270ms for the conventional, N and v as before) [Hunt *et al.*, 2006; Hunt *et al.*, 2007].

Another approximation approach was presented by Shevtsov *et al.* [2007a]. They used min-max binning (also known as pigeonhole sorting and bucket sorting) on AABBs. AABBs are stored as structures of arrays of bounds. For each level of the tree, 32 bins function as counters. The bins could also store primitive references, but counters are more efficient. Two sets of bins are used: a bin in the first set is updated where an AABB begins and, correspondingly, one bin in the second set is updated where an AABB ends. At the higher levels of the tree only every l-th ($l = \log_{10}(M)$, M = number of primitives in the current node) primitive is considered. SAH is estimated at bin boundaries with min-bins representing primitives to the left of the split and max-bins those to the right of the split. The position of the splitting plane is adjusted if it is warranted by empty space consideration and then the primitives are divided to subnodes. Switch to using the conventional algorithm is done when the amount of primitives in the current node is less than or equal to the number of bins used. The tree is stored as chunks which are linked into lists. Each chunk has a start pointer, chunk size and an end pointer.

The building process is parallelized by dividing the scene into disjoint regions in parallel in a phase called initial clustering. Initial clustering uses 512 * *T* bins, where *T* is the number of threads. Higher amount of bins is used because no rebinning is done during initial clustering. Arrays of primitives for each region are then created in parallel. Construction of the local kd-tree is split into smaller tasks and threads insert and fetch jobs from a shared task pool. While this method builds the Bunny model in 104 ms on a single 3 GHz core and achieves a 3.9x speedup on 4 cores, reaching a building speedup of 120-300x over a highly optimized kd-tree, the tracing speed is only 70% of that of exact SAH evaluation. This is because on the upper levels of a tree the algorithm uses object median split. Object median is used until building of the local kd-trees starts, because it produces subtrees of equal size effectively load balancing the parallel phase and because accurate SAH is more important at the deeper levels of a tree [Shevtsov *et al.*, 2007a].

Although methods which build top levels of a tree with a single core until there are enough subtrees to utilize all cores have achieved some limited parallelization speedups, the lack of demonstrated near linear scalability with higher core counts while retaining trace speed is not their only problem. Another issue is that as the amount of cores keeps increasing, the level at which all cores can be utilized gets further and further down in the tree. With sufficient increase in the amount of cores, the building process will be mostly done before all cores are utilized. Therefore an algorithm which is capable of using more cores on the top levels of a tree is needed [Choi *et al.*, 2010].

In the conventional algorithm (see Algorithms 4.1.4 and 4.1.5) node level parallelism can be complemented with geometry level parallelism. FindBestPlane is executed in three phases. First, the event list is divided into continuous chunks. For each chunk, the amount of start and end edges is counted in parallel. Secondly, a single thread determines the correct N_L and N_R values at the beginning of each chunk by summing the totals of previous chunks with the current chunk. Lastly, SAH value for each chunk is determined as in the first phase, after which the overall minimum SAH is easily deduced. ClassifyTriangles stage is not parallelized, because due to false-sharing the resulting performance improvement proved insignificant. FilterGeom stage is handled similarly to FindBestPlane stage. Geometry is separated into chunks and the number of triangles lying to left and to right of the splitting plane in each chunk is determined in parallel. All chunks are then updated by a single thread so that it is known how many triangles exists to the left and to the right of the splitting plane before this chunk. As the starting position of each chunk is now known, arrays for triangles to the left and to the right of the splitting plane can be updated in parallel. Triangles which straddle the splitting plane are copied to both arrays. This approach – called nested algorithm – was tested on a 32 core 2 GHz system. It built the Bunny in 68 ms, achieving over 5x speedup on 20 cores. At best the algorithm was able to reach 8x speedup on 20 cores when building the Angel model (see Appendix A). The results, however, showed that the algorithm had already reached its peak performance and that it exhibited decreased performance with increased core count past 20 cores. When run on all 32 cores, all test scenes showed decreased performance past 20 cores. Additionally, because the algorithm is sequential, like the conventional algorithm it is based on, its maximum speedup is bounded by Amdahl's Law [Amdahl, 1967; Choi *et al*., 2010]. Also like in the conventional algorithm, there is a lot of data movement when triangle and event lists are shifted from a parent to its children [Choi *et al*., 2010].

To counteract the negative aspects of their nested algorithm, Choi *et al.* [2010] also developed another algorithm called in-place. Data movement problem from nested algorithm is solved by associating nodes with triangles instead of triangles with nodes. This means that triangles track which nodes they belong to and events have a pointer to the triangle that generated them. Triangles are stored as a structure-of-arrays, where elements have pointers to their six events and a list of the current level's nodes to which it belongs. The algorithm proceeds in four phases: FindBestPlane, NewGen, ClassifyTriangles and Fill. While NewGen generates the next level of a tree and Fill translates the tree to the format produced by the sequential algorithm, most of the work is done by FindBestPlane and ClassifyTriangles. As in-place is a BFS algorithm, it processes entire levels at a time, where nodes at the current level are known as "live". FindBestPlane considers all "live" nodes in parallel and determines a split for those that do not become a leaf. ClassifyTriangles updates triangle node lists which can be done in parallel because all the required information is local. Once the current level has enough nodes, a switch to the parallel DFS building is made (see, e.g., Benthin [2006]). While this approach achieves at most 7x speedup (on the Fairy model – see Appendix A) on the same hardware as nested, building the Bunny in 50 ms, it achieves its peak later, at 24 cores. It only encounters a slight drop in performance with increased cores, unlike nested which saw a drop from 8x to 7x speedup when moving from 20 cores to 32. Choi *et al.* [2010] state that in-place does not actually saturate at 24 cores but is instead hampered by limited system resources. To confirm this they ran additional tests which according to them verified the hypothesis. Unfortunately the test results are not included in the paper [Choi *et al.*, 2010]. Algorithm 4.2.2 presents in-place.

