Speed optimized Recursive Ray-tracer with KD-Tree and SSE vector mathematics

Balázs Tóth*

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

This paper presents a new approach to achieve interactive frame rates with the ray-tracing image synthesis method used originally for high quality, off-line rendering. The rendering system uses a spatial subdivision algorithm and SIMD instructions to speed up the rendering process.

Keywords: Raytracing, SIMD, KD-Tree

1 Introduction

Interactive rendering systems provide a powerful way to explore complex environments. Until recently the processing power of computers did not allow us to achieve high frame rates with ray-tracing based algorithms. The only interactive rendering methods were hardware accelerated polygonal rendering systems, which are less flexible and are poorer in providing sophisticated lighting effects.
Software-only methods are easy to modify and extend, which makes them a good candidate to experiment with various interaction and rendering methods. Nowadays an optimized ray tracer-based software rendering system can reach the performance of a polygonal algorithm.
The recursive ray-tracing algorithm is fairly easy to understand and implement, but it's powerful enough to examine the possibilities of the algorithmic and implementational optimizations. With the use of spatial subdivision algorithm and some optimization we can reach decent frame rates.

2 Elements of the ray-tracer

In this section we describe the key elements of our ray-tracing engine. These parts are implemented with speed and efficiency in mind. Our implementation is based on a recursive ray-tracing algorithm and a subdivision structure to speed up the object searching. Our vector operators are optimized with SSE instructions, because these are frequently used in the rendering process.

2.1 Recursive ray-tracing algorithm

The idea behind this algorithm is to simulate the path of light rays [9]. The most important part of the algorithm is the light-trace function. In this function we must determine the object the ray hits first. In the next step we calculate the direct contributions of the light sources with the diffuse and specular components of the hit object's material. In order to handle the reflective and translucent materials, we must determine the next object through the hit point. In this case we spawn new rays (the origins are the hit points and the directions are calculated using Snell's law) and trace them to determine these components.

2.2 Intersection test

The ray object intersection test is an important part of the ray-tracer. We must test the ray object intersection fast, because in a ray-tracing system the 90% of computation time is spent on these calculations.

2.2.1 Ray triangle intersection

Fast ray triangle intersection test algorithm has long been an active field of research in computer graphics and has lead to a large variety of algorithms. Our implementation is based on baricentryc coordinate test [8].
In the first step we calculate the distance between the ray origin and the plane that embeds the triangle. With this test we can limit the test range. If the distance is larger than the limit the algorithm stops. If the triangle passes the distance test we calculate the baricentryc coordinates of the hit point by equation $H = O + d_{plane}D$, where O is the origin of the ray, d_{plane} is the previously calculated distance between the origin and the plane of the triangle and D is the direction of the ray. The baricentryc coordinate of H can be calculated by solving the $H = \alpha A + \beta B + \gamma C$[1] equation of the triangle. If the barycentric coordinates have positive values the intersection point is in the triangle.
This intersection test scheme can be optimized by exploiting the fact that projecting both the triangle and the hit-

*tbalazs@sch.bme.hu

[1] The baricentryc coordinates of the H are α, β, γ and $\alpha + \beta + \gamma = 1$

point onto any plane[2] do not change the baricentryc coordinates. If we project them onto one of the 2D coordinate planes[3], all further computations can be performed in 2D. For reasons of numerical stability, we should project to the plane in which the triangle has maximum projected area. After the projection, the equation is $H^{'} = \alpha A^{'} + \beta B^{'} + \gamma C^{'}$, where $A^{'}$, $B^{'}$, $C^{'}$ and $H^{'}$ are the projected points. With the $\alpha = 1 - \beta - \gamma$ substitution we can rearrange the equation to $\beta(B^{'} - A^{'}) + \gamma(C^{'} - A^{'}) = H^{'} - A^{'}$. In 2D this can be solved[4] as

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

2.2.2 Ray sphere intersection

The sphere object is defined by its center[5] and radius[6]. The points in the sphere can be represented by $(X - X_c)^2 - (Y - Y_c)^2 - (Z - Z_c)^2 = S_r^2$ equation. If we substitute the ray equation[7] to the sphere equation, we can solve it algebraically[8].

