A Hybrid GPU Rasterized and Ray Traced Rendering Pipeline for Real Time Rendering of Per Pixel Effects

Thales Luis Sabino, Paulo Andrade, Esteban Walter Gonzales Clua, Anselmo Montenegro, Paulo Pagliosa

A Hybrid GPU Rasterized and Ray Traced Rendering Pipeline for Real Time Rendering of Per Pixel Effects

Thales Sabino[1], Paulo Andrade[1], Esteban Clua[1], Anselmo Montenegro[1], and Paulo Pagliosa[2]

[1] Universidade Federal Fluminense, Niterói - Rio de Janeiro, Brazil,
{tsabino, pandrade, esteban, anselmo}@ic.uff.br
[2] Universidade Federal do Mato Grosso do Sul, Campo Grande - Mato Grosso do Sul, Brazil
pagliosa@facom.ufmt.br

Abstract. Rendering in 3D games typically uses rasterization approaches in order to guarantee interactive frame rates, since ray tracing, a superior method for rendering photorealistic images, has greater computational cost. With the advent of massively parallel processors in the form of GPUs, parallelized ray tracing have been investigated as an alternative to rasterization techniques. While many works present parallelization methods for the classical ray tracing algorithm, in order to achieve interactive, or even real time ray tracing rendering, we present a rasterized and ray traced hybrid technique, completely done in GPU. While a deferred render model determines the colors of primary rays, a ray tracing phase compute other effects such as specular reflection and transparency, in order to achieve effects that are not easily obtained with rasterization. We also present a heuristic approach that select a subset of relevant objects to be ray traced, avoiding traversing rays for objects that might not have a significant contribution to the real time experience. This selection is capable of maintaining the real time requirement of games, while offering superior visual effects.

Keywords: ray tracing, rasterization, OptiX, CUDA, GPU, hybrid rendering, OpenGL, GLSL, real-time, global illumination effects, deferred shading

1 Introduction

In the computer graphics field, it is a common belief that raster techniques are better suitable for real-time rendering while ray tracing is a superior technique to create photorealistic static images, due to the ray tracing processing cost. Recent Graphics Processing Units (GPUs) can also work as general-purpose massively parallel processors [1], promoting the possibility of Real-Time Ray Tracing (RTRT) implementations using GPUs as an alternative to raster based approaches due to its parallel nature by its concept [2]. Many works explore the

possibility to accelerate ray tracing in order to make it a real-time process with GPU only or hybrid CPU and GPU approaches. However, results of different investigations indicate that RTRT remains a challenging computational task [3–6], suitable only with specific conditions and constraints.

Offline renderers accurately simulate the way light interact with surfaces. Since the simulation of every beam of light in a 3D space is impractical, other approaches were developed to simulate the way light works. One of these approaches is the Backward Ray Tracing, where rays are launched from the camera position to different parts of the 3D space. Depending on the characteristics of the surfaces of the objects hit by the rays, secondary rays may be recursively generated. These secondary rays can be used to model many light effects such as soft shadows, mirror and diffuse reflection, depth of field, motion blur, refraction, transparency, among others.

RTRT is a hard computational task, not only because each pixel in the image must be calculated separately, but also because the final color of a single pixel can be affected by more than one recursive ray. Another consideration is that ray tracing algorithms waste from 75% to 95% of its execution time calculating intersection points between rays and objects [7].

In this paper, we describe an improvement of the graphics pipeline, by creating a hybrid ray trace and raster rendering process. In this method, the deferred rendering process is used to calculate the primary rays collision, while the secondary rays use a ray tracing approach to obtain shadow, reflection and refraction effects. This approach vastly improves ray tracing performance, not only because we avoid many unnecessary traditional ray tracing tasks, but also because we guarantee that a complete image will be available to show in a demanded time, even if there is not enough time to finish all the effects calculations. We also developed an efficient and customized data structure to manage part of the geometry inside the GPU memory and a heuristic approach that adequately chooses a subset of objects to generate visual effects using secondary rays.