In-place algorithm

```
Input: list of triangles in the scene
Output: pointer to the root of the constructed kd-tree
live ← {root ← new kdTreeNode() };
foreach triangle t in T do
   t.nodes ← {root};
end foreach
while nodes at current level < cores do
   foreach e in E[x] && e in E[y] && e in E[z] do
      foreach node in e.t.nodes do
         SAH ← CalculateSAH(e, node.extent);
         if SAH is better than node.bestSAH then
            node.bestEdge ← e ;
            node.bestSAH ← SAH ;
         end if
      end foreach
   end foreach
   nextLive ← { };
   foreach node in live do
      if node.bestEdge found then
         nextLive ←+ (node.left ← new kdTreeNode()) ;
         nextLive ←+ (node.right ← new kdTreeNode()) ;
      end if
   end foreach
   foreach t in T do
      oldNodes ← t.nodes ;
      clear ← t.nodes ;
      foreach node in oldNodes do
         if no node.bestEdge found then
            insert t in node.triangles ;
         else
            if t left of node.bestEdge then
               insert node.left in t.nodes ;
            end if
            if t right of node.bestEdge then
               insert node.right in t.nodes ;
            end if
         end if
      end foreach
   end foreach
   live ← nextLive;
end while
foreach t in T do
   foreach node in t.nodes do
      insert t in node.triangles;
   end foreach
end foreach
return root
```

Algorithm 4.2.2 In-place by Choi *et al.* [2010]

Instead of using all threads to either building or tracing, Yang *et al.* [2008] suggest dedicating a portion of the threads to building and using the rest for tracing. Their system does not require a new kd-tree for every frame, but instead, counts on deforming a tree until quality deterioration forces a rebuild. Whether to update or rebuild is determined by their "rebuild heuristic". The heuristic works by comparing the current SAH value of a node (current SAH values are stored in each node) with the SAH value the node would receive now and rebuilding if the value is above some threshold. The authors claim – citing Lauterbach *et al.* [2006] – that the approach works because quality degradation is not bad for the first few frames. However, no such statement for kd-trees can be found in the Lauterbach *et al.*'s [2006] article. Instead, when talking about updating, the article references spatial kd-trees [Lauterbach *et al.*, 2006]. The system consists of tracing and building threads which are connected to a shared memory buffer. The buffer contains vertex positions and kd-tree nodes. The tracing threads work with the current kd-tree until a new tree is finished, and the new tree is passed to the tracing threads as they finish the current frame. For every frame, vertices need to be updated and the SAH cost of intersected nodes is recomputed. Rebuilding the entire tree is not necessary, as the system can also rebuild just a subtree [Yang *et al.*, 2008].

Many articles on interactive ray tracing start from the assumption that all primitives in a scene can undergo totally random motion. While such an approach is laudable, primitives can be categorized into at least four groups: static, hierarchical movement, unstructured movement and PCM (short for piecewise coherent movement) [Günther *et al.*, 2006]. Static objects neither move nor deform and so their acceleration structure does not need to be rebuilt. Parker *et al.* [1999] combined static and dynamic objects by placing static objects in an acceleration structure and by leaving dynamic objects outside of the acceleration structure altogether, to be tested by each ray individually. Bikker [2007] built separate kd-trees for static and dynamic objects, but noted that tracing performance would degrade as each ray would have to traverse both trees. Unstructured motion is totally random and thus the acceleration structure of such objects needs rebuilding. In hierarchical motion all triangles that undergo the same transformation can be used to form a single object, as such primitives do not move relative to each other. Furthermore, instead of transforming the object and rebuilding its acceleration structure every frame, it is possible to inversely transform a ray and intersect the ray with an untransformed object. This way only the transformation matrix of a hierarchical object needs to be updated and the acceleration structure needs to be built only once. While this results in millions of matrix-vector multiplications per frame, such operations are relatively low cost – especially when utilizing SIMD instructions [Lext and Akenine-Möller, 2001; Wald *et al.*, 2003]. Static, unstructured and hierarchical objects can be combined efficiently by building a two-level kd-tree, where the top level kd-tree contains local kd-trees of objects. Local acceleration structures of static and hierarchical objects need not be rebuilt at all. Acceleration structures of objects subject to unstructured motion can be rebuilt independently. The top level kd-tree needs to be rebuilt when there is hierarchical or unstructured motion in the scene as it is invalidated. The rebuilding cost is, however, low, since it is determined by the number of objects – not triangles – in the scene [Wald *et al.*, 2003].

Like in hierarchical motion, a two level kd-tree and inverse transformation of rays can also be used for PCM. In PCM animation is assumed to be defined as deformations of a base mesh, i.e., animations consist of predetermined poses and the amount of poses is bounded. The connectivity of a base mesh of an object remains the same during all known deformations of the mesh and local motion is assumed to be coherent. In addition to local (residual) motion, triangles undergo common motion defined by affine transformations. Thus, motion is applying affine transformations plus residual motion to a rest pose. A rest pose is selected from all known poses and it minimizes residual motion over all clusters and time steps. Residual motion of vertices is bounded by a box known as a fuzzy box. A fuzzy box of a triangle is a union of the fuzzy boxes of the vertices of the triangle in question. A kd-tree is built over fuzzy boxes instead of the triangles. As in any kd-tree the extent of the boxes needs to be minimized because large and overlapping boxes would more likely cause intersection calculations. Residual motion can be minimized through motion decomposition by subtracting common motion from an animation. To subtract as much common motion as possible, a mesh is clustered into submeshes which undergo coherent deformations, using generalized Lloyd relaxation. Lloyd relaxation is an algorithm for sorting data points into groups [Du *et al.*, 1999; Lloyd, 1982; Günther *et al.*, 2006a; Günther *et al.*, 2006b].

