

# Offloading ray processing onto the GPU using cooperative worker threads

Stephan Reiter\*

## ABSTRACT

The processing power and flexibility of recent rasterization hardware permits to speed up image generation through raytracing. This paper introduces a method that performs ray processing on the GPU in parallel to shading operations on the CPU. It effectively exploits CPU-GPU parallelism in a way that can be integrated into existing implementations utilizing the standard recursive formulation of the problem.

**CR Categories:** I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

**Keywords:** Cooperative threading, GPGPU, hardware acceleration, ray batching, ray tracing

The latest releases of graphics hardware offer new concepts, such as branching, looping and higher instruction count limits for shader programs, into the rasterization world that are crucial for using the GPU as a general purpose processor [3]. Yet, using the GPU as a coprocessor usually requires substantial modifications to traditional, CPU-only code.

Many standard implementations of raytracers employ the concept of materials for the shading of objects: The color of a given ray is computed by invoking the material program of the respective object, which has been hit by the ray. In addition, secondary rays can be traced for advanced effects such as shadows, reflections, or refractions [2]. This direct implementation of the recursive formulation of the raytracing algorithm is incompatible with GPU-based acceleration techniques, because only a single ray is processed at a given time and the involved trace-function is expected to return immediately. In order to use hardware accelerated raytracing, where batches of rays are processed on the GPU [1], a way has to be found to separate shading operations from ray processing.

To solve this problem, we propose the following software architecture: Each of the primary rays is processed in parallel by a set of cooperative threads. Upon calling a trace-function, the ray is inserted into the processing queue and the current thread is put on hold by passing control to the next "worker" in the set of threads. When the size of the ray queue exceeds a given threshold, a batch is issued to the GPU, which begins processing the contained rays asynchronously to the CPU and delivers the results in a texture. Once all cooperative threads are in waiting mode, the available results are retrieved from the GPU and the affected threads are woken up, where control flow resumes after the call to the trace-function. This way trace-calls still behave like in the traditional implementation, but multiple rays can be processed in parallel on the GPU. Therefore no changes to material programs are necessary, although additional speedup may be gained by passing an array of rays to the trace-function at once, e.g. for smooth shadows.

The poster will present an overview of the approach together with results in the form of pictures and benchmarks of a proof-of-concept implementation. It will also deliver images to explain and illustrate the control flow in detail for a sample scene.

## REFERENCES

- [1] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.
- [2] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989.
- [3] Aaron Lefohn, Ian Buck, John D. Owens, and Robert Strzodka. Gpgpu: General-purpose computation on graphics processors. In *Tutorial #3 at IEEE Visualization*, October 2004.

---

\*e-mail: stephan.reiter@students.jku.at