This proposal differs from most related works, since the goal is not entirely replace the graphics pipeline by a ray tracing only approach. This work use GPU ray tracing only for a set of objects, seeking for those that will have more visual impact for each frame generated. The main focus is to render complex scenes in real-time using raster techniques and get the refined best visual look including algorithms that simulate global illumination effects. Since most global illumination effects are better simulated using ray tracing techniques, one approach proposes to combine these two techniques in the same rendering pipeline. This paper presents and discusses strategies applied to develop a real-time hybrid GPU-only ray tracer, that uses raster techniques to improve the performance of a ray tracer and a smart strategy for prioritizing regions and objects that will receive the ray tracing light effects.

This paper is organized as following: after a brief introduction we present different hybrid CPU and GPU raytracing approaches in Section 2, then we present a hybrid ray tracing model, detailing the composition methods in Section 3. Section 4 presents and discusses our heuristic approach for prioritizing regions

and objects for the ray tracing phase. In Section 7 we present implementation details, followed by the results, comparisons and discussions, in session 8. Finally, we conclude the work and state future proposals in Section 9.

2 Related Work

The concept of a hybrid Real-Time Raster and Ray Tracer renderer is not new. Beck et al [8] proposes a CPU-GPU Real-Time Ray-Tracing Framework and Bikker [9] developed a Real-Time Path Tracer called Brigade, which divides the rendering task seamlessly over both GPU and CPU available cores. Brigade aims the production of *proof-of-concept* games that use path tracing as the primary rendering algorithm.

Beck [8] proposal spread the traditional stages of ray tracing in independent tasks for the GPU and CPU. These render tasks can be summarized into three GPU render passes: a shadow map generation pass, a geometry identification pass and a blur pass. In the geometry identification pass, the triangle numbers are encoded as RGB colors in a framebuffer and the result of the shadow map pass is blurred inside the alpha channel, in the same framebuffer. The CPU receives the framebuffer for reflection and refraction generation pass using a ray tracing algorithm. Finally it is applied a phong shading pass that also merges the results of the other passes.

NVIDIA's OptiX [10] is a general purpose ray tracing engine targeting both NVIDIA's GPUs and general-purpose hardware in the current version. OptiX architecture offers a low level ray tracing engine, a programmable ray tracing pipeline with a shader language based on CUDA C/C++, a domain-specific compiler and a scene-graph based representation. OptiX is a GPU only solution with remarkably good results for interactive ray tracing.

Pawel Bak [11] implements a Real-Time Ray Tracer using DirectX 11 and HLSL. Similar to Beck's work [8], his approach also uses rasterization in order to achieve the best possible performance for primary hits but uses DirectX 11 and HLSL instead of OpenGL and GLSL.

Chen [12] presented a hybrid GPU/CPU ray tracer renderer, where a Z-buffered rasterization is performed to determine the visible triangles at the same time that primary rays intersections are determined. The CPU reads the data back in order to trace secondary rays.

Finally, Hachisuka [5] surveyed several ray-tracing algorithms for graphics hardware. The author concludes that although the ray tracing method seems to be extremely well studied and their parallel nature can be understand easily, many issues need to be solved to implement it efficiently. Some of the issues were presented, such as data retention, the requirement of graphics hardware to perform the same computation over several pixels and their stack-based characteristic. His work classifies different approaches based on their features. Additionally, these algorithms are compared considering their bottlenecks and possible improvements are also discussed.

3 Hybrid GPU Ray Tracing

Most works about hybrid ray tracing techniques tries to balance the workload between GPU and CPU in order to achieve maximum performance of both parts. Using modern graphics hardware and APIs that allows efficient use of rendering techniques such as Multiple Render Targets (MRT) and Deferred Shading [13], it is possible to use the traditional raster pipeline as the first stage of a complete ray tracing pipeline: the generation and intersection of primary rays.