Minimized residual motion is used as a cost function for clustering. In each iteration step triangles are assigned to a cluster in which their residual motion is the smallest. Clustering starts with a single cluster and new clusters are inserted until triangles no longer change their cluster or when the overall residual motion drops below a threshold. To stabilize the clustering procedure, each cluster has a seed triangle which represents common motion of the cluster. New clusters are formed by selecting as a seed the triangle with largest residual motion and combining it with its neighbours, so that a unique coordinate system is defined. Concurrently, in already existing clusters the triangle with the smallest residual motion is chosen as a seed triangle. Clustering time increases linearly in number of time steps and candidate rest poses and for the test scenes takes anything from 20 to 95 minutes. While building the fuzzy tree needs to be done only once in a preprocessing step, the clustering times are still quite long. When compared with an animation rendered using prebuilt static kd-trees, the static kd-tree is faster by a factor of 1.2 to 2.6. Still, test scenes were rendered at 5 to 15 FPS on a single 2.8 GHz core. This approach also has the problem of only being applicable to predetermined animations and it is thus unable to handle, for instance, animation produced by a character animation system known as euphoria [Günther *et al.*, 2006a; Günther *et al.*, 2006b; NaturalMotion, 2012].

The need to know the animation in advance can be overcome by using information from skeletal animation. In skeletal animation an object (even though the name would seem to imply that the object needs to have a skeletal structure, in reality skeletal animation can be used to animate pretty much anything) has a twofold structure. The animation is handled by a hierarchy of interconnected bones, which are covered by a mesh. The mesh – also known as a skin – depicts the actual look of the object. The rest pose of a skeletal animation model is the original mesh defined by the artist. Vertices are influenced by one or more bones. However, the amount of bones influencing a given vertex can be considered limited, as generally only neighboring bones influence vertices. Fuzzy boxes can be made to account for all possible motion of each bone. Doing so, however, produces large boxes and thus lower traversal performance. Smaller boxes can be achieved by restricting bone rotation relative to its parent and by applying joint limits. While this might seem to lead to a more realistic outcome – as no arbitrary rotations are possible [Günther *et al.,* 2006b] – it would also mean that animations such as breaking an arm at the elbow could not be done. Further restrictions on motion can be applied by considering only certain animations. As the size of a fuzzy box additionally depends on the cluster it belongs to, fuzzy boxes are considered as part of each bone to determine where their residual motion is minimized. Sampling time is roughly linear in the amount of triangles as before, but since the rest pose of a model is known, the sampling time is considerably reduced when compared to the previous approach (50.4s for a 271k model opposed to the tens of minutes for a ~5k model when using the previous approach) [Günther *et al.,* 2006a; Günther *et al.,* 2006b].

A gkDtree is similar in idea to two-level kd-trees – it is a group hierarchy. The hierarchy is constructed by recursively traversing a scene graph and groups of the hierarchy keep track of their level. Scene graph nodes with only a single child are merged with the child to avoid duplicate AABBs. A group has an AABB which includes all the AABBs of the primitives of the group. Each group can have a different data structure and be processed by a different algorithm. Static and dynamic groups are separated by using a flag. Static groups are built once and their transform matrices need not be updated. Dynamic groups, on the other hand, update their transform matrix each frame. Furthermore, static groups which are on level 1 or next to the root, form a two-level hierarchy instead of a multi-level one. Unlike kd-trees, variables of each group point to local data. Due to the multi-level hierarchy, construction of the tree is rapid, because group node boundaries act as split candidates. Primitive boundaries need to be used only in the leaf nodes. To effectively parallelize reconstruction of dynamic groups of a tree in a multi-level hierarchy, dependencies between nodes need to be removed. This is achieved by updating AABBs of dynamic primitives before rebuilding of local acceleration structures. The updating is done by assigning group nodes to threads level by level from the bottom up using round-robin. Acceleration structures of groups can then be rebuilt in parallel. While node primitive count would be a good way to load balance, it would be too time consuming on the upper levels of a tree. A simple load balancing scheme assigns groups to threads from the bottom up level by level. The scheme works because groups on the same level, while not equal, have similar computational loads. The parallelization approach obviously works only when there is a sufficient amount of groups for all threads. In case there are too few groups, for instance, in-place algorithm by Choi *et al.* [2010] can be used. When compared to a kd-tree gkDtree has an update performance of 1.1x to 166.4x and even when comparing to a binned kd-tree by Shevtsov *et al.* [2007a] it reaches 0.4x to 96.9x speedup (1.1x and 0.4x on a scene with a single dynamic group). With six threads parallelization speedup ranges from about 3.5 to about 5, when discounting the scene with only a single dynamic group. All of the reported numbers are results without the initial building time of gkDtree [Kang *et al.*, 2011].

5. Kd-tree traversal

The most obvious way of traversing a kd-tree – discounting the sequential algorithm due to its gross inefficiency – is the recursive ray traversal algorithm. The idea is simple: if both child nodes need traversing, then store the other in a stack and traverse the other. If an intersection is found in the first branch then the algorithm is done. On the other hand, if no intersection is found, then a new node is popped from the stack. The algorithm then continues by recursing until an intersection is found or the stack is empty. If the stack empties and no intersection has been found, then the ray has no intersections. The first traversed child node is selected based on the position of the ray origin with regards to the splitting plane, the "near" child is traversed first. If, at each node, only one child node needs to be traversed then recursion is done until a leaf is encountered [Havran, 2001].

Basic recursive traversal is not a robust algorithm, because it can generate an incorrect image in two cases. When the ray entry point is on the splitting plane and the ray continues to either side of the splitting plane both child nodes get the same value. Selecting one side results in the correct image while the other causes an incorrect one. However, if the ray stays on the splitting plane until the exit point, then selecting either side will produce the correct result. The problem cases can be corrected for by comparing both the entry and exit points of a ray to the node splitting plane. If the entry and exit point coordinates are less than or equal to the splitting plane coordinates then the left child is selected. The right child is selected in the remaining case [Havran, 2001].

