Fast *kd*-Tree Construction for 3D-Rendering Algorithms Like Ray Tracing

Sajid Hussain and Håkan Grahn

Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
{sajid.hussain,hakan.grahn}@bth.se
http://www.bth.se/tek/paarts

Abstract. Many computer graphics rendering algorithms and techniques use ray tracing for generation of natural and photo-realistic images. The efficiency of the ray tracing algorithms depends, among other techniques, upon the data structures used in the background. *kd*-trees are some of the most commonly used data structures for accelerating ray tracing algorithms. Data structures using cost optimization techniques based upon Surface Area Heuristics (SAH) are generally considered to be best and of high quality. During the last decade, the trend has been moved from off-line rendering towards real time rendering with the introduction of high speed computers and dedicated Graphical Processing Units (GPUs). In this situation, SAH-optimized structures have been considered too slow to allow real-time rendering of complex scenes. Our goal is to demonstrate an accelerated approach in building SAH-based data structures to be used in real time rendering algorithms. The quality of SAH-based data structures heavily depends upon split-plane locations and the major bottleneck of SAH techniques is the time consumed to find those optimum split locations. We present a parabolic interpolation technique combined with a golden section search criteria for predicting *kd*-tree split plane locations. The resulted structure is 30% faster with 6% quality degradation as compared to a standard SAH approach for reasonably complex scenes with around 170k polygons.

1 Introduction

Almost everyone in the field of 3D computer graphics is familiar with ray tracing - a very popular rendering method for generating and synthesizing photo realistic images. The simplicity of the algorithm makes it very attractive. However, it has very high computational demands and a lot of research has been done for the last couple of decades in order to increase the performance of ray tracing algorithms. Different types of acceleration techniques have been proposed like fast ray - object intersections, Bounding Volume Hierarchies (BVH), Octrees, *kd*-trees, and different flavors of grids including uniform, non-uniform, recursive, and hierarchical [5], [13], [15]. *kd*-trees, due to their versatility and wide range of application areas, are one of the most used techniques to generate efficient data structures for fast ray tracing and are increasingly being adopted by researchers around the world. Wald [14] and Havran [5] report that *kd*-trees are good adaptive techniques to deal with varying complexity of the scene and usually perform better, or at least comparable, to any other similar technique. *kd*-tree construction normally uses a fixed stop criteria, either depth of the tree or the number of objects in the leaf node. Since adaptation is the main property for the *kd*-tree to behave better as compared to other similar techniques, the stop criteria should also be adaptive for the best generation of *kd*-trees.

In this paper, we present an approach to improve and optimize the construction of *kd*-trees. We are concerned about the decision of the separation plane location. Our approach is to use parabolic interpolation combined with golden section search to reduce the amount of work done when building the *kd*-trees. The SAH cost function used to chose the best split location is sampled at three locations which brackets the minimum of the cost function. These three locations are determined by a golden section search. A parabolic interpolation is then used to estimate the minimum and the algorithm iteratively converges towards optimum. A minimum is then found for the optimum locations of split planes (see section 4).

We have evaluated our proposed approximation as compared to a standard SAH algorithm in two ways. First, using Matlab models, we have evaluated how well it predicts the actual split plane locations when building the *kd*-trees. Second, we have implemented our model in a real ray tracer and have used five common scenes with varying number of geometrical complexities up to 170k polygons. As compared to a standard SAH based *kd*-tree data structures, our approach is considerable faster and justified. The demonstrated improvement ranges from 7% to 30% as the scene becomes more and more complex. On the other hand, the tree quality decreases.

The rest of the paper is organized as follows. In section 2, we discuss some previous work on the topic under investigation. Section 3 presents the basics of the *kd*-tree algorithm and the cost function to be minimized followed by the theory behind parabolic interpolation and golden section search criteria in section 4. Section 5 implements the idea followed by conclusion and future work in section 6.

