Ray Tracing Algorithms

John Stone
Theoretical and Computational Biophysics Group, Beckman, UIUC

Introduction

Ray tracing is a popular rendering technique known for its ability to simulate reflection, refraction, dispersion, depth of field, and many other phenomena with relatively simple code. Since ray tracing involves point sampling geometry, curved surfaces look smooth at all scales, and it can easily be extended to render volumetric datasets. The ray tracing algorithm easily incorporates shadows from direct lighting, and can be extended to support area lights, ambient occlusion lighting, or may be used in conjunction with photon mapping for global illumination effects. The flexibility of the ray tracing algorithm is not free however, and comes at a great computational cost. This cost is addressed using various spatial acceleration data structures that eliminate unnecessary work. Further performance increases are typically achieved using SIMD vectorization and parallel processing techniques. Ray tracing is a very popular method for the creation of high quality renderings for molecular visualization, and will likely remain so for some time to come.

Description

As GPUs have become progressively more powerful and general purpose, the graphics community has explored GPU based ray tracing algorithms. CUDA and the G80 architecture provide several advances not available to previous GPU-based ray tracing implementations. The branchy and incoherent nature of the ray tracing process presents several challenges to the creation of an efficient GPU implementation. CPU-based ray tracers typically make heavy use of recursion, which must be replaced by iteration on current GPU hardware. The ability for CUDA kernels to read and write shared and global memory, and to perform barrier synchronizations provides greatly increased flexibility in implementing GPU ray tracing relative to previous work by Purcell, Carr, and others. This flexibility could allow stacks, queues, and other data structures to be implemented on the GPU, reducing dependence on the host CPU to manage low-level rendering state.

The main work performed by a ray tracing kernel is the computation of intersections between rays and scene geometry. In a typical ray tracer, the process begins with the generation of eye rays which determine the visibility of objects in the scene, and the process continues as lighting and texturing operations generate further shadow rays, reflection rays, and refraction rays on-demand as necessary to completely render the scene. Ray tracing kernels often randomly access scene geometry as intersection tests are performed. This is problematic even in CPU-based implementations, but particularly for GPUs, which do not have large cache memories. One possible way of ameliorating this problem in a CUDA ray tracing kernel might be to make extensive use of its explicitly exposed memory architecture and the latency hiding capabilities of the G80 hardware.

Objective

Since intersection calculation is typically the rate-limiting component of the ray tracing process, it is an interesting target for GPU acceleration.   This project will require the implementation of a basic ray tracing kernel on the GPU. For the sake of simplicity, the implementation will include one, or at most two geometric primitive types, e.g. spheres or triangles, and an extremely simple lighting model. If time allows, the project team could also try implementing one of the well known spatial acceleration technique such as uniform grids, k-D tree, bounding interval hierarchy, etc. The main goal of the project will be to measure the effects of divergent branching, incoherent memory accesses, and latency hiding on ray tracing performance, and to compare ray tracing on the G80 architecture with previous generation designs used by Purcell, Carr, and other published GPU ray tracers. The project team should read the published literature carefully before embarking on the project.

Background

Some background in computer graphics will be very helpful. Although the ray tracing algorithm is conceptually simple, measuring the interactions between ray tracing algorithms and hardware will likely require some prior ray tracing implementation experience or a willingness to spend extra time on the project. Programming expertise is assumed.

Resources

The project mentor will provide the team with various ray tracing sample source code, test scenes, and implementation consultation as-needed.

Algorithm references, papers, and sample code will be made available online:
http://www.ks.uiuc.edu/Research/vmd/projects/ece498/

Contact Information

John Stone: johns@ks.uiuc.edu