$$A = X_d^2 + Y_d^2 + Z_d^2,$$

$$B = 2((X_d(X_o - X_c) + Y_d(Y_o - Y_c) + Z_d(Z_o - Z_c)),$$

$$C = (X_o - X_c)^2 + (Y_o - Y_c)^2 + (Z_o - Z_c)^2 - S_r^2.$$

The solution of the quadratic equations is

$$t_0, t_1 = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

If t_0, t_1 are less than zero then there is no intersection. If they are greater than zero, then the smaller is the closest intersection point. The intersection point is

$$R_{i,x} = X_o + X_d t_i,$$

$$R_{i,y} = Y_o + Y_d t_i,$$

$$R_{i,z} = Z_o + Z_d t_i.$$

If t_0, t_1 has opposite signs, the ray is spawned inside the sphere. In this case we terminate the calculation.

2.3 KD-Tree implementation

The best subdivision method is based on a special data structure called kd-tree. My research is based on Vlastimil Havran's thesis [5], who did an extensive study of available spatial subdivision schemes (regular grids, nested grids, octrees and kd-trees). He concluded that kd-trees beat others in most cases. It was also shown that the average number of intersection tests to find the closest intersection can be made as small as 2-3 independently of the number of objects [7][8].

The kd-tree is an axis-aligned Binary Space Partitioning tree. The space is partitioned by splitting it into two halves. The halves are processed recursively until every partition contains only one object. The most important difference compared to other schemes is that the position of the partitioning plane is axis aligned but not fixed. The use of the axis-aligned splitting planes has several advantages. Most importantly, it makes intersection test inexpensive with low memory footage of the tree [6].

2.3.1 Building the tree

To build the tree, we must determine the right positions of the splitting planes. The simplest method is to choose it in a way which ensures that the numbers of objects on both sides of the plane are roughly the same. This method is not the best because it doesn't produce empty nodes and the ray-tracer must check all objects in each node during the ray traversal.

Better trees can be constructed using a heuristic splitting rule. A good heuristic tries to isolate the empty spaces. In such nodes the traversal algorithm can travel through without expensive intersection tests. Such a heuristic is the Surface Area Heuristic. The basic idea is that the probability of a ray hitting object is related to it's surface area. The area of the node is

$$width \times length \times height.$$

The cost of traveling in a node is

$$Travel + Area \times ObjectsInTheNode \times IntersectionTest,$$

where *Travel* is a constant traversing cost in an empty node and the *Intersection Test* is a constant cost of ray-object intersection test. The splitting of the node produces two new nodes, so the splitting cost is calculated by summing the new node's costs. If we always use the less expensive splitting plane, we get a good kd-tree. To find the good splitting plane we must test all the possible planes. There are many planes to choose from, but the number of the interesting positions is limited. The limitations are that the splitting plane must be axis aligned and must touch the border of at least one object.

Even with these constraints there are a large number of candidates. So building a kd-tree is a slow process, but with a static scene we must build the tree only once. The hit search with a good kd-tree is four times faster than the regular grid and many times faster than the naive test-with-all-objects method.

2.3.2 Storing the tree

The kd-tree is built up from nodes, that stores the position of the splitting plane, a flag that indicates whether the node

[2] Except planes that orthogonal to the plane of the triangle.
[3] XY, XZ, YZ
[4] $b = C^{'} - A^{'}, c = B^{'} - A^{'}, h = H^{'} - A^{'}$
[5] $S_c = [X_c, Y_c, Z_c]$
[6] S_r
[7] $R(t) = O + t * D$, where $t > 0$, O the origin of the ray and D is the direction
[8] $D = [X_d, Y_d, Z_d], O = [X_o, Y_o, Z_o]$

