Fast Ray Tracing for Modern General Purpose CPU

Jim Hurley, Alexander Kapustin, Alexander Reshetov, Alexei Soupikov
Intel

Abstract

We present a study of the system implications of various aspects of the classic Ray Tracing algorithm. We show how these algorithms can be modified to yield the highest performance on any given system, and present some of the results that we obtained.

1. Introduction

Computer engineering is a tough business, due to physical limitations of the current generation of computers we are forced to overcome them with some extra ingenuity, only to have a new generation of computers come along for which all this extra effort is overkill.

Perhaps, nowhere this process is more distinct, that in compute graphics. Countless approaches to hidden surface removal were washed away with an arrival of the hardware z-buffer acceleration. Current trends of extending programmability and functionality of graphical devices will have, no doubt, no lesser effect on the mainstream 3D computer graphics.

In this article, we will present the status of software ray tracing acceleration techniques, compared with the traditional graphics pipeline. We performed analysis of real-time ray tracing (RTRT) performance requirements; We will present a brief description of a global illumination system with performance in the 2M ray segments/sec range on a single general purpose IA-32 PC, and speculate about the feasibility of general-purpose processors for RTRT.

2. Performance

The major conceptual differences between the traditional graphics pipeline and the ray-tracing approach are as follows:

- Traditional pipeline: all (visible) objects in the scene are processed one by one. The resulting image is computed as a superposition of projections of objects into the viewing plane.
- Ray-tracing: Rays are cast from the eye position into the scene. The resulting image is computed by filtering the resulting colors for all rays coming though a particular pixel on the screen.

There are countless variations of these basic approaches, we are primarily interested in the performance behavior with respect to the number of objects in the scene. For the sake of simplicity, we will be talking only about tessellated polygonal objects and designate N as the total number of triangles in the scene.

For the traditional rasterization pipeline, it is hard to get rid of O(N) dependency for the rendering time. Even if most of the objects in the scene are not visible, they have to be processed nevertheless. There are different approaches to address this problem either through computing visibility offline or with some advanced run time computations [7].

At the same time, visibility determination is implicitly embedded into ray tracing approach; you don't have to do anything else except to following the ray! Ray tracing also offers great scalability due to various types of parallelism, the simplicity of programming shading computations and some other advantages, compared with traditional graphics pipeline (see [1]).

However, the computational costs of ray tracing are very high and until recently were not considered suitable for hardware implementation [3].

It is well known, that by using data preprocessing, the ray-object intersection time is proportional to O(log(N)) [2]. This is can be only be achieved with some geometric data partitioning to reduce the number of intersection tests from the naïve O(N) to a logarithmic value. The pre-processing time, varies from O(N) for simple uniform grids to $O(N^4)$ and higher for more elaborate schemes.

Let's first consider a static scene. The rendering time will be a multiplier of the following factors:

- N_p – Pixels on a display. Let's assume N_p = 1024 * 1024.
- N_r – rays coming through any single pixel. Let's assume N_r = 5.
- N_s – secondary (shadow) rays per primary ray. It is commonly agreed, that it is hard to achieve a good quality with value of N_s less than 100.
- N_x – average number of ray/triangle intersection tests.

- T_x – average processing time in CPU cycles for ray/triangle intersection, including memory access.
- T_t – average traversal time in CPU cycles. Here we will assume kd-tree spatial partitioning structures.
- For a well balanced acceleration structure, the traversal time T_t should be approximately equal to total time spent on intersection tests, which is N_xT_x.

The values N_x and T_x are strongly dependent on the implementation of the intersection test and cannot be estimated independently. We ran tests for two different implementations:

1. Rays tested directly against individual triangles. In this case, $T_x \approx 130$ and $N_x \approx 10$ for a scene with 10^6 polygons.
2. Vectorized code was used for the intersection. It helped to reduce T_x to 50 cycles (CPU and memory access), while N_x typically increases to 20.

These are approximate numbers for an average scene with about one million triangles. It is clear that vector instructions help improve the average processing time for one triangle by ~2.5x, however, the overall performance improvement computed as N_xT_x is improved only about 30%. This is because vector instructions require aligned data memory layout, in addition, processing of 1, 2, 3, or 4 triangles takes approximately the same time.

