Implementing and Analyzing a GPU Ray Tracer

Kristóf Ralovich*

Budapest University of Technology and Economics

Abstract

In this paper we discuss the implementation of a GPU ray tracer. Our ray tracer is inspired by Purcell's recursive GPU ray tracer using regular grid space subdivision and is improved with "proximity cloud" information. This kind of ray tracer implementation is capable of rendering static triangular meshes with dynamic camera and dynamic abstract point light sources. Having presented the implementation and our experiences gained during the implementation process, the performance and scalability of the algorithm are also considered. Scalability is defined in terms of the number of objects in the scene and the resolution of the screen.

Keywords: ray tracing, GPU, uniform grid

1 Introduction

Nowadays the parallel architecture of commodity graphics hardware (GPU) has evolved far beyond just accelerating rasterization, but it is capable of posing a threat on comparable software implementations in various assorted computing problems. Such a problem is ray shooting, one core concept of ray tracing. Ray shooting basically constitutes of finding the closest intersection along a ray spanning through a scene.

Ray tracing is a way of creating computer generated images usually accounted for superior visual quality. Producing realistic images is achieved by mimicking the nature by modeling the bounces of the rays of light. On the other hand, this quality is not coming for free and we are facing the major drawback of ray tracing: its high computational requirement is cutting it mostly out of widespread utilization in interactive graphics systems.

Achieving realtime image rendering performance requires relaxation of the problem either by approximated ray tracing or by accelerated ray tracing. Corresponding to Whitted [25] 95% of the rendering time is spent in finding the closest intersection of individual rays (however, this is not true anymore if quality shading models are used). So it is obvious that improving the ray shooting algorithm will likely to yield in overall speedup. Numerous attempts, such as (hierarchical) bounding volumes or spatial data structures have been invented to accelerate the process of ray tracing. Since finding the closest intersection of a ray does not influence the calculation of other rays, ray shooting is massively parallel task too.

Since the popularity and importance of triangular meshes outweigh other polygonal based and functional model representations in the field of realtime rendering, throughout this document the term object refers to triangles.

2 Related Work

The roots of recursive ray tracing are dating back to 1980 [25], while the first publication on ray intersection applicability to the GPU was done by Carr et al. [2], but their method still used the CPU for the rest of the rendering process.

Purcell reported a ray tracer implemented entirely on the GPU [16] making use of the uniform grid data structure. This is known to be the first ray shooting method using a spatial subdivision acceleration scheme on the GPU. Several attempts [4, 22] are known to exist sharing a common history of the stackless Purcell style streaming GPU ray tracing. For example, Karlsson and Ljungstedt [9] incorporated the "proximity cloud" improvement in grid traversing, an idea originally developed by Cohen and Sheffer [5] letting empty voxels being skipped.

Kd trees on the GPU were first introduced by Foley and Sugerman [6], Thrane and Simonsen mapped a threaded traversing approach of bounding volume hierarchies to the streaming ray tracer and compared it with the uniform grid and kd tree constructs [22]. While these techniques only enabled handling of static scenes and required often costly offline preprocessing, in 2006 Carr et al. [3] introduced ray tracing of dynamic meshes of static topology using geometry images, requiring careful parametrization.

Examination of ray shooting performance had been discussed in many papers.

Havran et al. [7, 8] had developed fair comparison methodology for ray shooting algorithms, and proposed

*kristof.ralovich@eestec.hu

testing procedures to find the best efficiency scheme.

Szirmay-Kalos et al. [21] investigated the asymptotic behavior and efficiency of different ray shooting acceleration schemes characterized by the terms of the expected number of ray-object intersections needed to identify the firstly intersected object, and the expected number of steps made on the space partitioning data structure. Introducing a simple probabilistic computational model of an infinite number of spheres of the same radius uniformly distributed in infinite large space, they concluded that both the expected number of intersections and the expected number of visited voxels are constant (does not depend on the number of objects) yielding their complexity to be in $O(1)$.

3 Streaming ray tracing

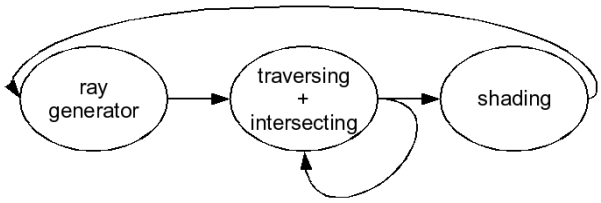

Figure 1: Recursive ray tracing decomposed to a series of computation kernels. Kernels are realized with GPU fragment shader programs.