3.1 Primary Rays Generation and Intersection with Deferred Shading

Deering [13] introduced the idea of deferred shading. Although the article never uses the term "deferred", it introduces a key concept; each pixel is shaded only once after depth resolution. The term *deferred shading* was adopted later, and its usage in real-time rendering applications, mainly video-games, became mainstream since 2008.

The basic idea behind deferred shading is to perform all visibility tests before performing any lighting computations. In traditional GPU rendering, the Z-buffer normally shades as it goes. This process can be inefficient, as a single pixel often generates more than one fragment. A rough front-to-back sort of objects can help to avoid this problem, but deferred shading perfectly solves it, and also solves light/material combination issues [14].

Deferred Shading takes advantage of a structure called G-Buffer [15]. This structure is filled with geometric data, using the presented geometry pass. Values saved includes the z-depth, normal, texture coordinates and material parameters. These values are saved to Multiple Render Targets accessed by the pixel shader program.

The values stored into G-Buffers depend on the application. For this work, we store the following per-pixel values, one for each target: position, normal, z-depth, albedo color and specular color. These values are used later in the lighting stage of deferred shading (see Figure 1). For information about deferred shading implementation in this work, please refer to Section 7.

Although the information stored in the G-Buffer is enough to determine the origin and direction of shadow rays, it is not possible to establish the need for tracing reflection and refraction rays only with geometry data. The G-Buffer is extended with one more render target that stores optical properties of each pixel. Reflectivity, index of refraction, opacity and specular exponent are included in this target.

Deferred shading does have some drawbacks. The video memory requirements and fill rate costs for the G-Buffers are significant. One of the most significant technical limitations is in the process of antialiasing and transparency. Antialiasing is not performed and stored for rendered surfaces. To overcome this limitation, antialiasing image-based techniques such as Shishkovtosov edge-detection method [16] and, more recently, Morphological Antialiasing [17] can be employed for better image quality.

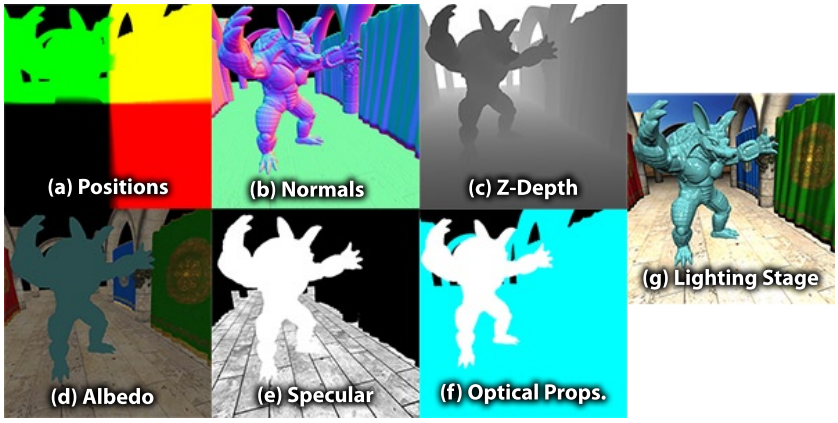


Fig. 1. The six textures generated by the geometry pass: (a) position, (b) normal, (c) z-depth, (d) diffuse or albedo color, (e) specular color and (f) encoded optical properties. (g) is the image produced after the lighting stage.

3.2 Selective Global Illumination Effects

In ray tracing, secondary rays are used to produce global illumination effects. Once primary rays are handled, shadows rays need to be traced from the hit point through the scene in the direction of light sources. Shadow maps present some advantages over ray traced shadows, including avoiding the knowledge of the scene geometry and automatic adjustment over geometry changes on GPU. On the other hand it requires a large amount of memory in order to avoid aliasing, and the scene must be rendered once per light source. Precise shadows resulting from the shadow map technique require a large amount of memory in order to avoid aliasing [18].

