Efficient Use of In-Game Ray-Tracing Techniques

Thales Luis Rodrigues Sabino
Paulo Andrade
Lucas Grassano Lattari
Universidade Federal Fluminense

Esteban Clua
Anselmo Montenegro
Universidade Federal Fluminense

Paulo Aristarco Pagliosa
Universidade Federal do Mato Grosso do Sul

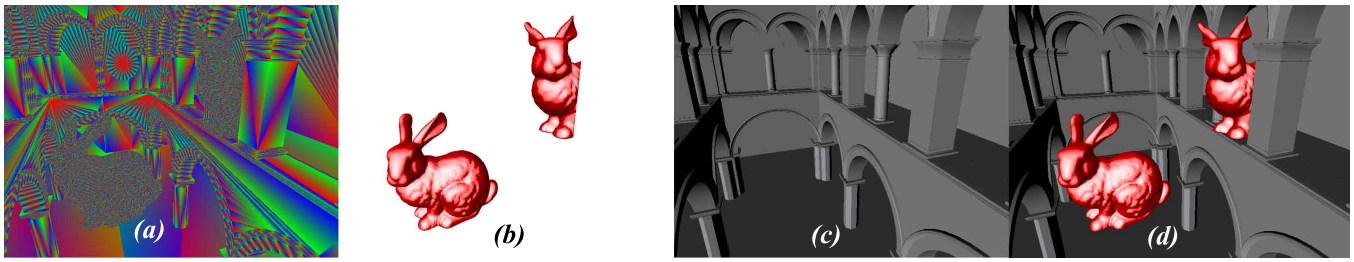

Figure 1: *Stages of our hybrid ray-tracing implementation. (a) shows a visual representation of primary rays intersections, (b) shows only the visible pixels of ray-traced objects. (c) shows the Sponza scene rasterized using a simple directional light by the using of the traditional graphics pipeline. Finally, (d) shows the final scene representation obtained by merging images (b) and (c). The Sponza scene has 44.404 vertices and 67.462 triangles. The Stanford Bunny has 34.834 vertices and 69.451 triangles. The complete scene has 114.072 vertices and 206.364 triangles. The total number of valid rays is 66.252 rendered at 10 frames per second.*

Abstract

Ray-tracing is a computational demanding image generation technique capable of create photo-realistic images. Due to its high demand of computational power, ray-tracing is not used for real-time applications, however, with the massive parallel capabilities of current Graphics Processing Units, and the fact that ray tracing is a suitable application for parallel processing, the use of GPU based Real-Time Ray-Tracing started to be seriously considered. This work investigates a GPU only approach for rendering images, where both raster and ray-tracing strategies are used in a hybrid approach, in order to improve visual quality while maintaining an interactive or near real-time performance.

Keywords:: Rendering, Ray-tracing, GPU, Real-time, Raster, OpenGL, GLSL, CUDA

Author's Contact:

{tsabino, pandrade, llattari, esteban, anselmo}@ic.uff.br
pagliosa@ufmt.br

1 Introduction

In the world of computer generated graphics, it is a common belief that raster techniques are better suitable for real-time rendering, while ray-tracing is a superior technique to create lifelike images. With the advent of massive parallel processors in GPUs, the possibility of using Real-Time Ray-Tracing with mainstream GPUs started to be considered as a serious option to substitute raster based renders, since ray-tracing is trivially parallelizable as shown in [Cook et al. 1984]. However, results of different approaches indicate that Real-Time Ray-Tracing (RTRT) remains a challenging computational task [Aila and Laine 2009; Lauterbach et al. ; Hachisuka 2009] and it is suitable only with special conditions and constraints.

The idea of a hybrid RTRT was developed before our current work. [Beck et al. 2005] proposes a CPU-GPU RTRT Framework and [Bikker 2007] develops a Real-Time Path Tracing called Brigade, which divides the rendering task seamlessly over available compute units, being compute units, GPU or CPU cores. Our main objective is to develop a GPU only renderer that uses both raster and ray-tracing techniques in order to produce real-time images faster than raster or ray traced only techniques. Rendering all the images only in the GPU avoids the common bottleneck problem related to transfer data between CPU and GPU main memories. It is not our goal to replace the raster process, but to improve it directly in the GPU, using ray-tracing only where it seems to be better suitable either for image quality or speed.

This paper presents a real-time GPU only renderer that uses both ray-tracing and raster techniques in order to achieve good performance, when compared to a raster only renderer. Our approach improve the raster process that happens inside the GPU, using ray-tracing only where it improves image quality or render time.