2 Related Work

kd-tree construction has mainly focused on optimized data structure generation for fast ray tracing. Wald [14] and Harvan [5] analyzed the *kd*-tree algorithm in depth and proposed a state-of-the-art $O(n\log n)$ algorithm. Chang [3], in his thesis work described the theoretical and practical aspects of ray tracing including *kd*-tree cost function modeling and experimental verifications. Current work by Hunt [7] and Harvan [6] also aims at fast construction of *kd*-trees. By adaptive sub-sampling they approximate the SAH cost function by a piecewise quadratic function. There are many different other implementations of the kd-tree algorithm using SIMD instructions like Benthin [2]. Another approach is used by Popov [10], where he experiments with stream *kd*-tree construction and explores the benefits of parallelized streaming. Both Hunt [7] and Popov [10] demonstrate considerable improvements as compared to conventional SAH based *kd*-tree construction.

The cost function to optimally determine the depth of the subdivision in *kd*-tree construction has been demonstrated by several authors. Cleary and Wyvill [16] derive an expression that confirms that the time complexity is less dependent on the number of objects and more on the size of the objects. They calculate the probability that the ray intersects an object as a function of the total area of the subdivision cells that (partly) contain the object. MacDonald and Booth [8] use a similar strategy but refine the method to avoid double intersection tests of the same ray with the same object. They determine the probability that a ray intersects at least one leaf cell from the set of leaves within which a particular object resides. They use a cost function to find the optimal cutting planes for a *kd*- tree construction. A similar method was also implemented by Whang [17]. *kd*-tree acceleration structures for modern graphics hardware have been proposed by Daniel [20] and Tim [21], where they experimented kd-tree acceleration structure for GPU raytracers and achieved considerable improvement.

3 Basics of *kd*-Tree Construction

In this section, we give some background about the *kd*-tree algorithm, which will be the foundation for the rest of the paper. Consider a set of points in a space R^d, the *kd*-tree is normally built over these points. In general, *kd*-trees are used as a starting point for optimized initialization of *k*-means clustering [11] and nearest neighbor query problems [12]. In computer graphics, and especially in ray tracing applications, *kd*-trees are applied over a scene *S* with points as bounding boxes of scene objects.

The *kd*-tree algorithm subdivides the scene space recursively. For any given leaf node L_{node} of the *kd*-tree, a splitting plane splits the bounding box of the node into two halves, resulting in two bounding boxes, left and right. These are called child nodes and the process is repeated until a certain criterion is met. Havran [5] reports that the adaptability of the *kd*-tree towards the scene complexity can be influenced by choosing the best position of the splitting plane.

The choice of the splitting plane is normally the mid way between the scene maximum and minimum along a particular coordinate axis [9] and a particular cost function is minimized. MacDonald and Booth [8] introduced SAH for the *kd*-tree construction algorithm which works on probabilities and minimizes a certain cost function. The cost function is built by firing an arbitrary ray through the *kd*-tree and applying some assumptions. Fig. 1 uses the conditional probability *P(y/x)* that an arbitrary fired ray hits the region *y* inside region x provided that it has already touched the region *x*. Bayes rule can be used to calculate the conditional probability *P(y/x)* as

$$P(y/x) = \frac{P(x/y)P(y)}{P(x)} . \quad (1)$$

P(x/y) is the conditional probability that the ray hits the region *x* provided that it has intersected *y*, and here *P(x/y)* = 1. *P(x)* and *P(y)* can be expressed in terms of areas [5].

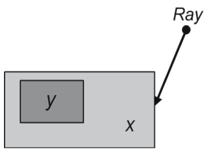

Fig. 1. Visualization of the conditional probability *P(y/x)* that a ray intersects region *y* given that it has intersected region *x*

In Fig. 2, if we start from the root node or the parent node and assume that it contains N elements and the ray passing the root node has to be tested for intersection with N elements. If we assume that the computational time it takes to test the ray intersection with element $n \subseteq N$ is T_n, then the overall computational cost C of the root node would be

$$C = \sum_{n=1}^{N} T_n \ . \tag{2}$$

After further division of root node (Fig. 2), the ray intersection test cost for each left and right child nodes changes to C_{Left} and C_{Right}. Thus the overall new cost becomes C_{Total} and

$$C_{Total} = C_{Trans} + C_{Right} + C_{Left} \ . \tag{3}$$

Where, C_{Trans} is the cost of traversing the parent or root node. The equation can be written as

$$C_{Total} = C_{Trans} + P_{Left} \cdot \sum_{i=1}^{N_{Left}} T_i + P_{Right} \cdot \sum_{j=1}^{N_{Right}} T_j \ . \tag{4}$$