This sections gives a brief overview of the Purcell style streaming ray tracing, from the point of view of the implementation used for testing.

As stated before uniform grid and proximity cloud information is used for accelerating ray shooting. Since only static scenes are used, these data structures need to be precomputed offline on the main processor.

In order to apply to the GPU, recursive ray tracing must be decomposed into computational stages (in the stream computing notation: *kernel*s) realized by fragment programs. Since the GPU does not feature a stack, all state information that would have been stored on the stack between recursive calls must be accommodated in memory (textures). Redirecting the results of a fragment program to framebuffer attachable images is called using *multiple render target*s [14].

The major computational stages are shown in figure 1 and are implemented by separate fragment programs. Fragment shaders operate on individual pixels, meaning rays are represented by fragments. The input stream and execution of all the fragment programs is initialized in the same way, by rasterizing a viewport sized quadrilateral. Furthermore this makes it possible use interpolated texture coordinates to identify rays.

The first kernel is responsible for generating either primary, shadow or reflection rays. Rays are characterized by their origin and direction, this output stream of data is written into two textures.

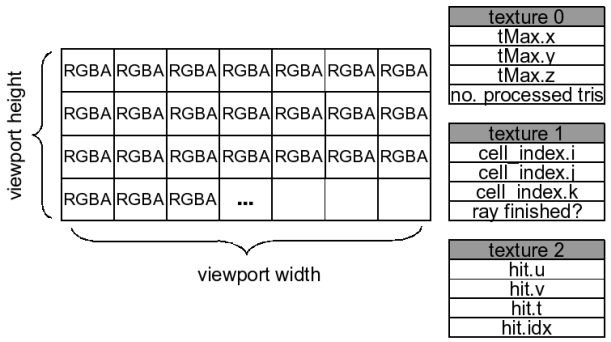

Figure 2: State information storage layout in textures. Traversal state is travelling in textures between successive calls of the traverse and intersect kernel.

The next computational stage is dealing with finding the closest intersection along a ray shot through the scene. This implies traversing the grid until a voxel with a non empty reference list is encountered and then intersecting the ray with all the enlisted triangles. If no intersection was found, this process is repeated. The actual grid traversal is accomplished with the fast 3D-DDA algorithm as described by Amanatides and Woo [1].

Hardware used for measurement purposes raised two important limits on compiled fragment shaders: limited instruction count and limited number of iterations of a loop. Satisfying these constraint, this kernel is designed not to be too long and must be called multiple times. Regarding the number of kernel calls depend on runtime parameters (eg. camera position), instead of constructing a worst-case heuristic limit of this count, an occlusion query [14] is providing the halting condition – in other words, we loop this kernel until no pixels (rays) are updated.

As shown on figure 2, traversal state between successive calls of this kernel is stored in three textures. The state includes $tMax$ vector (for the 3D-DDA algorithm) and ray state, current cell indices and the number of triangles processed, and hit record (barycentric triangle coordinates (u,v), ray parameter t and triangle ID) of the closest intersection so far.

In order to enable the GPU accessing the scene data, texture memory is employed in a read only manner. The applied memory layout is shown in figure 3. Using a uniform space subdivision scheme requires storing the list of referenced objects for each voxel of the grid. This is stored in a RGB 3D texture, where each grid cell is encoded in one texel as follows: the R component is a pointer to the first referenced object (in the voxel) in the list of objects, G holds the number of the referenced objects while the

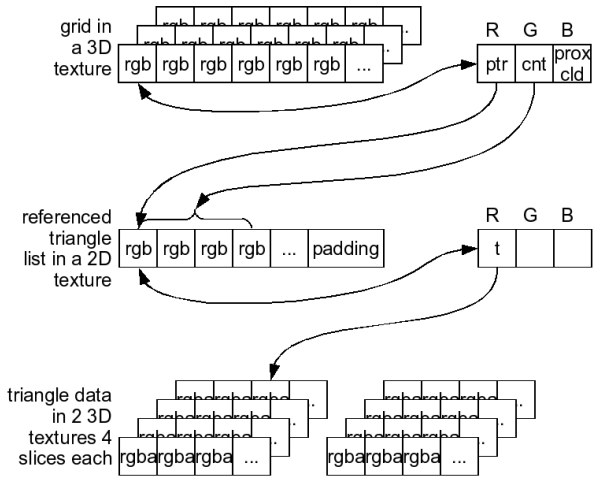

Figure 3: Grid data layout with two levels of indirection for storing the scene encoded in texture memory.

value in the third (B) channel gives the distance in voxels to the closest cell containing objects (for a non empty voxel, this distance is zero), this is used for skipping empty cells during traversing the grid ("proximity cloud"). The list of referenced objects is stored tightly packed and each texel corresponds to a pointer to actual triangle data. After two levels of indirection, actual triangle data is stored in two other 3D textures both featuring 4 depth slices. Texels with the same (X,Y) coordinates in different Z slices are storing vertex, normal and material information belonging to the same single triangle.