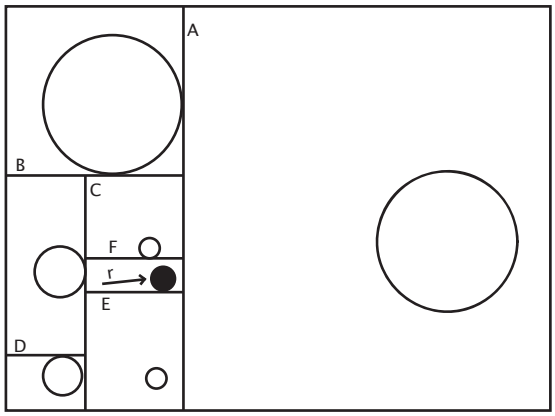

Figure 1: Example of the subdivision

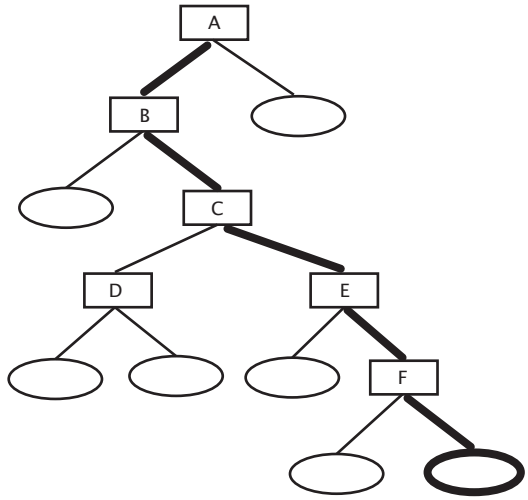

Figure 2: Traversing the tree

is a leaf node or not. If the node is not a leaf, then it must contain a pointer to its left child node. If the node is a leaf node it contains a pointer to the list of objects in the node. These data members of the nodes can be stored in 8 bytes. We allocate the child nodes in pair at a 16 bytes boundary. With this allocation scheme we can save a pointer in each node, because the position of the right child node is right after the left node. The size of the node pair is 16 bytes, therefore in a single 64 kbytes cache line we can store four node pairs. When the traversal algorithm reads the left child node, the right child is loaded into the cache because of the behavior of the cache loading process. This improves cache performance if the series of the successive rays walk through the same nodes.

2.3.3 Traversing the tree

The ray traversal algorithm is a simple repetitive point-location search in the tree along the ray path. First we determine the point of the ray origin in the tree. If the node is not empty we test the intersection of the objects with the ray and select the closest intersection point. If the node is empty or we didn't find an intersection, we determine the exit point of the node along the ray's direction. The exit point is slightly moved forward along the ray path to ensure that the next point-location search is in the next leaf and then the ray-traversal algorithm is called recursively. This recursion is terminated when a hit point is found or when the ray is out of the scene.
If the ray origin is out of the tree, we must determine the entry to the tree along with the ray in the first step.

The figure 2 is an example search of the location of ray *r* in figure 1. The process starts at the root node of the tree. On every level we compare the origin of the ray with the position of the splitting plane. We choose the left or right child node by the comparison, and we go down the tree until we find a leaf node. In this leaf node will be the location of the ray. We test all the objects in this node, and if there is an intersection point (in the scene in figure 1 the ray *r* will intersect the black sphere) we finish the searching process. If there is no intersection we go to the next leaf, which is selected by the exit point of the ray from the node.

2.4 Using the SSE extensions

Many modern processors have a SIMD[9] unit to accelerate the computing with large data set. This units supports the calculations with large data. In the Intel Pentium 4 [1] or AthlonXP [2] processor there are eight independent 128 bits wide registers that can be used with the SSE[10] instruction set [4]. In these registers we can store multiple float variables packed and we can do the same operation on each element.
With the SIMD instructions we can accelerate the commonly used vector operations, such as dot products, cross products, normalization and addition. We used the builtin intrinsic functions of the compiler to call SSE operations. The *float* vector variables was defined with

```
typedef float v4sf __attribute__ ((mode(V4SF)))
```

type. This type represents a vector with four *float* variables. We used an unnamed *union* to get the components of the vector easily.

```
union {
        float fmember[3];
        struct { float a,b,c,d; };
        v4sf vector;
}
```

With this structure we could easily convert the common vector operations to use the SSE instruction set. For example the addition of two vectors with common math operations is

[9] Single Instruction on Multiple Data
[10] Streaming SIMD Extensions

```
vector4 operator + (vector4 v1, vector4 v2) {
        return vector4( v1.a + v2.a,
                        v1.b + v2.b,
                        v1.c + v2.c,
                        v1.d + v2.d);
}
```

This function is contains four addition, with SSE it use only one vector addition.

```
vector4 operator + (vector4 v1, vector4 v2) {
        return vector4(__builtin_ia32_addps(v1.vector,
                                            v2.vector));
}
```