The G-Buffer produced by the deferred rendering stage contains information about *optical properties* of the underlying material of each pixel. The render target represented by Figure 1(f) is used to determine the need for tracing reflection/refraction rays. It is composed by reflectivity, index of refraction, specular exponent and opacity, respectively. The Armadillo was highlighted meaning that rays only need to be traced from its surface through the scene. Since it is composed by a refractive material, we are able to avoid trace of unnecessary rays in places where the material is neither refractive nor reflective.

At this point, the ray tracing algorithm can follow its own path. Any secondary ray generated will be traced against scene in order to produce other global illumination effects, such as reflections and refractions. The result of this stage can be understood as the generation of a ray trace effects layer. This effects layer will be blended to the image already generated, in order to improve its visual quality with global illumination effects.

4 Ray Tracing Heuristic For Real Time Rasterization Pipelines

In order to improve the performance of a hybrid ray tracer for real time applications, we propose a heuristic to dynamically select relevant objects to trace. The selected objects are ray traced for effects in a predefined and fixed time constraint. The heuristic is capable of selecting the most relevant objects in a scene and still maintain the expected frame rate. We define relevant objects as those that better contribute to the visual experience at a specific frame.

The idea of ray tracing a subset of objects and still have an improved experience in real time applications is based on the real world observation that when the perceived image constantly changes, as when we drive a car or walk, many elements are ignored by our conscious mind. This simple fact is the motivation of the development of warning signs, as the ones we see in highways to call our attention for relevant information. It is a fact that our attention is more focused in near objects, and objects directly in front of our field of view. Considering this observation, a simple heuristic would be to *trace the objects near the observer, and in front of his/her field of view*. Unfortunately, this simple heuristic is not capable of guaranteeing performance, also, not all objects improve in visual quality when using ray tracing instead of rasterization. The decision of what objects deserve to be ray traced is an environment designer decision since no technology can substitute the artist's eye. The heuristic tries to select the best candidates to ray trace, using some predefined information.

4.1 Pre-Production Phase

The first step happens during the scene design. This step is the selection of a group of objects that are candidates to ray trace for a given environment. For every object chosen, it is also necessary to define the kind of secondary rays that should be generated after the first hit (primary ray). In this work, we considered three different light effects to be ray traced if possible: reflection, refraction and shadows for high detailed objects.

The initial cost of the object is defined by its initial visibility (V), and the estimated number of secondary rays needed to represent the object light effect (Q). Visibility is defined by Equation 1 and Object cost (C) is defined by Equation 2. Visibility is defined by the average area initially visible of the object (A) times the 2D projected distance of the object considering the center of the projection (P) divided by the distance (D) of the object from the observer. The farthest the object, less visible it is, considering its area. Also, the farther the object is from the center of the 2D projection of the visible space, less relevant it is for the final image. P is a value in the range $[0, 1]$, where one is the exact center of the scene, and 0 means the object is outside the field of view. Both equations are simplifications of the complete formulation.

$$V = \frac{A \times P}{D} \tag{1}$$

$$C = V \times Q \tag{2}$$

Finally, every object has a relevance factor R defined by the fact that it was selected S before to be traced. We call as S a binary value of 0 (previously unselected) or 1 (previously selected), times a constant that represents the importance I of the render of previous rendered objects (defined by the application. Equation 3 represents relevance for our heuristic.

$$R = S \times I + V \tag{3}$$

All the tree equations are calculated for all the objects, frame by frame, in order to update the scene graph.