2 Related Works

[Beck et al. 2005] proposes a CPU-GPU Real-Time Ray-Tracing Framework with five render-passes. The first three passes are executed in the GPU and the results are moved to a Depth-Buffer, a frame-buffer with every triangle visible for the camera identified by its number in the RGB part of the frame-buffer, the shadow stored in the alpha channel and a blur pass using the alpha channel mentioned. The fourth pass involves the copy of the frame-buffer to the host memory, where the CPU uses the information generated by the GPU to trace reflection and refraction rays. The last pass occurs inside the GPU, that receives from the CPU a texture that is processed by a fragment shader in order to generate Global Illumination. Beck argues that the CPU is the bottleneck in this process.

NVIDIA's OptiX [Parker et al. 2010] is a ray-tracing engine that runs on NVIDIA's GPUs, also capable of running in general-purpose hardware. OptiX architecture offers a low level ray-tracing engine, a programmable ray-tracing pipeline, a programming model, a domain-specific compiler and a scene representation. It is based on CUDA [NVIDIA 2007] and since NVIDIA announced plans to port CUDA GPU programming platform to x86 processors, up to now, OptiX is a GPU only solution with very good results for interactive ray-tracing for CAD visualization.

In his Master Thesis, [Bak 2010] implements a RTRT using DirectX 11 and HLSL. His work also uses rasterization in order to achieve the best possible performance for primary hits. [Hachisuka 2009] surveyed several ray-tracing algorithms for graphics hardware. An overview of these different methods considering the graphics hardware architecture is described.

Finally, [Chen and Liu 2007] presented a hybrid GPU/CPU renderer, where a Z-buffered rasterization is performed to identify the triangles visible in the scene in order to trace secondary rays. Tests show that the performance of the GPU/CPU renderer outperforms the CPU only version. Their paper also proposes to move the traversal and shading operations to the GPU in order to reduce the memory transfer bottleneck. We extended Chen's work allowing the use of scenes without the need to merge the geometry into one single

mesh. We use one texture color channel as a mesh ID which allows a natural bond of our system with existing ones. Additionally, we remove the bottleneck of memory transfer between GPU and CPU by using the general code execution capabilities of modern GPUs to traversal and shading of secondary rays. Also, we used BVHs as the acceleration data structure for ray-traversal computation.

3 Hybrid GPU Ray-Tracing

Some ray-tracers tries to divide the workload between GPU and CPU in order to achieve maximum performance [Beck et al. 2005; Bikker 2007]. The main problem of this strategy is the bottleneck presented by the data transfer between CPU and GPU. Our approach avoids this using the existing raster pipeline inside the GPU to determine the visibility of objects, simulating the results of casting primary-rays. Secondary rays are generated through fetches on the previous render pass. This strategy takes advantage of the speed of the raster process to define the visibility of the elements in the scene and makes possible the use of ray-tracing to generate other effects, improving visual quality. Other advantage of this strategy is the possibility of ray trace only objects that will benefit of the ray-tracing process, improving the overall performance.

3.1 Deferred Shading Approach

In computer graphics, deferred shading [Hoberock et al. 2009; Deering et al. 1988] is a rendering technique that aims to minimize the lighting computation by splitting the shader task into smaller sub-tasks that are written into an intermediate buffer, the G-Buffer ([Saito and Takahashi 1990]). These subelements are then joined to form the final image. Deferred shading only became more suitable for real-time when the graphics APIs started to support Multiple Render Targets (MRTs) and allowed the use of render-to-texture techniques. One of the main advantages a deferred shading approach is it restricts the calculation of final pixel color only to visible fragments.

Our work combines the benefits of hardware capabilities such as MRT and deferred shading to generate and intersect primary rays using one extra color attachment while the other ones are used for shading non-reflective/refractive materials in the composing stage. The basic idea consists in render an extra color attachment in the G-Buffer where this attachment is the entire scene rendered with colored triangles in which each pixel has the information about where and when it was hit by a primary ray, the distance from the eye and to which object it belongs. This information is used for the subsequent ray-tracing stage for shading surface points correctly as well as to determine from where secondary rays should be traced through scene.

3.2 Primary Rays Generation and Intersection

In a traditional graphics pipeline each vertex that contributes for the final image is processed by a number of operations to determine the pixels that are going to receive its color contribution. With the appropriate attributes, these operations can be converted as an equivalent of the primary ray generation and intersection, making primary ray to scene intersection a raster process.

For each ray, the intersection vector (t, u, v) must be determined, where t is the distance to the intersection point and the pair (u, v) represents the coordinates of the collision inside the triangle. Initially, a frame buffer for offline rendering must be set up in order to render a texture that contains each ray intersection. Note that triangles are the only primitive currently supported. Instead of transfer data vertices with its respective colors, each vertex is transferred to the pipeline with one of the three orthogonal basis vectors: $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ as color information (Figure 2).