Generally, when traversing a kd-tree, packets are populated with rays that have the same sign, which might lead to underuse of SIMD as packet size decreases due to packet splitting. In this case the traversal ends when a node that is occluded by previous node(s) is encountered. If the packet were to include incoherent rays, then packet splitting would be unnecessary and SIMD utilization would increase. In this case a completely occluded node would not terminate traversal, but simply lead to popping the next node until the stack is empty. This approach reduces traversal steps by 2x for eye rays and 1.5x for secondary rays by 9th bounce. Performance improvement for eye and shadow rays decreases with bounce depth from 2x to 0.9-1.2x [Reshetov, 2006].

By using frustum culling algorithm (see Section 3.3), there is no need to conduct an exhaustive search for intersection points from the root. A ray frustum acts as a proxy for any number of randomly arranged rays as long as the following two conditions are satisfied:

1. For any given axis-aligned plane, compute a rectangle inside this plane, which contains all possible ray/plane intersection points. This rectangle does not have to be tight.
2. All rays go in the same direction.

The frustum can then be used to traverse the tree looking for a common entry point deeper in the tree for all the rays it represents. The traditional intersection search is started from this common entry point. The goal of the entry point search is thus not to find an actual intersection, but leaves where there is a possibility of an intersection. The possibility of an intersection exists when a leaf or leaves with objects, is or are fully or partially overlapped by a proxy frustum. If a proxy frustum overlaps a single leaf without overlapping any splitting planes, then the leaf is the entry point. On the other hand, if multiple leaves are overlapped, then the entry point is the common ancestor of the overlapped leaves. An entry point search starts with a stack known as a "bifurcation stack". It holds nodes, and their corresponding AABBs, which can be entry points. A tree is traversed from the root using a frustum culling algorithm in a depth first manner. While traversing, all nodes and their corresponding AABBs, with both children to be traversed, are placed in the bifurcation stack. The traversal ends when the first potential intersection is encountered, after which no new entries are added to the bifurcation stack and the current node is marked as an entry point candidate. The algorithm continues by popping a new node from the bifurcation stack. The previously unexplored branch under each node is traversed as before. This is continued until the bifurcation stack is empty. If, during the subsequent traversals, a new leaf node with potential intersections is found, then the node from which this leaf node was reached is marked as the current entry point candidate. When the bifurcation stack is empty, the current entry point and the AABB associated with it are returned as the entry point for all rays in the frustum [Fowler *et al.,* 2009; Reshetov *et al.,* 2005]

As a frustum represents many rays, some of which may diverge from the others, finding truly deep entry points for the frustum might be impossible. In such situations deeper entry points can be found by increasing ray coherency by dividing the frustum. For primary rays an image is easily divided into tiles of equal size. Such a division is also a common way to parallelize tracing of an image on a multicore system. Splitting or not splitting a tile can be decided by using the following parameters [Reshetov *et al.*, 2005]:

1. Initial Tile Size (ITS)
2. Minimum Tile Size (MTS), which automatically triggers intersection point search
3. Split Factor (SF), which defines how many pieces to split a tile into.

Tiles that are larger than MTS are split automatically if the entry point is not a leaf. Otherwise – based on measurements on over 2500 models – varying values of ITS, MTS and SF affects performance by about 10%. As such it is possible to use a single set of parameters for all scenes and still reach roughly optimal results. Currently best known results have been reached with ITS = 128x128 pixels and direct division to 16 subtiles as needed. This approach is known as Multi-Level Ray Tracing Algorithm or MLRTA and it is able to increase performance by 3.25x for eye rays and 2.75x for eye and shadow rays [Reshetov *et al.*, 2005].

A further improvement to MLRTA is known as AEPSA (Advanced Entry Point Search Algorithm). It is based on observations on the candidate find and entry point selection phases. For preparing the candidates Reshetov [2007] observed that even though a frustum reaches a leaf with objects, the probability that the rays intersect an object is low. Therefore, freezing the bifurcation stack as soon as a leaf with objects is found is premature. Instead, the stack is frozen when a leaf with an object overlapping the frustum is found. The existence of an intersection is easily ascertained by determining whether or not all triangles of an object are on the outside of a plane formed by a frustum face. Such a test easily extends to SIMD usage for all four planes simultaneously. Whether or not triangle vertices are on the same side of a plane is determined by comparing the signs of their dot products [Fowler *et al.,* 2009].

In the entry point selection phase MLRTA must, for all nodes in the bifurcation stack, traverse from a node to an occupied leaf. Because candidate nodes are ordered by depth, nodes deeper in the tree are in subtrees of nodes on higher levels. Thus, all nodes below the current one in the candidate list can be culled, if both subtrees of the current node are determined to have leaves which are not overlapped by the frustum. This means that AEPSA will visit as many or fewer nodes when compared to MLRTA. These improvements produce a speedup of up to 18% [Fowler *et al.,* 2009]. Algorithm 6.1 presents AEPSA.