So, multiplying all these numbers and taking into account the traversal time (approximately equal to the intersection time), the processing of one frame should require:

$N_f = N_p N_r N_s (N_xT_x + N_xT_x) \approx 10^{12}$ cycles.

A rendering system with the goal of a **constant frame rate** will necessarily involve the following steps:

1. The required modeling accuracy is estimated for each region of space (ratio of triangles cast onto a single visible pixel).
2. A spatial partition structure is created or updated, using some pre-computed Multi-Resolution Mesh (MRM) information. This step of the algorithm will use only polygons at the required level of detail. It seems possible to dynamically change objects, represented as MRMs, without recomputing the whole mesh.
3. The size of the spatial partitioning structure above, will not depend on number of polygons in the scene, but only on the screen resolution.
4. Consequently, the rendering will be performed in a constant time.

This algorithm, relies on our ability to dynamically create acceleration structures, without physically processing every triangle in the scene. There are many examples where this type of processing is almost trivial.

3. Spatial Subdivision

3.1. Memory & Performance

3D scenes may have thousands or millions of polygons. Rendering these scenes requires repetition of some elementary operations multiple times such as finding the intersection of a sample ray with the scene geometry. One way to increase the speed of these intersection calculations is to reduce the number of primitives tested for a given ray by pre-processing the geometry data and storing this information in a dedicated data structure. This data structure is usually called an “acceleration structure” in the context of the field of global illumination.

Acceleration structures realize a trade-off between memory storage, computation and perusal of a database. This pre-processing may have to be performed for every frame as the “acceleration structure” effectively optimizes the way that rays are likely to pass through the database of polygons, as the “camera” moves, the ideal layout of the structure will change too.

It is important to note that as the size of the acceleration structure increases, the relative performance improvement becomes smaller and smaller. In Figure 1 below, the rendering time of a sample scene (Image 1) is represented with respect to the size of the acceleration structure. It is clear, that the rendering time is improved greatly until the number of nodes in the acceleration structure approaches ½ of the total triangles in the scene. Due to the nature of the intersection operation, most of the best “acceleration structures” are based on some form of space partitioning. It was shown [9] that a kd-tree consistently outperforms other partitioning schemes.

Two major reasons for this can be summarized as follows:

1. Any algorithm, which uses axis-aligned split planes, can be converted into a kd-tree form.
2. Traversal of kd-trees can be implemented very effectively on current PC architecture.

Different computer architectures may require different approaches to optimal kd-tree creation, storing and traversal, or even completely different structures. We will analyze these issues later. Basically, there are 3 major steps in using any acceleration structure: Creation, Packing & Traversal.

Image 1. Sample scene 54K tri, 31 lights.

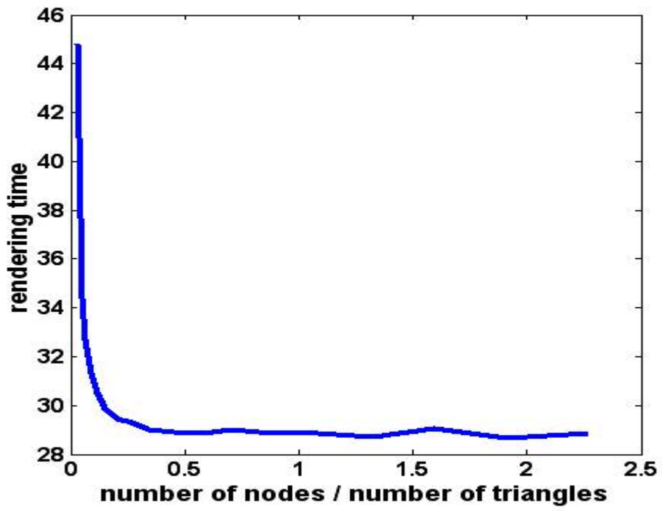


Figure 1. Render time *Vs* structure size

Kd-tree creation

Geometrically, a kd-tree can be represented as a binary tree. To create a kd-tree, the following 2 operations are repeated recursively.

1. One of the 3 axes (X,Y,Z) is chosen and some split value on this axis is selected. The axis number and split value defines a separation plane.
2. The first operation is repeated with all geometric elements to the "left" of the separation plane and with all elements to the "right" from this plane.