In order to identify to which triangle belongs a certain pixel we use a second color attachment on the frame buffer. Each triangle is transferred to the pipeline to be drawn on the second color attachment with color attributes that uniquely represents it. The red color channel carries the triangle ID, starting with 0, of the object's ID stored in the green channel. To avoid interpolation, each vertex of

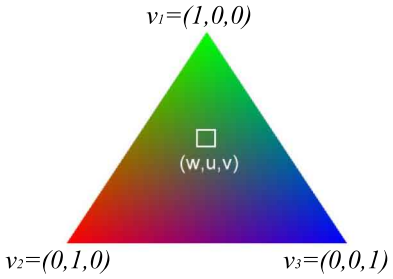

Figure 2: *Each triangle is sent to the pipeline to be rendered on the first color attachment. After rasterization, we get a vector (w, u, v) interpolated from v_0, v_1 and v_2 for each pixel inside the triangle. (w, u, v) vector can be described as the barycentric coordinates of each pixel inside the triangle.*

a triangle is set up with the same color attributes in order to spread this identifiers within fragments generated by that triangle (Figure 3).

After the rasterization is complete, it is obtained a vector (w, u, v) interpolated from the three basis vectors. The (w, u, v) vectors represents the collision coordinates inside the given triangle. The intersection point $\mathbf{x} = (x, y, z)$ can now be interpolated using the following equation

$$\mathbf{x} = wv_0 + uv_1 + vv_2 \tag{1}$$

where v_0, v_1 and v_2 are the vertices of the given triangle. Figures 1(a) and 3 shows a complete scene representation of the primary rays intersection. Note that there is not a visual representation of the second color attachment since we use render-to-texture to write identifiers of triangles and objects for each pixel.

In order to make our system as generic as possible, there is no need to modify any existing shader program to use within our system. A separate shader program is used in this stage at a cost of a render pass.

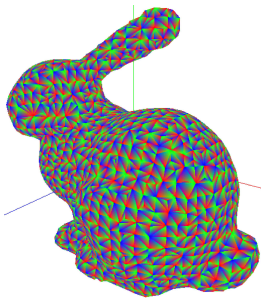

Figure 3: *Low resolution Stanford Bunny rasterized with colored triangles. This is the an example of image stored on the first color attachment.*

3.3 Shading the Appropriate Elements

With the intersection point found on the first rasterization stage, at least one type of secondary ray must be generated, known as shadow ray. This is a special kind of ray in which it is only necessary to find one intersection point regardless of whether this intersection position is. Reflection and refraction rays are also another type of secondary rays, conditionally generated by the type of material a primary ray hits. There are different approaches on how to deal with secondary rays. The simplest one is to assume that every primary ray will generate a secondary ray. In order to deal with this assumption, there must be one CUDA thread per ray, which yields one CUDA thread per pixel. The biggest disadvantage of this approach is that many compute units become inactive because many primary rays could not hit any object, being shaded with scene's background color or texture. With the purpose of not have idle compute units, we keep a list of active rays, which are actually rays to

be traced. The frame buffer is cleaned on the rasterization step with a flag that indicates whether a ray hit or not an object. According to this flag, the list of rays is compacted. The CUDA compute units are then allocated so they can handle this new list of active rays. This ensures that will be as many threads as active rays. Figure 4 shows an example of a list of rays to be traced after compression.

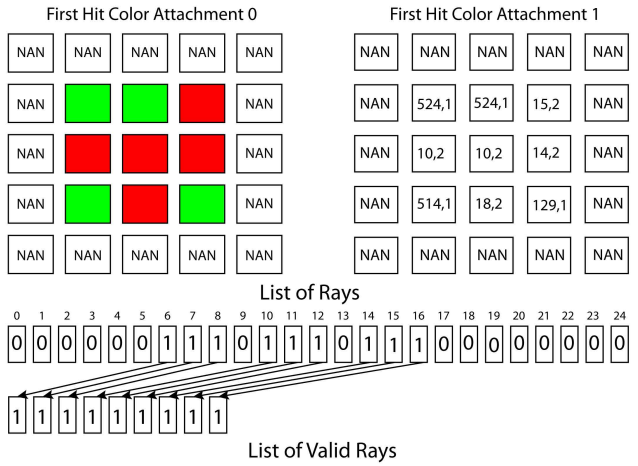

Figure 4: *Representation of the two extra color attachments used for primary rays generation and intersection. On the top left we have the image generated with colored triangles where each pixel contains the barycentric coordinate of the hit point relative to the intersected triangle. On the top right it is shown the content of the second color attachment which represents the object and triangle IDs of intersected points for each pixel. NAN is a symbolic constant representing areas that were not been achieved by any primary ray. On the bottom we have a list of rays generated by this trivial scene and the list of rays after being compacted.*