Where,

$$P_{Left} = \frac{A_{Left}}{A} \ , \tag{5}$$

and

$$P_{Right} = \frac{A_{Right}}{A} \ . \tag{6}$$

Where A is the surface area of the root node and the area of two child nodes are A_{Left} and A_{Right}. P_{Left} and P_{Right} are the probabilities of a ray hitting the left and the right child nodes. N_{Left} and N_{Right} are the number of objects present in the two nodes and T_i and T_j are the computational time for testing ray intersection with the i^{th} and j^{th} objects of the two child nodes. The *kd*-tree algorithm minimizes the cost function C_{total}, and then subdivides the child nodes recursively.

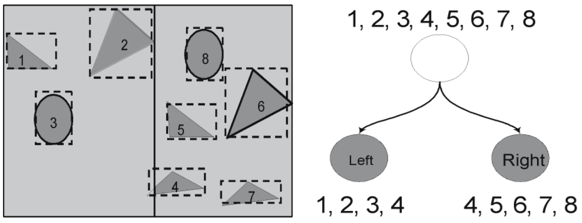

Fig. 2. Example scene division and the corresponding nodes in the *kd*-tree

As shown in [10], the cost function is a bounded variation function as it is the difference of two monotonically varying functions C_{Left} and C_{Right}. In [10], this important property of the cost function has been exploited to increase the approximation accuracy of cost function and only those regions that can contain the minimum have been adaptively sampled. We have used the golden section search to find out the region that could contain the minimum and combined it with parabolic interpolation to search for that minimum. In next section, we present the mathematical foundations of the technique and simulations in Matlab.

4 Parabolic Interpolation and Golden Section Search

The technique takes advantage of the fact that a second order polynomial usually provides a good approximation to the shape of a parabolic function. As the cost function we are dealing with is parabolic in nature, parabolic interpolation can provide good approximation of the cost function minima and hence the split plane locations. The idea behind is

$$f(x) = f(x_1)\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + f(x_2)\frac{(x-x_3)(x-x_1)}{(x_2-x_3)(x_2-x_1)} + f(x_3)\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \,. \quad (7)$$

Where, x_1, x_2 and x_3 are the three values of x for which the right hand side of the equation (7) is equal to its left hand side *f(x)*. Since a parabola is uniquely defined by a set of three points and this is the parabola that we want and we want the minimum, we differentiate equation (7) and put it equal to zero. After some algebraic manipulations we get the minimum of the parabola through these three points as

$$x_4 = x_2 - \frac{1}{2}\frac{(x_2-x_1)^2[f(x_2)-f(x_3)]-(x_2-x_3)^2[f(x_2)-f(x_1)]}{(x_2-x_1)\,[f(x_2)-f(x_3)]-(x_2-x_3)\,[f(x_2)-f(x_1)]} \,. \quad (8)$$

Where x_4 is the point where the minimum of the parabola occurs and $f(x_4)$ is the minimum of the parabola. The golden section search is similar in spirit to the bisection approach for locating roots of a function. We use golden ratio to find two intermediate sampling points as the starting points for optimization search.

$$\begin{aligned} x_1 &= x_L + h \\ x_2 &= x_U - h \end{aligned} \,. \quad (9)$$

Where x_1, x_2 are two intermediate points and x_L, x_U are lower and upper bounds of the cost function which contains the minimum. h can be calculated as

$$h = (\varphi - 1)(x_U - x_L) \,. \quad (10)$$

Where φ is called the golden ratio and $\varphi \approx 1.618$. The cost function is evaluated at points x_1 and x_2 and $f(x_1)$ is compared with $f(x_2)$. If $f(x_1) < f(x_2)$, then x_2, x_1 and x_U are used in equation (8) to find the minimum of the parabola.