In Figure 2, the first separation plane is shown along axis OY. This separation plane splits the original cell into two. All triangles are color-coded, depending on weather they end up in:

- The left sub-cell (blue)
- The right sub-cell (green)
- both (red)

To continue splitting, we will use blue and red triangles with the left sub-cell and green and red triangles with the right sub-cell. Figure 3 shows the results of multiple splits for the same geometry. It is easy to see that some of the cells are empty. This is a major advantage of the kd-tree approach as the effective purging of empty space is done automatically by the algorithm and a traversal of empty cells costs almost nothing (see 0).

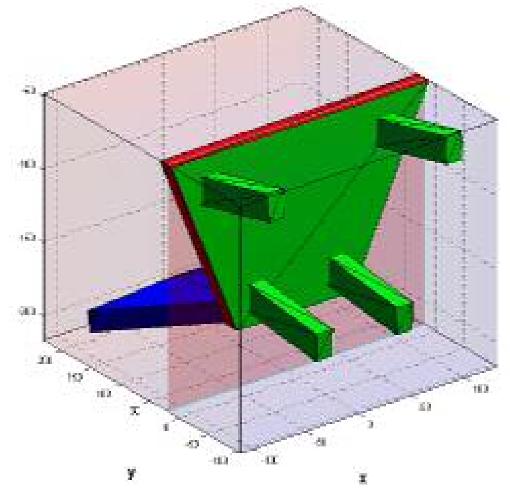


Figure 2. 1st separation plane, y = 18.9

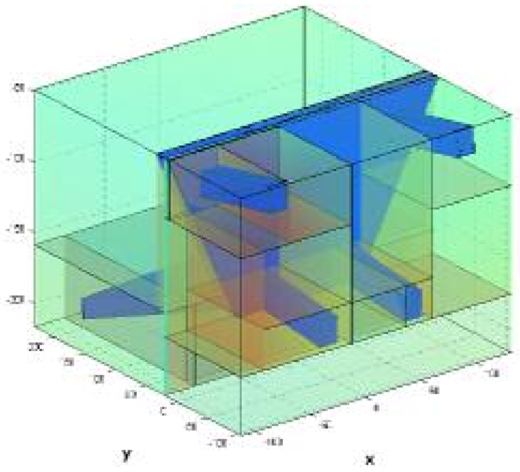


Figure 3. After Multiple splits

3.2. Split Plane Selection

There are different approaches to the selection of the separation plane, yielding different performance results. However, there are some discrete heuristic approaches with reasonable performance. This is usually based on a rough estimation of the traversal cost for a built kd-tree. The partition plane is chosen to minimize this cost. One natural selection for the cost function is the total rendering time for a particular scene. However, this approach has some major shortcomings:

- Rendering time depends on not only how geometry is partitioned, but also how structures are packed in memory and traversed during run-time. It is paramount to consider all packing and traversal algorithms in terms of CPU cache performance.
- For static scenes, acceleration structures are reused for different frames. One particular partitioning scheme may exhibit very good performance for some camera positions, but behave rather poorly for other viewpoints.
- We suggest some very simple local approximation of rendering time, which helps to find best split plane and, at the same time, provides some automatic termination criteria (whether to continue splitting a cell or create a leaf node).

For all splits, we estimate the following delta, approximating the improvement in the rendering time if the split is made:

$$\begin{aligned} Cost_{delta} &= Cost_{no\text{-}split} - Cost_{split} \\ Cost_{no\text{-}split} &= r_{12} * (t_t + n_{12} * t_x) \\ Cost_{split} &= r_{12} * t_t + \\ &\quad r_1 * (t_t + n_1 * t_x) + \\ &\quad r_2 * (t_t + n_2 * t_x) \end{aligned} \qquad (1)$$

where

n_1 - triangles in one (let's say left) sub-cell;

n_2 - triangles in another (right) sub-cell;

n_{12} - triangles in un-split cell ($n_1 + n_2 \geq n_{12}$);

t_t - average time to traverse a cell in kd-tree;

t_x - average ray/triangle intersection time;

r_1 - rays passing through left sub-cell;

r_2 - rays passing through right sub-cell;

r_{12} - rays passing through un-split sub-cell

The values t_t and t_x include both CPU processing time and memory access overhead and cannot be easily computed directly. We will describe later a very simple method of estimating these values using some linear regression model.