4.2 Selection Graph

When the GPU loads the information necessary to render the objects selected for ray tracing in its memory, a directed graph with information of the objects is built for the ray trace phase. Since the GPU is a massive parallel processor, the number of initial objects to be traced is proportional to the number of Stream Processors of the GPU architecture. The graph is a directed graph, where every node represents an object. A node points to another whose cost (C) is smaller than the actual node but equal or higher than the cost of all other nodes. Also, every node points to all N other nodes that are more relevant but not selected yet to be ray traced. The N pointers in every node are ordered considering their relevance. When an object finishes rendering, the graph is traversed in order to find the node whose rendering cost is less than the cost required to render it, considering the time still available for the ray tracing phase. If the node points to other node more relevant node, this one is selected to be ray traced, and the graph is updated. Figure 2 presents the graph structure. Red circles are the first nodes selected because of their relevance, and the gray circles are nodes that will still be selected. Black arrows represent the nodes order according to their cost (C) and the blue dashed arrows represent the N more relevant nodes (R) than the selected node, and with less cost.

Table 1. Cost and Relevance for the Selection Graph represented in Figure 2

Node	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
C	-	-	-	-	-	-	-	-	20	19	18	17	16	15	**14**	**14**	13	12	11	10	**9**	**9**
R	-	-	-	-	-	-	-	-	22	21	18	30	31	32	28	27	9	29	1	20	4	3

4.3 Object Selection and Graph Rebuild

When the render of the first object is finished, the next one must be selected according to the time left to render. The selection graph is traveled until a node

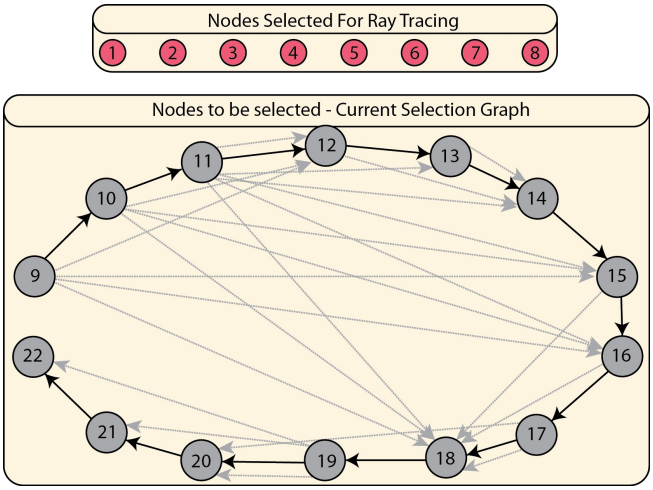


Fig. 2. Selection Graph.

with an estimated cost less than the cost left to render is achieved. In this node, it is verified if it points to another node less relevant. If so, this node is also verified for relevance, until the most relevant node is found. When the node is selected to render, it is also moved to another graph, under construction for the next frame. Using the information provided by render engine, this node and all the nodes with cost higher than the cost used to find the first node are inserted into the new graph and the graph is properly updated.

The current graph is also updated if the selected node is not at the node that was selected only according to its cost. If no time is left to render any object, this time is used to move all the nodes not selected to the new graph. Figure 3 represent the selection of a new node to render and the new future selection graph. Considering that the time left to render allows just a node with cost 18 or less to be selected, nodes 9 and 10 are removed from the selection graph and will be moved to the new graph, in construction. Since node 11 has cost 18 but is not the most relevant, node 14 is selected and the corresponding object is ray traced.

5 Final Scene Composition

This step consists in assembling the final image that will be displayed. This process is done by a blend operation between the rasterized image and the ray traced image. The weights for this blend are relative to the desired amount of ray tracing effects desired for the final image. At this point, the frame buffer contains two color attachments, the first one generated by the raster stage and the second one filled by the ray tracer stage.