One expensive operation of ray shooting besides traversal is evaluating the ray-triangle intersection test. Möller and Trumbore [10] introduced the fastest known single sided algorithm working with barycentric triangle coordinates that is commonly used on the GPU.

The last kernel on figure 1 is shading. Due to focusing on ray shooting acceleration, this part of ray tracing is somewhat eclipsed. The implemented method is a simple per-pixel diffuse lighting with shadows and reflections.

Finally let us have the used optimizations enumerated at a glance:

- During the execution of an additional kernel (right after the ray generator, but suppressed in figure 1) rays that are known not needing further processing (eg. rays missed the scene AABB, shadow rays for rays not intersecting any objects) are "masked" out as soon as possible, and excluded from further calculations.

- Early Z culling can be exploited, to skip a costly fragment program (esp. traverse and intersect) being executed on rays already terminated, this requires and additional pass initializing the Z buffer with proper depth values prior to executing the kernel.

- In the case of shadow rays determining any (instead of closest) intersection closer than the given light source is sufficient.

- Instead of storing the first two vertices (v1 and v2) of a triangle, the corresponding edges (edge1 and edge2) are stored in order to save the GPU two vector subtractions every ray-triangle intersection test.

- Utilizing non power of two sized textures yields in the minimal usage of data padding.

4 Performance and evaluation

Assuring decent numerical precision during computation and for texture memory, 32-bit per channel floating point numbers are used. Evidently storage space might be a bottleneck for large scenes, so we have to look at this limitation.

Each grid cell is stored in a RBG texel occupying 12 bytes, the triangles vertices, normals and material properties are sparsely encoded in 8 texels resulting in 96 bytes. The total length of the triangle list depends on the distribution of objects among the grid cells, realistic values are supposed to be in the magnitude of 1 (average number of referenced triangles per voxel), this is meaning 4 bytes (1 component texel) on average per grid cells. So storing the scene data requires M bytes that may be expressed as:

$$M = n \cdot 96 + g_x \cdot g_y \cdot g_z \cdot (12+4)$$

where n is the number of triangles in the scene, while g_x, g_y, g_z are the dimensions of the grid.

The test system GPU is equipped with 256 MB of memory making possible to accommodate scenes with as many as 2.3 million triangles (using the heuristic of $g_{x,y,z} = \sqrt[3]{n}$). But this is a theoretical result since shader programs and traversing state requires additional space too.

The asynchronous operation of the GPU pipeline makes CPU timers unfeasible for accurate measuring of GPU computation time, but the GL_EXT_timer_query [14] OpenGL extension fills this gap by providing a 64-bit precision, nanosecond resolution timing information on the GPU. Throughout the measurements this timing mechanism is used. Parameters of the machine used for testing are as follows:

- GPU: Geforce 6800GT 400 MHz
- GPU memory: 256 MB 1100 MHz
- CPU: AthlonXP 3200+ 2200 MHz
- CPU memory: 512 MB 400 MHz
- OS: GNU/Linux 2.6.16.29
- GPU driver: NVIDIA-Linux-x86 1.0-9746

Knight scene

Figure 4: The Knight scene, image ray traced using the presented implementation (shadows and reflective material properties are enabled).

Torus Knot scene

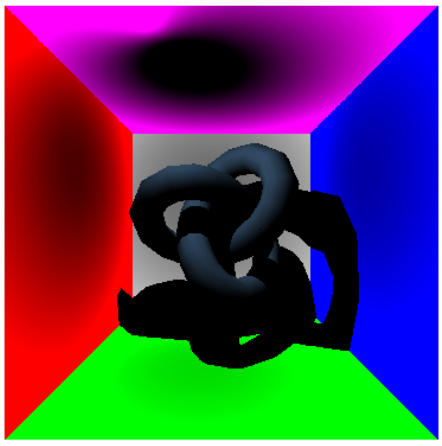

Figure 5: The Torus Knot scene. Image was ray traced via the presented implementation using diffuse material properties and shadows.