Among all possible splits, the one that yields the best positive value of $Cost_{delta}$ is chosen. If such a split does not exist (all values of $Cost_{delta}$ are negative), then a terminal leaf is created. During kd-tree traversal, all rays, passing through this cell, will be tested against all n_{12} triangles.

By observing equation (1), it is easy to note that the absolute values of t_t and t_x do not matter, only the ratio t_t / t_x. The same is true for the numbers of rays passing through cells. In fact, the values of **r** can be approximated with the area of the appropriate cell, as it well known from the literature ([5], [9], [10])

The costs (1) are linearly dependent on the number of triangles in the cells. Here are some further criteria:

1. Restrict maximum tree depth.
2. Restrict minimum cell size
3. Don't create very small cells (by area or volume).
4. We can tilt expression (1) in favor of splits which create some empty cells. By purging extra empty space, we are improving global characteristics of the acceleration structure. We found, that by reducing the cost associated with empty splits by 20%, we will get about 5% overall improvement.
5. We can consider ratio t_t / t_x not as a given value, but rather as a parameter of the model.
6. There is nothing to prevent us from introducing some additional parameters into the formula (1). For example, we can use some exponential expressions like $\textbf{area}^{\textbf{p}}$, where **p** is some parameter.

By varying parameters 1-6, we can create different acceleration structures. It may happen that some of them will be better for a particular viewpoint or model. Suppose we had a series of measurements with different acceleration structures and accumulated the following:

T_r - total rendering time ($T_r = [T_1, T_2, \ldots T_n]$);

N_t - traversed cells (vector of size n with each component representing total number of traversed cells in particular experiment);

N_x - number of intersection tests.

By neglecting the cost of shading and other extra processing, we may assume that

$$T_r = N_t * t_t + N_x * t_x \qquad (2)$$

Note that, by definition, expressions for t_t and t_x comprise CPU cost and "average" memory access cost. Now we can solve the over-defined system of equations (2) to find approx values of the unknown variables t_t and t_x.

For Image 1, we found the values for t_t and t_x to be 70.91 and 49.33 CPU cycles. It is possible to feed these values back into equations (2) to estimate an error of these estimations. For n = 150 (number of experiments), this error was found to be about 2%, which is rather remarkable, given the variability of operating environment (Windows 2000) and different sizes of the data sets.

For different scenes, the values for t_t and t_x will vary due to the different cache behavior and different number of polygons. Image 2 produced values of t_{t} = 101.74 and t_x = 101.74. The bigger values are primarily due to the bigger size of the model. The ratio of t_t to t_x was found to be 1.4375 for Image 1 and 1.399 for Image2, remarkably consistent.

Image 2. Bar scene (234K triangles, 69 lights)

3.3. Possible Split Planes

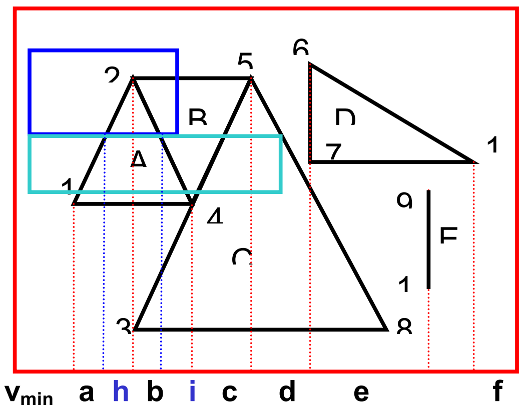
For static scenes, the pre-processing cost is amortized among many frames. In a dynamic situation, we are trying to minimize the total cost: acceleration structure creation/update + rendering. For a scene containing N triangles, the first term (creation) behaves, at best, as O(N) (most likely O(N log(N)). At the same time, rendering time is proportional to log(N). Consequently, creation/rendering balance will be different for each scene and will be dominated by the creation cost for very big scenes. It will make sense then to use a very simple approach for selecting the split planes.

One possible approach is to always choose the median of the longest side. Even in this case, it may make sense to evaluate a cost approximation as a way of determining whether to stop or continue splitting.

Conceptually, the kd-tree creation algorithm is very simple: among all possible splits one is selected, which minimizes rendering cost. This operation is repeated recursively, until termination criteria are met. Due to the necessity of having a very effective kd-tree traversal algorithm, only planes orthogonal to one of the three axes are considered. So, to completely define a split, we need to know its axis number and some position on it. We will analyze 2 major approaches in finding possible splits:

1. Vertex-based: only vertices are considered when finding out possible split planes.
2. Intersection-based: triangle/cell intersections are considered as well.