One issue that needs to be handled in our hybrid approach is to avoid hits of primary rays in rasterized objects that lies in front of ray-traced objects. This issue can be solved naturally within the graphics pipeline using the Z-Buffer. When rendering the scene, only fragments of visible ray-traced objects will remain inside the pipeline. Avoiding primary ray hits on rasterized objects could generate objects without shadow in areas that should be shadowed. In order to deal with this, the entire scene receives primary ray hits, but an extra attribute is stored in the frame buffer, indicating whether a pixel belongs to a rasterized object or not.

Once primary rays are handled, shadows rays need to be traced from the hit point through the scene in the direction of light sources. As the entire scene geometry is known by the ray-tracer, such rays can be blocked by objects where the ray-tracer is not being applied. This implies in rasterized objects ray-tracer shadows naturally. Knowing that this kind of shadow is more accurate than real-time implementations of shadow maps, this is an advantage of our implementation. Precise shadows resulting from the shadow map technique requires a large amount of memory in order to avoid aliasing [Williams 1978].

Figures 6 and 5 shows two examples of objects appearance after shading primary-rays. Figure 6 shows that only visible pixels of objects affected by ray-tracing are shaded due to Depth-buffer algorithm used in generation of the first hit color attachment.

3.4 Scene Rasterization of non-Reflexive / Refractive Objects

One major advantage of a hybrid approach on ray-tracing is the possibility to have only certain objects to be affected by these effects such as crystals, glasses and any other kind of truly reflective/refractive material.

Using MRTs, any object can be rasterized using any kind of shader. The result of the rendering will be written at the main color attachment of the associated frame buffer and composed with the ray tracer at the final stage. Note that this stage can require multiple render passes to achieve the desired result. Since this stage is completely independent from others, this can be done in parallel with primary rays generation and intersection stage only because the targets they aim are different and have no read and write hazards.

This step can be viewed as the result of current rendering techniques, without any modification. Graphics shaders of any kind can by used on this step allowing a natural bond of existing rendering passes with our system. Figure **??**(c) represents a result of such rendering. We use a simple directional light to render the Sponza Atrium scene [Dabrovic 2002] with grayscale tones material in order to best present the integration of ray-tracing effects in rasterized scenes.

Table 1

Number of Vertices and Triangles		
Scene	*Vertices*	*Triangles*
Sponza (Fig. 1(c))	44.404	67462
Bunny (Fig. 5)	34.834	69.451
Toad (Fig. 7)	12.912	25.820
Teaser (Fig. 1(d))	114.072	206.364

Table 2

Performance Tests		
Scene	*Valid Rays*	*FPS*
Teaser Scene (Fig. 1(d))	66.252	10
Toad Scene 1 (Fig. 7)	67.734	5
Toad Scene 2	5.876	10
Bunny Scene 1 (Fig. 5)	68.203	5
Bunny Scene 2	26.442	7

3.5 Final Scene Composition

This step consists in assembling the final image that will be displayed. This process is done by a weighted mean between the rasterized image and the ray traced image. The weights for this mean are relative to the desired amount of ray tracer effects the final image will have. At this point, the frame buffer contains two color attachments, the main one generated by the raster stage and the second one filled by the ray-tracer stage. The depth-buffer and normal-buffers must be fetched to shade appropriately rasterized objects according to a specific material.

Figure 8 shows an example of composed rendered scene. The two purple teapots are not affected by ray-tracing and were rendered using the fixed-function OpenGL pipeline. The red, green and blue planes are diffuse objects rendered with ray-tracing calculations. The golden bunny has a specular property and reflects the background. This scene has two point-light sources which make the bunny to cast shadows onto the background planes.

4 Results

We implemented the system using C/C++ language, OpenGL 3.1 and GLSL 1.2 library for rasterization purposes and CUDA C/C++ for ray-tracing related tasks. The performance results obtained are presented on Table 2 and were measured in a computer equipped with an AMD Phenom(TM) II X4 of 3.4GHz, 4GB of RAM with a NVIDIA 9800 GTX+ GPU with 128 CUDA cores running Windows 7 Professional. In Table 1 it is stated the number of vertices and triangles for different scenes and models we used for test purposes.