AEPSA algorithm

```
Algorithm aepsa(tree, frustum)
Input: tree to search for entry points in, frustum representing rays
Output: entry point for all rays
  queue
  find_candidates(root(tree), frustum, queue)
  while !empty(queue) do
    node = dequeue(queue)
    if traverse_to_leaf(frustum, node) along path to leaf not taken
       overlaps non-empty leaf then
        return node
    end if
  end while
  return null

Algorithm find_candidates(node, frustum, queue)
Input: root node of a tree, frustum representing rays, queue for candidate nodes
Output: current entry point candidate, queue with nodes to search
  if node is leaf then
    i = intersect(frustum, leaf);
    if i == TRUE then
      enqueue(stack, node);
      return current node, queue;
    end if
  end if
  s = find_candidates(left(node) OR find_candidates(right(node));
  if s == TRUE then
    enqueue(queue, node)
end if
```

Algorithm 5.1 AEPSA from Fowler *et al.* [2009]

Ray tracing is inherently a parallel process, as pixels are independent and the used data structures are read-only when the actual tracing is done. Near linear scalability has been observed, for example, by Parker *et al.* [1999] for 128 processors. The process is simple: all primary rays are placed in a queue, then individual threads lock the queue, pop a ray and unlock the queue. This is repeated until the queue is empty. Processing singular rays causes synchronization overhead, so rays are bundled into groups which in turn are placed in a queue. As a load balancing measure, the groups have decreasing size [Parker *et al.*, 1999]. A locking based approach is not the only way to implement access control to the shared queue. A lock-free approach, using for example the Compare-and-Swap atomic synchronization described in Algorithm 5.2, is also a possibility. A lock-free system should work better than a lock based system if contention is high. Another possible method is to distribute tasks to thread local queues, for instance using round robin [Nunes and Santos, 2009].

Compare-and-Swap algorithm

```
Algorithm CAS(location, cmpVal, newVal)
Input: memory location, compare value, new value
Output: whether location and compare value were the same
   if location == cmpVal then
      location = newVal
      return true
   end if
   return false
```

Algorithm 5.2 Compare-and-Swap from Nunes and Santos [2009]

6. Conclusions

This thesis has explored some aspects of ray tracing and research done to push these aspects towards real-time execution. Even though ray tracing as a whole is such a large field that covering everything is impossible, space limitations have made it so that covering even all integral parts – for instance ray-AABB intersection, see, e.g., Haines [1989], Williams *et al.* [2005] and Eisemann *et al.* [2007] for discussions on this topic – of a ray tracer in this thesis, has not been possible. Apart from the topics cut at the end of Chapter 1, shadow rays have been discussed on a basic level, but examining more advanced methods, such as volumetric occluders – see, for example, Djeu *et al.*, [2009] – and different methods for producing soft shadows – see, for instance, Johnson *et al.* [2009] and Laine *et al.* [2005] – has been omitted. Moreover, further discussion of data layouts, such as Yoon and Manocha [2006], was cut. Exploration of other triangle structures in addition to triangle fans, such as triangle strips and clusters – see, for instance, Lauterbach *et al.* [2007] and Garanzha [2009] – was also left undone. A similar decision was made regarding other ray-triangle intersection methods, such as the one presented by Segura and Feito [2001]. Compression, see, e.g., Hubo *et al.* [2006], is also left unexplored. Use of multiple different types of data structures for different purposes – see, for instance, Hunt [2008b] – is likewise unexplored.

Ray tracing has seen impressive efficiency increases since 2000, but it still has ways to go before it can challenge rasterization in real-time applications. While there are serious discussions about whether some features, or ray tracing as a whole, is desirable or even needed at all in interactive graphics, ray tracing is experiencing a boom of interest [PcPerspective, 2011; Stratton, 2013]. Real-time ray tracing is not just an academic exercise anymore, as Intel and nVIDIA have active research projects and have released demos showing their technology at work. First dedicated ray tracing hardware, though not for real-time execution, has also been released [Imagination, 2013]. But the most impressive proof of concept, that can run in real-time on hardware available to consumers today, is Brigade path tracer [Brigade, 2013]. Capability to render billions of triangles at 30 FPS at 720p resolution with global illumination is a sight to behold, even if it is a little noisy [Lapere, 2013]. With all the recent advancements and ongoing research in ray tracing theory, this is truly an intriguing time to be interested in ray tracing – whether or not real-time execution becomes a reality within the decade or century.

References

[Amdahl, 1967] Gene M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS ’67,* (1967), 483-485.

[Appel, 1968] Arthur Appel, Some techniques for shading machine renderings of solids. In: *AFIPS '68*, (1968), 37-45.

[Arvo and Kirk, 1989] James Arvo and David Kirk, A survey of ray tracing acceleration techniques. In: Andrew S. Glassner (ed.), *An Introduction to Ray Tracing*. Academic Press, 1989, 201-262.

[Badouel, 1990] Didier Badouel, An efficient ray-polygon intersection. In. Andrew S. Glassner (ed.), *Graphics Gems*. Academic Press, 1990, 390-393.

[Benthin, 2006] Carsten Benthin, Realtime ray tracing on current CPU architectures. Ph.D. thesis, Saarland University.

[Benthin and Wald, 2009] Carsten Benthin and Ingo Wald, Efficient ray traced soft shadows using multi-frusta tracing. In: *Proc. Conference on High Performance Graphics 2009*, (2009), 135-144.

[Bentley, 1975] Jon Louis Bentley, Multidimensional binary search trees used for associative searching. *Comm. ACM* **18** (1975), 509-517.

[Bikker, 2007] Jacco Bikker, Real-time ray tracing through the eyes of a game developer. In: *Proc. 2007 IEEE Symposium on Interactive Ray Tracing,* (2007), 1-10.

[Boulos *et al.*, 2007] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald, Packet-based whitted and distribution ray tracing. In: *Proc. Graphics Interface 2007,* (2007), 177-184.

[Boulos *et al.*, 2008] Solomon Boulos, Ingo Wald, and Carsten Benthin, Adaptive ray packet reordering. In: *Proc. 2008 IEEE Symposium on Interactive Ray Tracing,* (2008), 131-138.

[Boulos *et al.*, 2006] Solomon Boulos, Ingo Wald, and Peter Shirley, Geometric and arithmetic culling methods for entire ray packets. Technical Report No UUCS-06-10, University of Utah.

[Brigade, 2013] Brigade path tracer, http://igad.nhtv.nl/~bikker/. 2013. Checked 5.5.2013.