Figure 4. Sample cell and possible split positions



One of the simplest approaches to compute a set of possible splits is represented in Figure 6. For the sake of simplicity, everything is represented in 2D. We are trying to define a set of possible splits for the red cell, which contains triangles A, B, C, D, and E. If we consider only the vertices, which are in the cell, the possible splits will be represented by positions a, b, c, d, e, f, and g. We then compute the cost approximation for each split and choose the best one (among all 3 axes).

This approach works well for a while, however it has some major drawbacks. For example, for the blue cell, the possible set includes only one position – b. For the cyan cell, this approach will not produce any candidate split at all. If, however, we include triangle/cell intersection points into the consideration, we may find better splits. For the blue cell, it may be h, which creates one empty cell and one cell containing 2 triangles.

To evaluate the cost function, we need to know the number of triangles to the left and to the right of the split (see equation (1)). Vertex projections can be used as a crude approximation for this number. For example, if any of the projections for a particular triangle are less than the split value, we can assume that this triangle belongs to the left sub-cell.

Another critical issue is what to do with triangles which are entirely contained in a split plane (like triangle E on Figure 4 for split plane f). Should it go to the left, right or both sub-cells? Since we are trying to minimize the cost expression (1), the following algorithm achieves this goal:

> Triangles, which are completely contained in a split plane, go into the smaller sub-cell, except for the

case when we can create an empty cell by moving these triangles to the bigger sub-cell.

We experimented with the two different approaches to find the split plane. The gains depend on whether ray/triangle intersections are computed as 1-to-1 or 1-to-4 (using SSE engine). In the first case, the gain is about 30%, 10% in the second. Since using SSE results in a smaller value for t_x – average ray/triangle intersection time in equation (1), the cost-based termination criteria will tend to create bigger leaf nodes with more triangles in them. Therefore, in this case, finding the exact triangle/cell intersections is less beneficial.

Another observation, confirmed in the literature, is that accurately estimating the cost function is more important for deeper levels of the tree, where cells are smaller. Indeed, closer to the top of the tree, most of the triangles are completely included into each cell (see red cell on Figure), while at the bottom most of the triangles are intersecting cell's boundaries. Based on this, we may adopt a hybrid approach: using vertex projections at higher levels of the tree, while resorting to finding intersections at lower levels.

Another cost function technique is based on restricting a set of possible splits. To explain it, let's look more closely at expression (1). For each possible split, we have to find the number of triangles to the left and to the right of the split. It can be implemented using the following three steps:

1. For each triangle, find candidate split positions and store them in memory. Note that each triangle can produce either 1 or 2 candidate positions, because we are interested only in extreme positions.
2. An array of possible splits is sorted out.
3. We then loop through this array, computing the cost function delta. For each entry, we immediately know the split position and can easily compute the number of triangles to the left and to the right of this position

Since these steps involve sorting, for a scene with N triangles, the kd-tree creation algorithm will require O(N Log^2(N)) operations (the tree will have ~ Log(N) levels, each level requiring ~ N Log(N) operations).

We can reduce the N Log(N) number of operations, required by the sorting step by reducing the accuracy of computations. If we use only the first k bits, sorting may be completed in O(k n) operations [12]. One algorithm, achieving this performance, is called Pigeonhole Sorting. This can be easily implemented using a binning approach. For a required accuracy of k bits, we split an interval of possible splits [v_{min}, v_{max}] into 2^k equal bins. We will also need three arrays of size 2^k+1, storing the following values:

1. Candidate split value, closest to the k-th bin boundary (bins are spaced equally, while candidate values may not).
2. Cumulative values of number of triangles to the left and to the right from the k-th split value.

All these values can be easily computed in the triangle processing loop, similar to steps 1-3, described earlier. To study how bin size influences performance, we rendered Image 1 with different numbers of bins and plotted the execution time on Figure 8. This graph shows that for numbers of bins ≥ 2^6 = 32, it doesn't really affect run-time performance at all (smaller number of bins requires less pre-processing time though).