Table 2 shows the results of various tests performed with our implementation. For each scene, except the for the teaser (Figure 1(d)), we presented the Frames Per Second (FPS) measurements. Scenes marked with 1 has the camera near the objects, resulting in a greater number of valid rays. On the other hand, scenes marked with 2 has a more distant camera, which implies in a smaller number of valid rays and a higher FPS rate. Note that the FPS rate has a strong dependence on the number of valid rays and on the distance between objects.

5 Conclusion

We have described a ray-tracer with first hits accelerated using traditional graphics pipeline taking advantage of the depth-buffer algorithm implemented on hardware. We extended the work presented in [Chen and Liu 2007] through the possibility of having multiple objects with unrelated meshes. The results were stated in terms of performance for different models and scenes. Finally, it was removed the read-back overhead of data from GPU to CPU memory keeping scene data in GPU all the time.

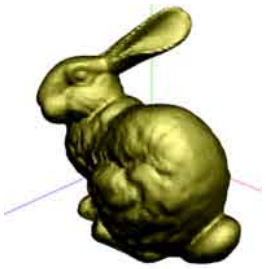

Figure 5: *Bunny rendered with primary rays shading. There are 68.203 valid rays on this scene.*

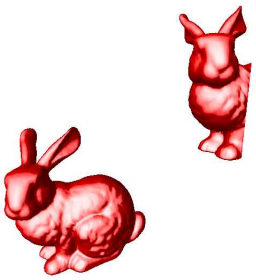

Figure 6: *Example of an image generated using ray-tracing where only the visible pixels of objects affected by ray-tracing effects are shaded. There are 66.252 valid rays on this scene.*

Our system, however, it is not capable of handle interaction of light between objects being rasterized or ray-traced. Also, we do not trace and shade secondary rays other than shadow rays. These features will be implemented in future. We also proposes the use of NVIDIA Optix in the ray-tracing stage of our system taking advantage of the optimized implementation they provide.

As stated, ray-tracing is a technique capable of generating high quality images. With more efficient implementations, along with the advance of computational hardware, it will be possible to simulate effects only reliable in offline rendering.

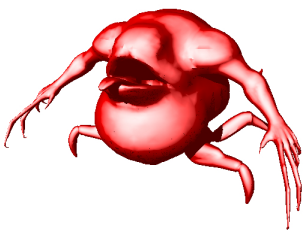

Figure 7: *Toad model rendered with primary rays shading. There are 67.734 valid rays on this scene.*

Acknowledgements

The authors gratefully acknowledge, CNPq, for the financial support of this work.

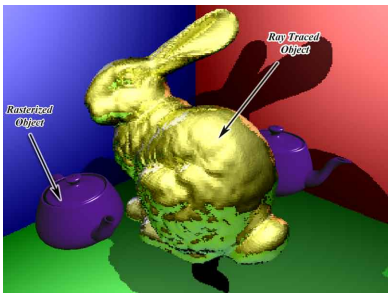

Figure 8: *Composed final image. The red, green and blue planes together with the golden bunny are objects affected by ray-tracing. The two teapots are rasterized objects. Note that the Depth-Buffer keeps visible fragments in front.*

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, 145–149.

BAK, P. 2010. *Real Time Ray Tracing*. Master's thesis, IMM, DTU.

BECK, S., C. BERNSTEIN, A., DANCH, D., AND FRHLICH, B., 2005. Cpu-gpu hybrid real time ray tracing framework.

BIKKER, J. 2007. Real-time ray tracing through the eyes of a game developer. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, Washington, DC, USA, 1–10.

CHEN, C.-C., AND LIU, D. S.-M. 2007. Use of hardware z-buffered rasterization to accelerate ray tracing. In *Proceedings of the 2007 ACM symposium on Applied computing*, ACM, New York, NY, USA, SAC '07, 1046–1050.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18* (January), 137–145.

DABROVIC, M., 2002. Sponza atrium. http://hdri.cgtechniques.com/ sponza/files/.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: a vlsi system for high performance graphics. ACM, New York, NY, USA, vol. 22, 21–30.

HACHISUKA, T. 2009. Ray tracing on graphics hardware. Tech. rep., University of California at San Diego.

HOBEROCK, J., LU, V., JIA, Y., AND HART, J. C. 2009. Stream compaction for deferred shading. In *Proceedings of High Performance Graphics 2009*, 173–180.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. Fast BVH Construction on GPUs. *Computer Graphics Forum 28*, 2, 375–384.

NVIDIA. 2007. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics* (August).

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. 197–206.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *In Computer Graphics (SIGGRAPH 1978 Proceedings*, 270–274.