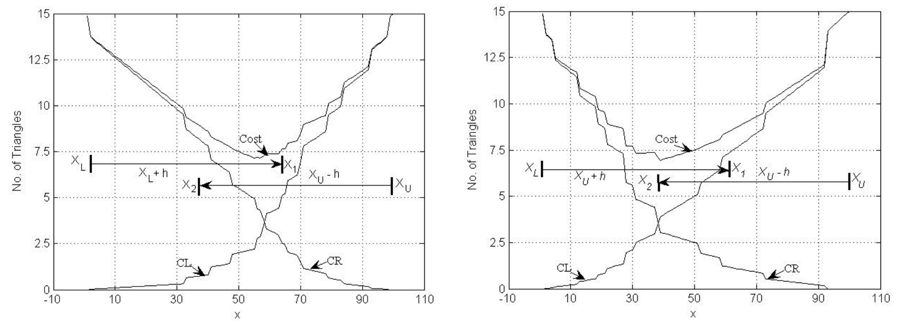

Fig. 3. Calculation of initial guess for parabolic interpolation

If $f(x_2) < f(x_1)$, then x_L, x_2 and x_1 are used in equation (8) to find the minimum of the parabola. This will give us a good initial guess to start our optimization through parabolic interpolation. With a good initial guess, the result converges more rapidly on the true value. Fig. 3 depicts the whole situation in graphical format. Two cost functions in Fig. 3 have been generated using Matlab and care has been taken that one of the cost functions contains the minimum in the right half of the x-axis and the other one contains the minimum in the left half of the x-axis. This is done to simulate the golden section search direction information. After x_4 has been calculated, $f(x_4)$ is evaluated and compared with the intermediate point which is x_1 in x_2, x_1 and x_U case and x_2 in x_L, x_2 and x_1 case. If *$f(x_4)$ < Intermediate Point (either $f(x_1)$ or $f(x_2)$)*, then we shift the parabola towards the right direction in the next iteration. In this case the upper bound remains the same, while the lower bound is shifted toward right with a constant step size called μ and the intermediate point (either x_1 or x_2) becomes x_4. If *$f(x_4)$ > Intermediate Point (either $f(x_1)$ or $f(x_2)$)*, then we shift the parabola towards the left direction in the next iteration. In this case the lower bound remains the same and we shift the upper bound towards left with the same step μ. The choice of the step size varies and the size decreases with increased tree depth as the number of primitives decreases and the size of the division axis also decreases. Consider a real case depicted in Fig. 4

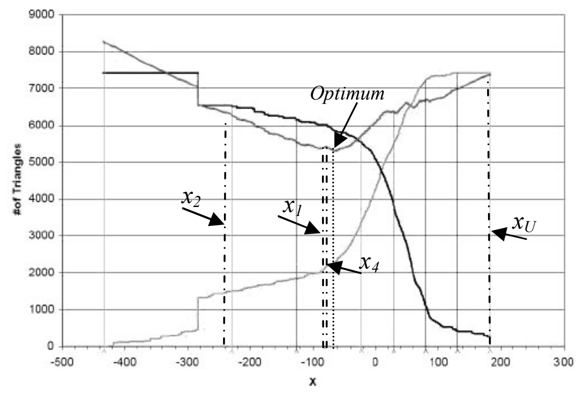

Fig. 4. Optimum search with parabolic interpolation

and taken from [7]. After, golden section search, x_1 and x_2 are found and it is clear from Fig. 4 that $f(x_1) < f(x_2)$. We chose x_2, x_1 and x_U for interpolation and from equation (8) we find x_4. Since $f(x_4) < f(x_2)$, means we have to move towards right. For next iteration, x_1 becomes x_4, x_U remains unchanged and x_2 increases to $x_2 + \mu$. Using $\mu = 10$, we reach to the optimized solution as indicated by a dotted line in Fig. 4. Note that for each iteration other than first one, we need to sample the cost function only at one location i.e., in the second iteration, we only have to sample the cost function at $x_2 + \mu$. In this case we only need to sample the cost function at five different locations x_1, x_2, x_4, $x_2 + \mu$ and x_U.

5 Experimental Evaluation

To evaluate the performance of our algorithm, we have compared it with the conventional SAH construction techniques and measured the time needed for building the *kd*-tree structure for different test scenes. We have also tried to measure the cost overhead introduced by the approximation errors. As [7] uses SIMD support (SSE) and proposes to sample the cost function at more than one location during a single scan. Our algorithm exhibits the property to sample the cost function at four different locations during a single scan. The locations are x_1, x_2, $x_2 + \mu$ and x_U. Hence, it provides the base for SIMD support. However, we have not used this strategy in our implementations in this version.

