Project Proposal - EEC 277
Accelerating Ray Tracing on The Graphics Pipeline

Yuxin Chen and Ahmed H. Mahmoud

21 February 2017

1 Abstract

We propose to explore few challenges regarding implementation of ray tracing through the graphic pipeline using OpenGL fragment shader. Using efficient data structure is the natural choice to think of when implementing a CPU-based ray tracing due to well-developed implementation of these data structure on CPU and virtually unlimited amount of memory available. It is expected that that mapping such data structure into fragment shader will have an overhead due to the nature of the graphics pipeline. In this work, we would like to explore the challenges of implementing ray tracing on GPU in terms of which data structure is more efficient, how the necessary elimination of recursion calls would affect the performance and comparison with CUDA-based ray tracing where greater flexibility is available.

2 Motivation

1. Why GPU?
   In the last decade, the performance of GPUs has increased more rapidly than that of CPUs and most of current GPUs incorporate floating points units even double float points units, making GPUs a computing beast for parallelable computation. As the GPU evolves to include

more instruction set and supports larger data types operations, it becomes a more and more general-purpose streaming processor. The programmable stages of graphics pipeline are enhanced and allow more flexibility and longer program that basic ray tracing components can be loaded as a single program. So the convergence of the three separate trends: sufficient raw performance for single-chip real-time ray tracing; increasing GPU programmability; and faster performance improvements on GPUs than CPUs make GPUs an attractive platform for real-time ray tracing.

2. Why using graphics pipeline?
   Ray tracing can be easily formulated into a streaming of computations which is easily mapped into a streaming processor. When we say streaming computing, it differs from traditional computing in that the system reads a sequential stream of data and execute a kernel on each element of the input stream of data. In this sense, a programmable graphics processor executing a vertex program on a stream of vertices, and a fragment program on a stream of fragments is a streaming fragment processor.The streaming model of computation leads to efficient implementations on streaming processors for three reasons: 1) process the data in parallel since there is no dependence between data, 2) kernels achieve high arithmetic intensity, 3) streaming hardware can hide the memory latency of texture fetches by using prefetching [1]. So in this work, we are going to break the ray tracing into several kernels, chain those kernels with streams of data and map they onto streaming processor.

3. Elimination of control flow and recursion
   Ray tracing is usually used in a recursive manner. To compute the color of primary rays, recursive ray tracing algorithm casts additional, secondary rays creating indirect effects like shadows, reflection or refraction. It is possible to eliminate the need for recursion and to write the ray tracer in an iterative way which runs faster since we don't need function calls and the stack. Note that the graphics pipeline doesn't include data-dependent conditional branching in its instruction set. To overcome this limitation, conditionals can be mapped to the hardware architecture by using the multipass rendering technique by [2]: the conditional predicate is first evaluated using rendering passes, and then a

stencil but is set to true or false depending on the result. The body of the conditional is then evaluated using additional rendering passes, but values are only written to the framebuffer if the corresponding fragment's stencil bit is true. It is quite efficient to use multipass rendering using large fragment programs under the control of the stencil buffer.

4. Spatial data structure for acceleration
   In ray tracing process, finding the closest object hit by a ray requiring to check a set of objects but only take the closest one. A brute force approach that checks all the object with a ray is too expensive. Therefore, to accelerate this process, we can only check a subset of objects by using spatial data structure. Many spatial data structures are available such as BSP trees, KD-trees, octrees, uniform grids, adaptive grids, etc.

5. OpenGL and CUDA are two programming model on GPU, and they both support streaming computing. Although CUDA support branching and recursion on the hardware level, as we discuss before, control flow and an alternative of recursion can also be implemented efficiently on rasterization pipeline. This raise the questions of which programming model is more efficient and suitable to implement ray tracing; and for a certain model, what configuration gives superior performance.

3 Related Work

Ray tracing was first introduced in 1980 [3] in order to introduce global illumination effects in the rendered scenes and thus more realistic effects are attainable. Since then extensive research has been done in order simulate more effects like reflection from glossy surfaces [4, 5], reflective [6], anti-aliasing [7, 8, 4], motion blur[4, 9], depth of field [10], etc.

In parallel, substantial amount of research has been devoted to accelerate the process of ray tracing in order to include ray tracing in interactive applications to, for example, introduce more realism in computer games. One way to accelerate the ray racing is by utilizing efficient data structure. Without such data structure, every ray would be tested against all objects in the scene making the complexity linear with the number of the primitives in the scene.