[Cadet and Lécussan, 2007] Gilles Cadet and Bernard Lécussan, Coupled use of BSP and BVH trees in order to exploit ray bundle performance. In: *Proc. 2007 IEEE Symposium on Interactive Ray Tracing*, (2007), 63-71.

[Catmull, 1974] Edwin Catmull, A subdivision algorithm for computer display of curved surfaces. Ph.D. thesis, University of Utah.

[Choi *et al*., 2010] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart, Parallel SAH k-d tree construction. In: *Proc. Conf. High Performance Graphics 2010*, (2010), 77-86.

[Cook, 1989] Robert L. Cook, Stochastic sampling and distributed ray tracing. In: Andrew S. Glassner (ed.), *An Introduction to Ray Tracing*. Academic Press, 1989, 161-199.

[Cook *et al*., 1984] Robert L. Cook, Thomas Porter, Loren Carpenter, Distributed ray tracing. *Computer Graphics* **18** (1984), 137-145.

[Djeu *et al.*, 2009] Peter Djeu, Sean Keely, and Warren Hunt, Accelerating shadow rays using volumetric occluders and modified kd-tree traversal. In: *Proc. Conf. High Performance Graphics 2009,* (2009), 69-76.

[Dmitriev *et al.*, 2004] Kirill Dmitriev, Vlastimil Havran, and Hans-Peter Seidel, Faster ray tracing with SIMD shaft culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut Für Informatik.

[Du *et al.*, 1999] Qiang Du, Vance Faber, and Max Gunzburger, Centroidal voronoi tessellations: Applications and algorithms. *SIAM Review* **44** (1999), 637-676.

[Eisemann *et al.*, 2007] Martin Eisemann, Thorsten Grosch, Stefan Müller, and Marcus Magnor, Fast ray/axis-aligned bounding box overlap tests using ray slopes. *J. Graphics Tools* **12** (2007), 35-46.

[Erickson, 1997] Jeff Erickson, Plücker coordinates. http://tog.acm.org/resources/RTNews/html/rtnv10n3.html#art11. 1997. Checked 18.3.2013.

[Foley *et al*., 1990] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice, 2nd ed.* Addison-Wesley, 1990.

[Fowler *et al.*, 2009] Colin Fowler, Steven Collins, and Michael Manzke, Accelerated entry point search algorithm for real-time ray-tracing. In: *SCCG '09*, (2009), 59-65.

[Fussell and Subramanian, 1988] Donald Fussell and K. R. Subramanian, Fast ray tracing using k-d trees. Technical report, Tx78712-1188, The University of Texas at Austin, 1988.

[Galin and Akkouche, 2005] Eric Galin and Samir Akkouche, Fast processing of triangle meshes using triangle fans. In: *2005 International Conference on Shape Modeling and Applications* (2005), 326-331.

[Garanzha, 2009] Kirill Garanzha, The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. *Computer Graphics Forum* **28** (2009), 1199-1206.

[Georgia Tech, 2013] Georgia Institute of Technology, Large geometric models archive. http://www.cc.gatech.edu/projects/large_models/. 2013. Checked 6.4.2013.

[Glassner, 1989a] Andrew S. Glassner, An overview of ray tracing. In: Andrew S. Glassner (ed.), *An Introduction to Ray Tracing*. Academic Press, 1989, 1-31.

[Glassner, 1989b] Andrew S. Glassner, Surface physics for ray tracing. In: Andrew S. Glassner (ed.), *An Introduction to Ray Tracing*. Academic Press, 1989, 121-160.

[Goldsmith and Salmon, 1987] Jeffrey Goldsmith and John Salmon, Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* **7** (1987), 14-20.

[Gribble and Ramani, 2008] Christiaan P. Gribble and Karthik Ramani, Coherent ray tracing via stream filtering. In: *Proc. 2008 IEEE Symposium on Interactive Ray Tracing*, (2008), 59-66.

[Günther *et al.*, 2006b] Johannes Günther, Heiko Friedrich, Hans-Peter Seidel, and Philipp Slusallek, Interactive ray tracing of skinned animations. *The Visual Computer* **22** (2006), 785-792.

[Günther *et al.*, 2006a] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek, Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum* **25** (2006), 517-525.

[Haines, 1989] Eric Haines, Essential ray tracing algorithms. In: Andrew S. Glassner (ed.), *An Introduction to Ray Tracing*. Academic Press, 1989, 33-77.

[Havel and Herout, 2010] Jirí Havel and Adam Herout, Yet faster ray-triangle intersection (using sse4). In: *IEEE Trans. on Visualization and Computer Graphics*, (2010), 434-438.

[Havran, 2001] Vlastimil Havran, Heuristic ray shooting algorithms. Ph.D. thesis, Czech Technical University.

[Havran and Bittner, 2002] Vlastimil Havran and Jirí Bittner, On improving kd-trees for ray shooting. In: *Proc. WSCG 2002*, (2002), 209-216.

[Hubo *et al.*, 2006] Erik Hubo, Tom Mertens, Tom Haber, and Philippe Bekaert, The quantized kd-tree: Efficient ray tracing of compressed point clouds. In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, (2006), 105-113.

[Hunt, 2008a] Warren Hunt, Corrections to the Surface Area Metric with Respect to Mail-Boxing. In: *Proc. 2008 IEEE Symposium on Interactive Ray Tracing*, (2008), 77-80.

[Hunt, 2008b] Warren Hunt, Data structures and algorithms for real-time ray tracing at the University of Texas at Austin. Ph.D. thesis, The University of Texas at Austin.

[Hunt *et al*., 2007] Warren Hunt, William R. Mark, and Don Fussell, Fast and lazy build of acceleration structures from scene hierarchies. In: *Proc. 2007 IEEE Symposium on Interactive Ray Tracing*, (2007), 47-54.