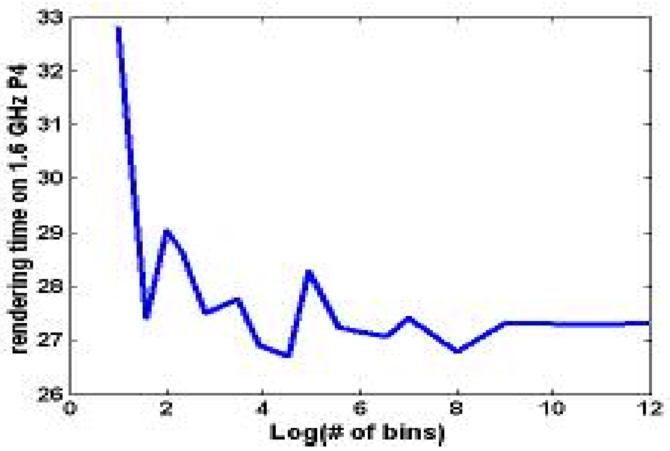


Figure 5. Impact of binning size on performance

This effect clearly demonstrates the capacity for adaptation of the kd-tree algorithm: even though it cannot use the best possible split position at some level due to the discrete nature of the binning method, it will do so when the binning size decreases (due to the smaller cell's size).

3.4. Data Storage

Since any acceleration structure requires additional memory, it is important to look for an optimal data layout. There are two closely related characteristics of an optimal layout:

1. Overall memory footprint is minimized.
2. The values, which most likely will be used together, should be placed close to each other. Usually, when the parent node of a kd-tree is accessed, one or both children are accessed next.

Pursuing goal 1 per se is not enough, as poor data layout may trash cache performance. The kd-tree traversal algorithm, which always starts at the top of the tree, requires only 2 values to be stored explicitly. They are the axis number and the position of split plane.

The kd-tree can be stored as a complete binary tree (each node having 2 children) computing the children's addresses on the fly. However, due to the nature of the optimization criteria expressed by equation (1), kd-trees for real scenes are not balanced. To effectively represent unbalanced trees, addresses or offsets of nodes may be used. If records for both children are kept together, only one address field is required.

In our implementation, each kd-tree node is described by 2 4-byte fields. Internal nodes include the axis number, split value and address of the left child. Leaf nodes include the address of the first triangle in the leaf plus the size of the data. Since all address fields are aligned by 8-bytes, it can easily be combined with a 2-bit value, representing the axis number.

Kd-tree Traversal

The kd-tree traversal procedure is the most time consuming part of the GI code. It may take up to 75% of all costs. Code optimization is critical to improve performance. Essential points of this process are:

- use predicates to reduce the number of code branches;

- avoid recursive function calls;

- stack and memory layout optimization;

4. Intersection Test

Global illumination solutions based on the ray tracing technique are considered to be expensive for the following reasons:

a. ray-primitive intersection test is expensive

b. huge number of primitives tested by a given ray for intersection.

The previous discussion describes the ways to address issue (b) by reducing the number of primitives to be tested using a space partitioning approach. Issue (a) also deserves thorough consideration.

Lots of effort has been spent to find an efficient solution to the ray-primitive intersection test. The most efficient solutions pursue the point of balance between memory accesses and the computations for any given platform. Once this balance point is achieved exploiting data parallelism is the way to increase performance further. The next question coming into consideration is how to vectorize the problem. Two straightforward ways are observed. The first is to test multiple rays against one primitive (so called N:1 approach). The second is testing one ray against some number of primitives (1:N approach correspondingly). The first approach allows for using multiple types of primitives and seems scalable to infinity. Though in practice there are certain limitations imposed by platform architecture like

a. limited register (or fast memory) space doesn't allow to store long vectors,

b. non-random access to vector elements requires data shuffling or repacking

c. data alignment requirements.

Trying to resolve these issues by using various shuffling instructions usually steals all benefits given by parallelism or makes the performance even worse. The reasons for such performance degeneration lay in the non-coherent control flows for the given set of rays and in the SIMD nature of vector instructions present in modern processors.

The 1:N approach is more favorable for SIMD usage because the set of primitives can be formatted in suitable way during scene preprocessing. There is one limitation though, the algorithms should stick to one primitive for complex object representation. So we use triangles as one and only one allowed primitive.

Vectorized test takes 21 clock ticks/triangle if no intersection happens and 46 in worst case (intersection), while scalar code takes typically 160-120. Numbers measured on PIV 1600Mhz processor.