Bunny scene (including a simplified low polygon version of the Stanford Bunny model)

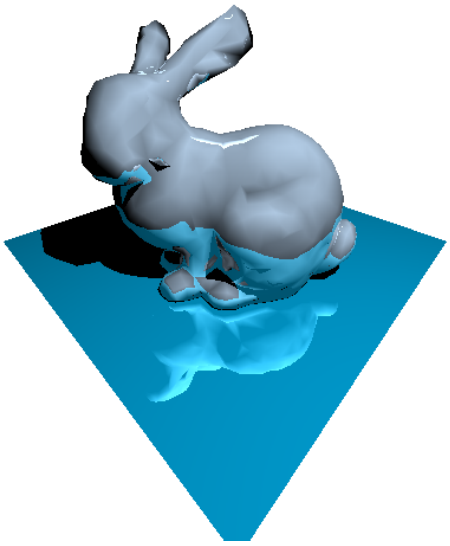

Figure 6: Image was ray traced using the presented implementation (shadows and reflective material properties are in use). Note the reflected ear on the back of the rabbit.

Stanford Bunny scene

Figure 7: The original Stanford Bunny model. Image is produced with the presented implementation using raycasting.

Figures 4, 5, 6 and 7 are visualizing the evaluated test scenes. Measurements were conducted using grid dimensions our experiments showed to be the most efficient: $14 \times 16 \times 14$, $16 \times 16 \times 16$, $20 \times 20 \times 20$ and $128 \times 128 \times 128$ for the four scenes respectively.

Taking in consideration that the deployed OS utilizes time sharing preemptive scheduling, all the presented measurement results are best values of multiple test runs.

The measurement results are summarized in tables 1, 2 and 3.

rendering resolution	Knight $n = 636$		Torus knot $n = 1024$		Bunny $n = 1764$		Stanford Bunny $n = 69451$
	g	pc	g	pc	g	pc	g
256×256	45	47	55	55	37	37	142
512×512	103	106	119	133	86	89	327
768×768	177	180	210	229	149	144	551

Table 1: Ray casting (eye rays only) time: summary of measurement results of different scenes with different rendering resolutions. Columns g and pc stands for uniform grid and uniform grid with proximity cloud information, respectively. Values are in milliseconds.

rendering resolution	Knight $n = 636$		Torus knot $n = 1024$		Bunny $n = 1764$		Stanford Bunny $n = 69451$
	g	pc	g	pc	g	pc	g
256×256	96	91	169	164	75	75	235
512×512	210	215	365	401	176	181	563
768×768	349	360	671	734	302	298	960

Table 2: Shadow casting (eye and shadow rays) time: summary of measurement results of different scenes with different rendering resolutions. Columns g and pc stands for uniform grid and uniform grid with proximity cloud information, respectively. Values are in milliseconds.

rendering resolution	Knight $n = 636$		Torus knot $n = 1024$		Bunny $n = 1764$		Stanford Bunny $n = 69451$
	g	pc	g	pc	g	pc	g
256×256	168	164	306	300	129	130	482
512×512	383	390	656	691	299	306	1243
768×768	630	643	1173	1244	505	501	2188

Table 3: Ray tracing (eye, shadow, and reflection rays) time: summary of measurement results of different scenes with different rendering resolutions. Columns g and pc stands for uniform grid and uniform grid with proximity cloud information, respectively. Values are in milliseconds.

5 Conclusion

Despite the Bunny scene is featuring higher number of triangles, than the Knight and Torus Knot scenes, it is clear from all the test results presented, that both of these scenes are rendering slower. According to Havran et al. [8], the uniform grid subdivision scheme performs well with scenes of uniformly distributed objects and considering the triangles are more uniformly distributed in space in case of the Bunny scene, the results are not contradicting any more.

In case of the Knight, Bunny and Stanford Bunny scenes the applied shadow ray optimizations yielded roughly 20% speedup compared to the naive implementation of shadow rays.The Torus Knot scene could not benefit of this, because all shadow rays are originated inside the box, thus their paths always intersect the walls.

During the ray tracing evaluation two levels of recursion (resulting in at most three rays per pixel) was used. The rendering times show that at this level the time spent on shading computations is not negligible.

Thrane and Simonsen [22] evaluated their GPU ray tracer using the Stanford Bunny model too (their hardware and setup approximates our ray casting configuration at the resolution of 512×512, using the uniform grid) and their results yields comparable performance.

Although Karlsson et al. [9] reported significantly shorter rendering times using the proximity cloud information on dedicated sparse scenes, our test scenes seem to be dense regarding this question and – except for a few setups with marginal performance gain – the measured results are almost always worse than not using proximity clouds.

Our conclusion is that although the GPU performs efficiently on the ray shooting problem, shading computations still provide a wide field for improvements.