Another good example is the vector normalization. The following pure C++ code requires 4 multiplication, 4 addition, 4 division and a square root calculation.

```
vector4 normalize(vector4 v) {
float sq = sqrt(v.a*v.a+v.b*v.b+v.c*v.c+v.d*v.d);
return vector4(v.a/sq, v.b/sq, v.c/sq, v.d/sq);
}
```

The SSE optimized version of the normalization requires 2 multiplication, 3 addition, 3 shuffling operation and a reciprocal square root calculation.

```
const int p1 = _MM_SHUFFLE(0,3,2,1);
const int p2 = _MM_SHUFFLE(1,0,3,2);
const int p3 = _MM_SHUFFLE(2,1,0,3);

vector4 normalize(vector4 v) {
v4sf sq, tmp;

sq = __builtin_ia32_mulps(v.vector, v.vector);
tmp = __builtin_ia32_addps(sq,
                    __builtin_ia32_shufps(sq, sq, p1));
tmp = __builtin_ia32_addps(tmp,
                    __builtin_ia32_shufps(sq, sq, p2));
tmp = __builtin_ia32_addps(tmp,
                    __builtin_ia32_shufps(sq, sq, p3));
tmp = __builtin_ia32_rsqrtsps(tmp);
return vector4(__builtin_ia32_mulps(v.vector, tmp));
}
```

This version of the calculation is more than two times faster than the pure C++ code. With proper data alignment this function is speeded up about five percent.
From these examples we can conclude, that SIMD extensions are very useful to optimize the speed of the basic calculations of the ray-tracer.

2.4.1 Data alignment

The efficiency of the vector processing units depends on the speed of the data access [3]. A data is accessed most efficiently if it stored at a memory address which is divisible by the size of the data. To use the SIMD floating point operations the in-memory operands must be aligned at 128 bytes boundary, otherwise we must use the costly data load operations to move the data to the vector register. The penalty is at least 2-3 clock cycles if the data is not in the cache and not properly aligned. If the data crosses the 32 byte boundary the penalty is much higher.
To eliminate the data access latency the variables used must be stored on proper memory regions. If the compiler does not give the ability to control the alignment of the allocated variables, it is a good choice to use custom memory allocation functions which allocate a large continuous chunk of memory and divide it to 32 bytes long regions.
We used the built in functions of the compiler to align the variables.

```
float x __attribute__ ((aligned (128)))
```

This declaration will align *float* variable *x* to 32 bytes boundary.

2.4.2 Cache coherency

The cache is a small but high speed memory closer to the processor than the main memory. It is used to store frequently used data, which can be accessed much faster. Modern processors have two or three cache levels. The level-1 data cache in a P4 or AMD AthlonXP processor can contain 8 kbyte of data organized as 128 cache lines of 64 bytes each [3]. The cache lines are aligned to physical addresses divisible the cache line size and in a cache line the system can store data from particular region of the main memory only.
We can take advantage of the cache only when the frequently used variables are properly aligned and arranged in 64 byte blocks at 64 byte boundary.

3 More speed

To improve the performance of the ray-tracer the key parts are the spatial subdivision algorithm and the improved hit calculation with the SIMD instruction set. But there are several ways to get a smaller speedup. These methods are raise the overall performance by around ten percent.

3.1 Importance sampling

A recursive ray-tracer spawns new rays at every intersection point to determine the contributions of the light sources and other objects. This results in lot of rays that must be traced. The amount of the secondary ray's contributions is based on the scene setup and the attributes of the materials. Diffuse materials need far less secondary rays than the shiny, reflective surfaces.
The depth level of a secondary ray can be limited too. If the contribution of a secondary ray is negligible we can stop the ray spawning process before the depth reaches the maximum.
With these limitations we can reduce the count of the traced rays with a small decrease of quality.
In figure 3 the rays belong to white pixels are terminated before they reached the limit of the ray-tracing depth.

3.2 Supersampling

A common way to improve the calculated picture's quality is supersampling. This means that the ray-tracer gets more

Figure 3: Importance sampling

samples through a pixel and calculates the average of them to get the final color. This results in an anti-aliased picture, but the supersampling process virtually enlarges the picture size, which requires more rays.
Supersampling is more important at the edges of objects. If supersampling is used only at the edges, overall quality does not reduce much, but the speed gain is huge. In 4 the white pixels are represent the region wherein we used supersampling.
To detect object switch we maintain a list of the objects that were visible in the previous line and a variable which stores the object of the previous pixel. With these two variables we can determine the object switching both in horizontal and vertical direction. If there was a switch we spawn more primary rays in that pixel to avoid aliasing. The origins of the primary rays are modified with a small random number to take advantage of the multiple samples.