The accelerated data structure are used in order to subdivided the scene space. There has been two ways for subdivision: spatial subdivision and

object subdivision [8]. The spatial subdivision relies on subdividing the space into sub-regions and storing primitives in each sub-region. If a ray passes through a sub-region (which is less expensive to compute), only primitives in this sub-region are checked against. One the simplest techniques for spatial subdivision is the uniform or adaptive grid where each primitives is registered into one of the grid cells [11]. Then only primitives in a certain cell are tested if a ray hits that cell. Better performance can be gained by using tree structure instead (BSP trees, kd-trees, octrees) especially for scenes with nonuniform distributions of geometry and/or with arbitrarily moving objects especially [12]. Object subdivision breaks down the objects in a scene into smaller constituent objects which makes it easier to cull the parts that does not hit a ray. Objects here are made of primitives. Each major object contains minor objects. Thus, we can build a tree for the major objects of minor objects [13].

Evaluating the performance of two or more different graphics APIs has been studied before in order to understand the performance limitation. Comparison between OpenGL pipline, CUDA and OpenCL has been done for various applications such as Monte Carlo simulation of quantum spin system [14]; sorting algorithms [15]; Sobel filter, Gaussian filter, and Median filter [16]; volume ray casting [17]; and others [18]. The conclusion of all these studies could be there can not be a certain API that is the best for all even though they all share similar capabilities and functionalities.

4 Target

The target of our project can be summed in the following milestone

1. Implementing of ray tracing on fragment shader. (1 Week)
2. Implementing uniform gird, kd-tree to accelerate the performance
3. Experimenting with complex scenes. (1 Week)
4. In parallel, implementation on CUDA for final comparison between OpenGL-based and CUDA-based ray tracing.
5. Report and presentation. (1 Week)

Deliverables We intend to deliver an explanation on why basic ray tracing using one API outperforms the other. So far, we expect that CUDA will have superior performance since so little overhead needed to carry on the computation. In contrast, OpenGL needs to go through all the pipeline stages till it reach the fragment stage. But it was observed that using OpenGL pipeline outperforms CUDA-based implementation for volume ray casting [17]. Additionally, we are going to report our experience with KD-tree implementation within fragment shader and recursion elimination with ray casting and how they impacts the run-time and the quality of final image respectively.

References

[1] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 703–712.

[2] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar, "Interactive multi-pass programmable shading," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 425–432.

[3] T. Whitted, "An improved illumination model for shaded display," in *ACM SIGGRAPH Computer Graphics*, vol. 23. ACM, June 1980, pp. 343–349.

[4] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," in *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3. ACM, 1984, pp. 137–145.

[5] G. J. Ward, F. M. Rubinstein, and R. D. Clear, "A ray tracing solution for diffuse interreflection," *ACM SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 85–92, August 1988.

[6] J. Amanatides, "Ray tracing with cones," in *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3. ACM, 1984, pp. 129–135.

[7] P. Shirley and R. K. Morley, *Realistic Ray Tracing*. AK Peters, Ltd., 2003.

[8] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann, 2016.

[9] R. L. Cook, "Stochastic sampling in computer graphics," *ACM Transactions on Graphics (TOG)*, vol. 5, no. 1, pp. 51–72, January 1986.

[10] M. Shinya, "Post-filtering for depth of field simulation with ray distribution buffer," in *Graphics Interface*. Canadian Information Processing Society, 1994, pp. 59–59.

[11] J. M. Snyder and A. H. Barr, "Ray tracing complex models containing surface tessellations," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 119–128, Aug. 1987. [Online]. Available: http://doi.acm.org/10.1145/37402.37417

[12] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2005, pp. 15–22.

[13] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 269–278, Aug. 1986. [Online]. Available: http://doi.acm.org/10.1145/15886.15916

[14] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *arXiv preprint arXiv:1005.2581v3*, 2011.

[15] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–10.

[16] C. L. Su, P. Y. Chen, C. C. Lan, L. S. Huang, and K. H. Wu, "Overview and comparison of opencl and cuda technology for gpgpu," in *2012 IEEE Asia Pacific Conference on Circuits and Systems*, Dec 2012, pp. 448–451.

[17] F. Sans and R. Carmona, "Volume ray casting using different gpu based parallel apis," in *2016 XLII Latin American Computing Conference (CLEI)*, October 2016, pp. 1–11.

[18] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of opencl programs," in *The fifth international workshop on automatic performance tuning*, vol. 66, 2010, p. 1.