6 Future work

As GPGPU computations are maturing, it would be desirable to conduct a full scale comparison of different ray shooting acceleration schemes (*RAS*) as had been done for software implementations [8].

With the introduction of GPUs capable of Shader Model 4.0, integer operations and geometry shaders are supported too. For example optimized RAS traversal and texture addressing might exploit faster integer operations.

Another interesting question is to fine-tune the way triangle data is stored in textures in order to achieve increased cache coherency during data fetch. This is supposed to improve the shading part of this ray tracing implementation.

References

[1] John Amanatides and Andrew Woo: A Fast Voxel Traversal Algorithm for Ray Tracing. *Eurographics '87*, pp 3-10, 1987.

[2] Nathan A. Carr, Jesse D. Hall, John C. Hart: The Ray Engine. *Graphics Hardware*, The Eurographics Association, pp 1-10, 2002.

[3] Nathan A. Carr, Jared Hoberock, Keenan Crane, John C. Hart: Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. *Proceedings of Graphics Interface*, pp 203-210, A K Peters, 2006.

[4] Martin Christen: Ray Tracing on GPU. Diploma Thesis, University of Applied Sciences Basel. 2005.

[5] Daniel Cohen and Zvi Sheffer: Proximity clouds — an acceleration technique for 3D grid traversal. *The Visual Computer*, Volume 11, pp 27-38, Springer-Verlag, 1994.

[6] Tim Foley and Jeremy Sugerman: KD-Tree Acceleration Structures for GPU Raytracer. *Graphics Hardware*, The Eurographics Association, 2005.

[7] Vlastimil Havran and Werner Purgathofer: Comparison Methodology for Ray Shooting Algorithms. Technical Report / TR-186-2-00-20, November 2000, 2000.

[8] Vlastimil Havran, Jan Prikryl, Werner Purgathofer: Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical Report / TR-186-2-00-14, May 2000, 2000.

[9] Filip Karlsson, Carl Johan Ljungstedt: Ray tracing fully implemented on programmable graphics hardware. Master's Thesis, Chalmers University of Technology, Göteborg, 2004.

[10] Möller, T. and Trumbore, B.: Fast, Minumum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*. vol. 2, no. 1, pp 21-28, 1997.

[11] NVIDIA Corporation: *GPU Programming Guide*. Version 2.4.0, 2005.

[12] NVIDIA Corporation: NVIDIA OpenGL Extension Specifications. November 8, 2006. 2006.

[13] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis: OpenGL Programming Guide Fifth Edition. Addison-Wesley, 2005.

[14] OpenGL Extension Registry:
GL_ARB_draw_buffers,
GL_ARB_occlusion_query,
GL_ARB_texture_float,
GL_EXT_framebuffer_object,
GL_EXT_timer_query extension specifications.
`http://www.opengl.org/registry`, 2003, 2004, 2005.

[15] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan : Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*. vol. 21, no. 6, pp 703-712, 2002.

[16] Timothy John Purcell: RAY TRACING ON A STREAM PROCESSOR. Ph. D. Dissertation, Stanford University, 2004.

[17] Randi J. Rost: OpenGL Shading Language, Second Edition. Addison Wesley Professional, 2006.

[18] Mark Segal, Kurt Akeley: The OpenGL Graphics System: A Specification. *Version 2.1 - December 1, 2006*, 2006.

[19] Sudhanshu K. Semwal and Hakan Kvanstrom: Directed safe zones and the dual extent algorithms for efficient grid traversal during ray tracing. *Proceedings of the conference on Graphics interface '97*, Canadian Information Processing Society, 1997.

[20] László Szirmay-Kalos, György Antal, Ferenc Csonka: Háromdimenziós grafika, animáció és játékfejlesztés. ComputerBooks, 2003.

[21] László Szirmay-Kalos, Vlastimil Havran, Benedek Balázs, László Szécsi: On the Efficency of Ray-shooting Acceleration Schemes. *Spring Conference of Computer Graphics*, pp. 89-98., Budmerice. 2002.

[22] Niels Thrane, Lars Ole Simonsen: A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis. 2005.

[23] Ingo Wald and Philipp Slusallek: State of the Art in Interactive Ray Tracing. *Eurographics State of the Art Reports*. 2003.

[24] Ingo Wald and Philipp Slusallek: Afrigraph Tutorial B: Interactive Ray-Tracing. *Afrigraph 2001 Course on "Interactive Ray Tracing"*. 2001.

[25] Turner Whitted: An improved illumination model for shaded display. *Communications of the ACM*. vol. 23, no. 6, pp 343-349, 1980.