[Hunt *et al*., 2006] Warren Hunt, William R. Mark, and Gordon Stoll, Fast kd-tree construction with an adaptive error-bounded heuristic. In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, (2006), 81-88.

[Hurley *et al.*, 2002] Jim Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov, Fast ray tracing for modern general purpose CPU. In: *Proc. 2002 International Conference GraphiCon*, (2002).

[Imagination, 2013] Imagination, Caustic. https://www.caustic.com/. 2013. Checked 5.5.2013.

[Johnson *et al.,* 2009] Gregory S. Johnson, Allen Hux, Christopher A. Burns, Warren A. Hunt, William R. Mark, and Stephen Junkins, Soft irregular shadow mapping: Fast, high-quality, and robust soft shadows. In: *Proc. 2009 Symposium on Interactive 3D Graphics and Games*, (2009), 57-66.

[Kang *et al.*, 2011] Yoon-Sig Kang, Jae-Ho Nah, Woo-Chan Park, and Sung-Bong Yang, gkDtree: A group-based parallel update kd-tree for interactive ray tracing. *J. Systems Architecture* **59** (2013), 166-175.

[Kensler and Shirley, 2006] Andrew Kensler and Peter Shirley, Optimizing ray-triangle instersection via automated search. In: *Proc. IEEE Symposium on Interactive Ray Tracing 2006*, (2006), 33-38.

[Komatsu *et al.,* 2008] Kazuhiko Komatsu, Yoshiyuki Kaeriyama, Kenichi Suzuki, Hiroyuki Takizawa, and Hiroaki Kobayashi, A fast ray frustum-triangle intersection algorithm with precomputation and early termination. *IPSJ Online Transactions* **1** (2008), 1-11.

[Laine *et al.*, 2005] Samuli Laine, Timo Aila, and Ulf Assarsson, Soft shadow volumes for ray tracing. In: *SIGGRAPH '05,* (2005), 1156-1165.

[Lapere, 2013] Samuel Lapere, Ray tracey's blog. http://raytracey.blogspot.co.nz/. 2013. Checked 5.5.2013.

[Lauterbach *et al*., 2007] Christian Lauterbach, Sung-Eui Yoon, and Dinesh Manocha, Ray-strips: A compact mesh representation for interactive ray tracing. In: *Proc. IEEE Symposium on Interactive Ray Tracing 2007*, (2007), 19-26.

[Lauterbach *et al*., 2006] Christian Lauterbach, Sung-Eui Yoon, and David Tuft, RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In: *Proc. IEEE Symposium on Interactive Ray Tracing 2006*, (2006), 39-46.

[Lext and Akenine-Möller, 2001] Jonas Lext and Tomas Akenine-Möller, Towards rapid reconstruction for animated ray tracing. In: *Eurographics 2001 – Short Presentations,* (2001).

[Lloyd, 1982] Stuart P. Lloyd, Least squares quantization in PCM. *IEEE Trans. Information Theory* **28** (1982), 129-137.

[MacDonald and Booth, 1990] J. David MacDonald and Kellogg S. Booth, Heuristics for ray tracing using space subdivision. *The Visual Computer* **6** (1990), 153-166.

[Mahovsky and Wyvill, 2006] J. Mahovsky and B. Wyvill, Memory-conserving bounding volume hierarchies with coherent ray tracing. *Computer Graphics Forum* **25** (2006), 173-182.

[Månsson *et al.*, 2007] Erik Månsson, Jacob Munkberg, and Tomas Akenine-Möller, Deep coherent ray tracing. In: *Proc. 2007 IEEE Symposium on Interactive Ray Tracing*, (2007), 79-85.

[Möller and Trumbore, 1997] Tomas Möller and Ben Trumbore, Fast, minimum storage ray/triangle intersection. *J. Graphics Tools* **2** (1997), 21-28.

[Moon *et al.*, 2010] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon, Cache-oblivious ray reordering. *ACM Trans. Graphics* **29** (2010), article 28.

[NaturalMotion, 2012] NaturalMotion, euphoria. http://www.naturalmotion.com/products/euphoria/. 2012. Checked 15.9.2012.

[Navrátil, 2010] Paul Arthur Navrátil, Memory-efficient, scalable ray tracing. Ph.D. thesis, The University of Texas at Austin.

[Navrátil *et al*., 2007] Paul Arthur Navrátil, Donald S. Fussell, Calvin Lin, and William R. Mark, Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In: *Proc. 2007 IEEE Symposium on Interactive Ray Tracing*, (2007), 95-104.

[Nunes and Santos, 2009] Miguel Nunes and Luís Paulo Santos, Workload distribution for ray tracing in multi-core systems. In: *Proc. 17º Encontro Português de Computação Gráfica*, (2009).

[Overbeck *et al*., 2008] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark, Large ray packets for real-time whitted ray tracing. In: *Proc. 2008 IEEE Symposium on Interactive Ray Tracing*, (2008), 41-48.

[Parker *et al.*, 1999] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen, Interactive ray tracing. In: *Proc. I3D '99,* (1999), 119-126.

[PcPerspective, 2011] PcPerspective, John Carmack interview: GPU race, Intel graphics, ray tracing, voxels and more!. http://www.pcper.com/reviews/Editorial/John-Carmack-Interview-GPU-Race-Intel-Graphics-Ray-Tracing-Voxels-and-more. 2011. Checked 7.5.2013.

[Peachey, 1990] Darwyn Peachey, Texture on demand. Technical Memo #217, Pixar Animation Studios.

[Pharr *et al*., 1997] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan, Rendering complex scenes with memory-coherent ray tracing. In: *SIGGRAPH '97*, (1997), 101-108.

[Popov *et al*., 2006] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek, Experiences with streaming construction of SAH kd-trees. In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, (2006), 89-94.

