IMPLEMENTING INTERACTIVE RAYTRACER

Tomáš Marek, Marek Vavruša, Vojtěch Vítek
Bachelor Degree Programme, FIT BUT
E-mail: {xmarek11, xvavru00, xvitek19}@stud.fit.vutbr.cz

Supervised by: Filip Orság
E-mail: orsag@fit.vutbr.cz

ABSTRACT

With increasing hardware resources, interactive ray tracing has become a reality.
However, implementation is not trivial because of numerous different approaches and platform-specific problems. The goal of this paper is to review state-of-the-art ray tracing algorithms and discuss common performance bottlenecks. As a result we present fast raytracer implementation on CPU.

1 INTRODUCTION

Recent research [Wal04] proved the possibility of interactive ray tracing by efficiently using resources provided by modern CPUs [HKRS02]. This approach includes usage of SIMD instruction sets and low-level optimizations, which leads to portability issues. We chose IA-32 as target architecture and designed acceleration libraries for exploiting SSE2 and SSE3 instruction sets.

2 RENDERER DESIGN

We implemented Whitted-style raytracer [Wh80] with iterative main loop and rendering directly to framebuffer for better performance. Whole projection plane is divided to rectangular tiles and queued for processing. Tiles in queue are processed by thread pool in parallel. This architecture takes full advantage of multi-core and multi-processor systems, increasing performance by approx. 70% per extra core. We also implemented full-scene antialiasing by casting square of rays and averaging final color.

3 RAY CASTING

General idea is to cast rays from camera through each pixel on projection plane. According to implemented perspective camera model, ray direction needs to be calculated for each pixel, but enables us to move and rotate viewport. While casted ray intersects any object, it's color is added to ray color according to used shading model. Ray origin and direction is recalculated on each hit. Object collisions are the most expensive calculations in our raytracer, thus we optimized vector operations and object collision by using inline SSE2/3 code. This increased performance by approx. 30%.

3.1 ACCELERATION STRUCTURES

Advanced binary space partitioning schemes subdivide space by using arbitrary splitting planes, cutting out empty volumes, thus resolving performance issues of regular structures. However, exact algorithm for split plane positioning is not known and several heuristics are being used instead. *Surface Area Heuristic (SAH)* yields best results on most scenes. It is based on probabilistic chance, that ray hits the volume is related to volume surface and approximated cost of traversal.

3.2 K-DIMENSIONAL TREES

K-dimensional tree *(kd-tree)* is an advanced binary space partitioning scheme, derived from BSP tree. General idea is that every non-leaf node generates split plane, that adaptively subdivides the node in two halves. Unlike regular structures, split plane position is not fixed but it's always aligned to one of the coordinate axes. In fact, it's position determines efficient tree from poor one, so we chose *Surface Area Heuristic*, which yields the best possible performance in most scenes [Hav01].

However, memory management during hierarchy construction is problematic, because final node count is hard to estimate. We resolved this by allocating continuous memory block with estimated number of nodes, thus improving cache performance at the cost of larger memory footprint. We chose *kd-tree* as our acceleration structure, because it's generaly best scheme for static scenes.

3.3 REFLECTION MODEL

We used *Blinn - Phong* reflection model, because it's fast and more accurate version of *Phong* shading model, being state-of-the-art in modern renderers *(OpenGL, Direct X)*. It takes into account **ambient**, **diffuse** and **specular** color component of the material, thus provides accurate approximation.

3.4 REFRACTION MODEL

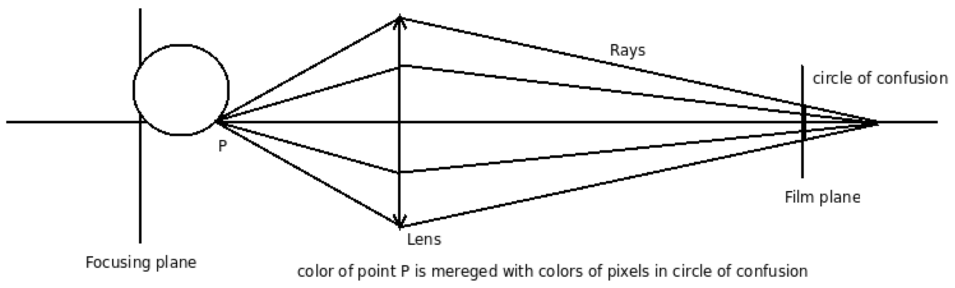
We also implemented refraction model, described by *Snell's law*, which states that the angle of incidence is related to the angle of refraction by well-known formula $\frac{\sin\alpha_1}{\sin\alpha_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1}$. Considering speed and negligible effect on the result, we decided to restrict number of reflections and refractions to definite value.

Both reflection and refraction models couldn't be implemented without absorption model. Every object in the scene has it's material specification, which contains **reflective ratio**, **refractive index** and **absorption ratio**.

4 POST-PROCESSING

Post-processing is the last operation in rendering chain and operates on rasterized image. It is very common in computer graphics to enhance perception of inaccurate shading models and often hardware accelerated by pixel shaders technology. With efficient methods we are able to implement many effects in single pass. We chose *depth of field (DOF)* effect, because it enhances image sense of depth to photorealistic level.

Depth of field is an effect where objects within certain range appear to be in focus but objects out of this range appear blurry. We chose *circle of confusion* [MJL00] algorithm, because it's moderately fast for interactive ray tracing in comparison with simulating real lens. Ray tracer generates image along with *depth map*, in which we store distance from lens to first intersection, thus we are able to calculate *circle of confusion (CoC)*.


Rays
circle of confusion
P
Film plane
Lens
Focusing plane
color of point P is mereged with colors of pixels in circle of confusion

Circle of confusion is an area, that determines the strength of blur in final picture.

5 CONCLUSION

As a result, we created a platform and API for further development, which implements several advanced features. Although we did not achieve the performance of *real-time* ray tracer on CPU only, we are able to render moderately complex scenes at interactive rate *(see Table 1)*.

Scene	Objects	Lights	Elapsed time	Casted rays
RGB spheres	3	3	150ms	0.340M
Ray gems	1680	3	370ms	0.349M
Ray gems + Floor	1680	3	1260ms	0.642M
Susan	15988	3	500ms	0.364M
Susan + Floor	15989	3	1500ms	0.657M
Susan + 4x AA	15988	3	1800ms	1.456M

Table 1: Raytracer performance on Intel T7250 (2GHz) CPU.

Our future research will focus on ray packet casting, which makes extensive use of SIMD instruction sets and paralell ray tracing on a computer clusters, which will bring even better scalability and performance.

REFERENCES

[Hav01] Vlastimil Havran: Heuristic Ray Shooting Algorithms, Czech Technical University in Prague 2001, http://www.cgg.cvut.cz/ havran/phdthesis.html

[HKRS02] Jim Hurley, Alexander Kapustin, Alexander Reshetov, Alexei Soupikov: Fast Ray Tracing for Modern General Purpose CPU, Intel 2002

[MJL00] Mulder, Jurriaan, and Robert van Liere: Fast Perception-Based Depth of Field Rendering 2000, http://www.cwi.nl/ robertl/papers/2000/vrst/paper.pdf

[Wal04] Ingo Wald: Realtime Ray Tracing and Interactive Global Illumination, Saarland University 2004, http://www.sci.utah.edu/ wald/PhD/

[Wh80] Turner Whitted: An improved illumination model for shaded display, June 1980, Communications of the ACM, 23(6):343-349