Figure 4: Region of interest in supersampling

3.3 Region based rendering

To improve the cache behavior we have used region based rendering. This means the pixel space is divided into small rectangular regions and each of these regions are rendered independently after each other. The rays that traced in a region are much more likely use the same data, they have more chance to hit the same objects in the scene. The optimal region size depends on the density of the objects in the scene, but in average case an 64×64 size does not trash the cache but large enough to compensate the cost of the administration of the regions.
This rendering method is roughly ten percent faster than the scanline based.

4 Results and conclusion

The ray-tracer was tested on a system equipped with a AMD AthlonXP+ 2500 processor with 1 Gbyte ram. The operating system was Linux with X.org. We compiled our program with the GNU GCC v3.4. To measure performance we used test scenes with 10, 100, 1000 and 50000 spheres. The material of the spheres had diffuse, specular, reflective and refractive components. All of the measurements were with 640×480 picture size with $32bits$ color depth.

Figure 5: Example scene

Number of objects	10	1000	50000
Naive	2.2s	161.5s	-
SSE Math	1.1	142.4s	-
KD-tree+SSE	0.1s	3.5s	6.3s
KD-tree+SSE+Region	0.1s	3.1s	5.8s

Table 1: Rendering times

Our tests showed that the most important optimization technique is the spatial subdivision. Using a KD-tree the rendering process is more than thousand times faster than the naive implementation in large scenes. With SSE vector operations we can reach another ten percent performance gain. See in table 1.

The region based rendering is useful to improve the cache coherency. On our test machine the 64×64 regions are the best. With larger region sizes the cache can not be utilized properly. See in table 2.

n×n region	1	16	32	64	128
time	1.78s	1.69s	1.45s	1.30s	1.5s

Table 2: Region based rendering

Method	C math	SSE	SSE + Alignment
time	409μs	249μs	221μs

Table 3: 50M vector multiplication times

The simple vector multiplication operator is used frequently during the ray-tracing process, for that we measured the performance gap between the naive and the SSE optimized version. As we expected, the SIMD version is almost twice as faster. See in table 3.

Method	C math	SSE	SSE + Alignment
time	4.6s	1.6s	1.4s

Table 4: 50M vector normalization times

Our another SIMD operation example was the vector normalization. The speed up of the SSE optimized version is more than thousand percent. See in table 4.

Our goal was to implement a recursive ray-tracing program to examine of the various optimizations opportunities. With the presented methods we could reach almost interactive performance, and there are place for more optimization too.

5 Future work

There are several ways to further improve performance. It is a plausible possibility to convert more parts of the renderer to use SIMD extensions. If we can trace a bundle of rays in parallel with the SIMD method, theoretically we can achieve 200-300% speedup. Another way is to make the rendering system distributed. The algorithms used can easily be converted to utilize the improved performance of a clustered computer system, but there are some serious implementation problems with the nature of distributed systems. The Graphics Processing Units (GPU's) that exist in modern computers provide an interesting potential too. This opportunity requires a large amount of modification of the used algorithms, but it has great future.

References

[1] Intel Corporation. *IA-32 Intel Architecture Software Developre's Manual*. Intel Corporation, 2004. Available at http://developer.intel.com/design/Pentium4/.

[2] Advanced Micro Devices. *AMD Athlon Processor - x86 Code Optimization Guide*. Advanced Micro Devices, 2002. Available at http://www.amd.com/us-en/Processors/TechnicalResources/.

[3] Agner Fog. *How to optimize for the Pentium family of microprocessors*. 2004. Available at http://www.agner.org/assem.

[4] Richard Gerber. *The Software Optimization Cookbook*. Intel Press, 2002.

[5] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[6] Laszlo Szecsi. *An effective implementation of the k-D tree*, pages 315–326. Charles River Media, Inc., 2003.

[7] L. Szirmay-Kalos, V. Havran, B. Benedek, and L. Szécsi. On the efficiency of ray-shooting acceleration schemes. In *Proc. Spring Conference on Computer Graphics (SCCG '2002)*, pages 97–106. Comenius University Press, 2002.

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

[9] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.