[Ramani *et al*., 2009] Karthik Ramani, Christiaan P. Gribble, and Al Davis, StreamRay: A stream filtering architecture for coherent ray tracing. In: *ASPLOS '09*, (2009), 325-336.

[Reshetov, 2006] Alexander Reshetov, Omnidirectional ray tracing traversal algorithm for kd trees. In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, (2006), 57-60.

[Reshetov, 2007] Alexander Reshetov, Faster ray packets - triangle intersection through vertex culling. In: *Proc. 2007 IEEE Symposium on Interactive Ray Tracing*, (2007), 105-112.

[Reshetov *et al.*, 2005] Alexander Reshetov, Alexei Soupikov, and Jim Hurley, Multi-level ray tracing algorithm. In: *SIGGRAPH '05*, (2005), 1176-1185.

[Rusinkiewicz, 1997] Szymon Rusinkiewicz, A survey of BRDF representation for computer graphics. http://www.cs.princeton.edu/~smr/cs348c-97/. 1997. Checked 6.4.2013.

[Schlick, 1994] Christophe Schlick, An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum* **13** (1994), 233-246.

[Segura and Feito, 2001] Rafael J. Segura and Francisco R. Feito, Algorithms to test ray-triangle intersection. Comparative study. In: *WSCG (Short Papers) 2001*, 2001, 76-81.

[Shevtsov *et al.*, 2007a] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin, Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum* **26** (2007), 395-404.

[Shevtsov *et al.*, 2007b] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin, Ray-triangle intersection algorithm for modern CPU architectures. In: *Proc. 2007 International Conference GraphiCon*, (2007).

[Shirley et al., 2009] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen R. Marschner, Erik Reinhard, Kelvin Sung, William B. Thompson and Peter Willemsen, *Fundamentals of Computer Graphics*. A K Peters, 2009.

[Shirley *et al.*, 2008] Peter Shirley, Kelvin Sung, Erik Brunvand, Alan Davis, Steven Parker, and Solomon Boulos, Fast ray tracing and the potential effects on graphics and gaming courses. *Computers & Graphics* **32** (2008), 260-267.

[Shoemake, 1998] Ken Shoemake, Plücker coordinate tutorial. http://tog.acm.org/resources/RTNews/html/rtnv11n1.html#art3. 1998. Checked 18.3.2013.

[Smits, 1998] Brian Smits, Efficiency issues for ray tracing. *J. Graphics Tools* **3** (1998), 1-14.

[Soupikov *et al.*, 2008] Alexei Soupikov, Maxim Shevtsov, and Alexander Kapustin, Improving kd-tree quality at a reasonable construction cost. In: *Proc. 2008 IEEE Symposium on Interactive Ray Tracing*, (2008), 67-72.

[Stanford, 2013] Stanford University, The Stanford 3D scanning repository. http://graphics.stanford.edu/data/3Dscanrep/. 2013. Checked 6.4.2013.

[Stratton, 2013] Josh Stratton, State of ray tracing (in games). http://www.cs.utah.edu/~jstratto/state_of_ray_tracing/. 2013. Checked 7.5.2013.

[Utah, 2013] The University of Utah, The Utah 3D animation repository. http://www.sci.utah.edu/~wald/animrep/. 2013. Checked 6.4.2013.

[Wächter, 2004] Carsten Wächter, Realtime ray tracing. Ph.D. thesis, Universität Ulm.

[Wald, 2004] Ingo Wald, Realtime ray tracing and interactive global illumination. Ph.D. thesis, Saarland University.

[Wald, 2007] Ingo Wald, On fast construction of SAH-based bounding volume hierarchies. In: *Proc. 2007 IEEE Symposium on Interactive Ray Tracing*, (2007), 33-40.

[Wald *et al.*, 2003] Ingo Wald, Carsten Benthin, and Philipp Slusallek, Distributed interactive ray tracing of dynamic scenes. In: *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003,* (2003), 77-85.

[Wald *et al.*, 2007b] Ingo Wald, Solomon Boulos, and Peter Shirley, Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graphics* **26** (2007), article 6.

[Wald *et al.*, 2007a] Ingo Wald, Christiaan Gribble, Solomon Boulos, and Andrew Kensler, SIMD ray stream tracing – SIMD ray traversal with generalized ray packets and on-the-fly re-ordering. Technical report #UUSCI-2007-012, University of Utah.

[Wald and Havran, 2006] Ingo Wald and Vlastimil Havran, On building fast kd-trees for ray tracing, and on doing that in O(N log N). In: *Proc. 2006 IEEE Symposium on Interactive Ray Tracing*, (2006), 61-69.

[Wald *et al*. 2006] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker, Ray tracing animated scenes using coherent grid traversal. In: *SIGGRAPH '06*, (2006), 485-493.

[Wald *et al*., 2001] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner, Interactive rendering with coherent ray tracing. *Computer Graphics Forum* **20** (2001), 153-165.

[Williams *et al.*, 2005] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley, An efficient and robust ray-box intersection algorithm. In: *SIGGRAPH '05,* (2005), Article No. 9.

[Whitted, 1980] Turner Whitted, An improved illumination model for shaded display. *C. ACM* **23** (1980), 343-349.

[Yang *et al.*, 2008] Xin Yang, Duan-qing Xu, and Lei Zhao, Ray tracing dynamic scenes using fast kd-tree base on multi-core architecture. In: *2008 IEEE Inter. Conf. on Comp. Sci. and Software Eng.*, (2008), 1120-1123.

[Yoon and Manocha, 2006] Sung-Eui Yoon and Dinesh Manocha, Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum* **25** (2006), 507-516.

Appendix A

The Angel model, 474048 triangles [Georgia Tech, 2013].

The Bunny model, 69451 triangles [Stanford, 2013].

The Fairy Forest model, 174117 triangles [Utah, 2013].