We implemented our algorithm in C++ and tested on a variety of scenes (Fig. 5). The BUNNY model is used from the Stanford 3D Scanning Repository [19]. This image is acquired from a 3D scanner and hence contains regularly distributed triangles. On the other hand, Square Flake, F15 and HAND are designed with CAD tools and the regularity of primitive distribution varies. As, the algorithm described above behaves well if the cost function is slowly varying. The BUNNY model is more suitable scene for our algorithm. All the measurements were performed on a workstation with an Intel Core 2 CPU 2.16GHz processor and 2GB of RAM.

We see from Table 1 that the cost increase for BUNNY is approximate 1.62% despite of 26% increase in speed. The reason is that our algorithm performs well if the primitives are more towards normal distribution. As, this scene was acquired using a 3D scanner and the primitives are uniformly distributed, our technique works well in this case. On the other hand, take the example of spheres. Although the complexity is less as compared to that of BUNNY, but the increase in cost is 2.5% because of the fact that primitives are not uniformly distributed in this case. The Fairy scene contains more empty spaces and the cost increase jumps to 5.72%. The main reason behind is that the algorithm performs poor once there is a step change or abrupt change in the cost function as discussed in [7]. Its quality degrades because parabolic interpolation exhibits swinging characteristics while interpolating step changes.

To overcome this problem, we can use splines piecewise interpolation to minimize oscillations by interpolating lower order polynomials, in this case linear spline piecewise interpolations.

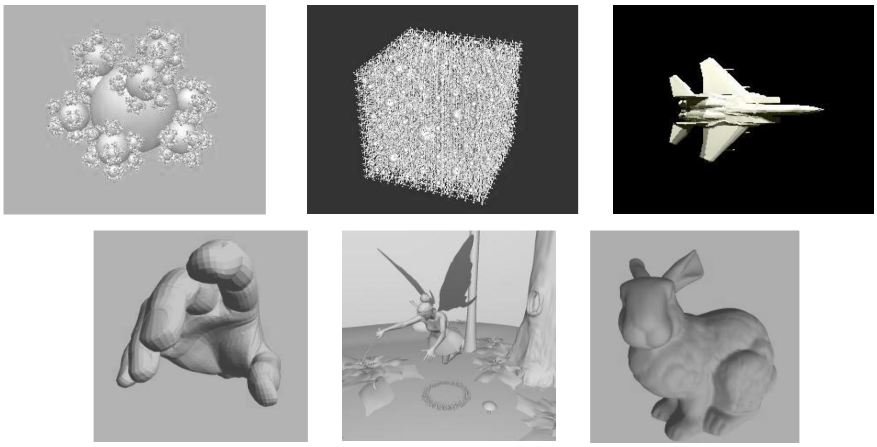

Fig. 5. Test scenes : Spheres, Mesh, F15, Hand, Fairy and Bunny

Table 1. Comparison of conventional *kd*-tree SAH and modified *kd*-tree SAH. The comparison is made in built time and quality of kd-tree.

Scene	**Primitives**	**Conventional SAH**		**Modified SAH**		**Speedup**	**Cost**
		(msec)	Exp.Cost	(msec)	Exp.Cost		**Increase**
F15	9250	80.32	45.10	73.12	46.23	9.84%	2.27%
Hand	17135	140.54	73.24	130.33	74.12	7.83%	1.20%
Mesh	56172	480.53	83.36	401.98	85.21	19.54%	2.21%
Spheres	66432	540.28	92.17	450.25	94.46	20.06%	2.51%
Bunny	69451	692.36	96.31	550.35	97.85	25.81%	1.62%
Fairy	174117	1450.4	105.29	1120.2	111.32	29.46%	5.72%

6 Conclusion and Future Work

In this paper, we have presented an algorithmic approach to improve and optimize the split plane location search criteria used for the *kd*-tree construction in fast ray tracing algorithms. The major contribution is the combination of golden section search criteria with parabolic interpolation. The golden section search provides us with a better initial guess for the sampling locations of the cost function. The better the initial guess, the faster is the convergence of the algorithm. We have demonstrated in section 3 that the algorithm reaches the optimum within two iterations in a real scenario taken from [7]. Our idea gives us two important parameters, first, the initial guess and second, towards which direction we should move to reach the optimum. The information is very critical in order to get the convergence faster.