5. Results

Fully functional GI code was implemented, covering all of the results of the topics discussed previously. The code was written in C++. The code works only with triangulated objects, texturing (with mip-maps) is supported. Point lights and spotlights are implemented. The code was tested on 2.2GHz P4 machine (single CPU). For testing purposes we used scenes courtesy of Cornell and Saarland Universities. The testing results for various scenes are shown in Figure 6 below. Performance is measured as number of ray – scene intersections per second.

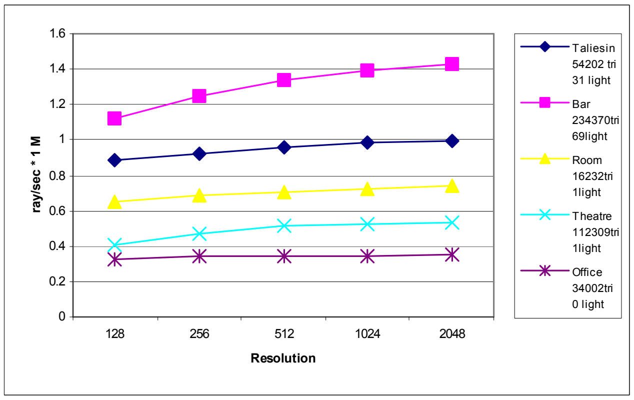


Figure 5. Testing results for various scenes

As we can see, performance is only weakly dependent upon scene size thanks to the sub linear time of ray scene intersection for Kd-trees. The number of lights multiplies the cost of shadow rays. Large images have more coherent primary rays, so performance efficiency slightly increases with increasing image size.

Image 2. Bar scene with specular component

Image 3. Room scene with lambertian materials

Image 4. Room scene: lambertian & specular effects.

6. Conclusion and Future Work

On modern desktop PCs it is possible to achieve approx. 2M Rayseg/S, performance varies by scene. This performance is achieved by paying very careful attention to data structure creation, packing and perusal, also important are generating rays that maximize likely coherency of data fetched from these data structures, additional benefit is achieved by exploiting vectorization opportunities. It would be especially interesting to analyze hybrid approach: N:1 for tree traversal and 1:N for intersection test. In the past, the achievable performance was much lower, and acceleration structures were pre-computed and used only to generate static scenes. With this kind of performance now interactive frame rates can be achieved, and all of a sudden the cost of building acceleration structures on the fly becomes an important issue. Hence, our next steps will be to investigate methods by which we can build such data structures efficiently while not losing too much of the performance that an optimally constructed acceleration structure would yield.

7. Bibliography

1. Interactive Rendering with Coherent Ray Tracing, Ingo Wald, Philipp Slusallek, Carsten Benthin, EG 2001 Proceedings
2. Stabbing and ray shooting in 3 dimensional space, Marco Pellegrini, Proc. 6th Symp. Computational Geometry, ACM, Jun 1990, pp. 177-186
3. Ray Tracing on Programmable Graphics Hardware, Timothy J. Purcell Ian Buck William R. Mark, Pat Hanrahan, Processing of Siggraph 2002
4. State of the Art in Interactive Ray Tracing, Ingo Wald and Philipp Slusallek, Proceedings of Eurographics 2001.
5. LCTS: Ray Shooting using Longest Common Traversal Sequences, Vlastimil Havran, Jiri Bittner, Proceedings of Eurographics (EG'00)
6. On Improving Kd-Trees For Ray Shooting, Vlastimil Havran Jiri Bittner, http://www.mpi-sb.mpg.de/~havran/ARTICLES/wscg2002.pdf
7. Visibility Preprocessing for Urban Scenes using Line Space Subdivision, Jiri Bittner, Peter Wonka, Michael Wimmer, Proceedings of Pacific Graphics (PG'01)
8. The IBM T221 Display, http://www.monitoroutlet.com/342898.html
9. Heuristic Ray Shooting Algorithms, Ph.D. Thesis by Vlastimil Havran, November 2000
10. MacDonald,J, Booth,K: Heuristics for ray tracing using space subdivision, The Visual Computer, Vol. 6, No. 3, pp. 153--166, 1990
11. Intel® Software Development Products, http://developer.intel.com/software/products/compilers/c60/index.htm
12. Wikipedia: Sort Algorithm. http://www.wikipedia.com/wiki/Sort+algorithm