We have evaluated two aspects of our proposed modified model. First, we evaluated how well it predicts the actual split locations of the planes when building the *kd*-tree using Matlab models. Second, we have implemented our modified model in a ray tracer and compared its performance to a standard SAH ray tracer. We have used six different scenes, with varying complexity levels. The performance improvement is small for the scenes with low complexity as the *kd*-tree depth is small. As the *kd*-tree depth increases for the more complex scenes, the performance improvement increases and we have successfully demonstrated an improvement of 30% for a scene complexity of around 170k polygons with only less than 6% degradation of the tree quality. We expect the performance difference to increase when the complexity and the number of objects in a scene increases.

Further use of the technique could be demonstrated with the combination of SIMD support and spline interpolation where the cost function is very ill-behaved or changes abruptly. We hope to increase the tree quality even more with this technique and also intend to implement another version of this algorithm on dynamic scenes with multiple objects like Fairy in Fig.5.

References

1. Amanatides, J., Woo, A.: A Fast Voxel Traversal Algorithm. In: Proceeding of Eurograph -ics 1987, pp. 3–10 (August 1987)
2. Benthin, C.: Realtime Raytracing on Current CPU Architectures. PhD thesis, Saarland University (2006)
3. Chang, A.Y.: Theoretical and Experimental Aspects of Ray Shooting. PhD Thesis, Polytechnic University, New York (May 2004)
4. Fussell, D.S., Subramanian, K.R.: Automatic Termination Criteria for Ray tracing Hierarchies. In: Proceedings of Graphics Interface 1991 (GI 91), Calgary, Canada, pp. 93–100 (June 1991)
5. Havran, V.: Heuristic Ray Shooting Algorithm. PhD thesis, Czech Technical University, Prague (2001)
6. Havran, V., Herzog, R., Seidel, H.-P.: On fast construction of spatial hierarchies for ray tracing. In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 71–80 (September 2006)
7. Hunt, W., Mark, W., Stoll, G.: Fast kd-tree construction with an adaptive error-bounded heuristic. In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 81–88 (September 2006)
8. MacDonald, J.D., Booth, K.S.: Heuristics for ray tracing using space subdivision. The Visual Computer 6(3), 153–166 (1990)
9. Kaplan, M.: The Use of Spatial Coherence in Ray Tracing. In: ACM SIGGRAPH 1985 Course Notes 11, pp. 22–26 (July 1985)
10. Popov, S., Gunther, J., Seidel, H.-P., Slusallek, P.: Experiences with Streaming Construction of SAH KD-Trees. In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 89–94 (September 2006)
11. Redmonds, S.J., Heneghan, C.: A method for initializing the K-means clustering algorithm using kd-trees. Pattern Recognition Letters 28(8), 965–973 (2007)
12. Stern, H.: Nearest Neighbor Matching Using kd-Trees. PhD thesis, Dalhousie University, Halifax, Nova Scotia (August 2002)
13. Stoll, G.: Part I: Introduction to Realtime Ray Tracing. SIGGRAPH 2005 Course on Interactive Ray Tracing (2005)
14. Wald, I.: Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, Saarbrucken, Germany

15. Zara, J.: Speeding Up Ray Tracing - SW and HW Approaches. In: Proceedings of 11th Spring Conference on Computer Graphics (SSCG 1995), Bratislava, Slovakia, pp. 1–16 (May 1995)
16. Cleary, J.G., Wyvill, G.: Analysis of an algorithm for fast ray tracing using uniform space subdivision. The Visual Computer, (4), 65–83 (1988)
17. MacDonald, J.D., Booth, K.S.: Heuristics for ray tracing using space subdivision. The Visual Computer (6), 153–166 (1990)
18. Whang, K.-Y., Song, J.-W., Chang, J.-W., Kim, J.-Y., Cho, W.-S., Park, C.-M., Song, I.-Y.: An adaptive octree for efficient ray tracing. IEEE Transactions on Visualization and Computer Graphics 1(4), 343–349 (1995)
19. Stanford 3D scanning repository.: http://graphics.stanford.edu/ data/3Dscanrep/
20. Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-d tree GPU raytracing. In: Symposium on Interactive 3D Graphics I3D, pp. 167–174 (2007)
21. Foley, T., Sugerman, J.: KD-tree acceleration structures for a GPU raytracer. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 15–22 (2005)