The simplest approach to generate the final image, given a filled framebuffer, is to superimpose the ray traced image with the rasterized image. This can be

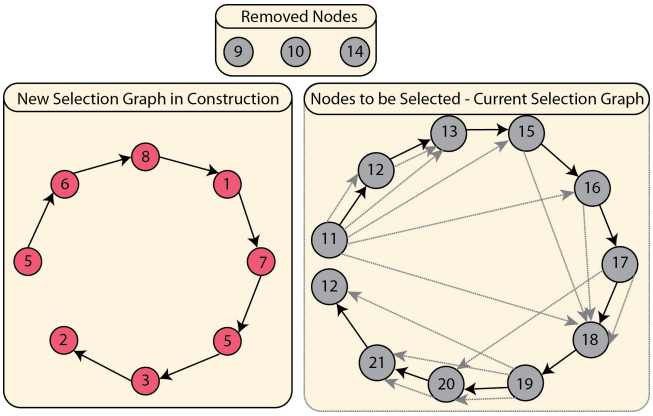


Fig. 3. Current Selection Graph and the Future Selection Graph in Construction.

easily accomplished with a shader program adding another light-weight render pass to the process, since we only need to render a screen-size quadrilateral primitive. Note that a simple superimposing of the ray traced image with the rasterized one will not present additional problems since only pixels that were shaded using the ray-tracing engine are visible.

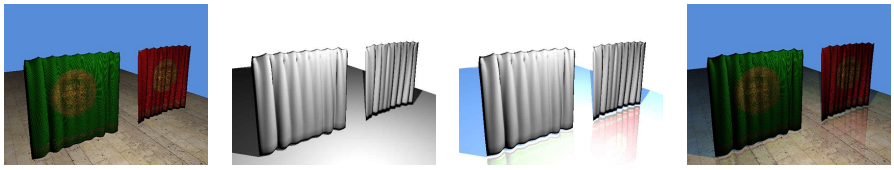
6 Hybrid Pipeline Overview

In previous sections we described the stages that compose our hybrid raster/ray tracer pipeline. In this Section, we present an overview of the complete pipeline.

In order to illustrate the process, we use a simple scene composed of three textured objects and one point light. This scene is composed by 4,746 vertices and 9,470 faces. Figure 4 shows the five stages of our proposed pipeline:

- **Deferred rendering and primary Rays resolution:** Stage 4(a) consists in rendering the scene using a deferred rendering technique resulting in a filled G-Buffer. The obtained G-Buffer contains information about visible fragments, and it can be seen as a primary ray resolution stage of a traditional ray tracer. The result image of this stage has the edge-detection antialiasing algorithm applied [16].
- **Shadows:** With data extracted from the G-Buffer, shadow rays are generated for valid hit points. In Figure 4, invalid hit points are those that must be shaded using the sky color or environment map, being such pixels the blue ones of Figure 4(a). The shadow pass result is presented in Figure 4(b).
- **Reflections and refractions:** Stage 4(c) follows the ray tracing algorithm in order to calculate reflection and refractions. For this example, we use reflections on the ground to illustrate an application of such effect.
- **Final Composing:** The final composing stage 4(d) is responsible for blending images generated at stages 4(a) and 4(c). We compose this scene using a

shader program to draw the ray tracing effects layer with an OpenGL blending operation. The amount of ray traced effects on the final image may also be configured.



(a) (b) (c) (d)

Fig. 4. Overview of our hybrid rendering raster/ray tracing pipeline. The first stage *(a)* consists in rendering the scene using a deferred rendering technique resulting in a filled G-Buffer and a basic rendered image. Stage (b) traces shadow rays. Stage (c) is responsible for tracing and shading secondary rays. In this example, the ground is made of a reflective material. Finally, stage (e) is responsible for blending images generated at stages (c) and (a).

7 Implementation Details

In this section, we give more details of our implementation concerning efficiency and robustness. This system was implemented using C++ as the main language. The *Open Asset Import Library* was used in order to read common scene file formats. The *Free Image Project* was used to read texture images. We use GLSL as the shading language on the primary rays resolution stage and for rendering purposes. The NVIDIA OptiX engine was adopted to trace and shade secondary rays, due to its practicability in terms of freedom on building the ray tracing pipeline and hiding GPU hardware details when writing ray tracing shaders. Besides the advantages of being a generic ray tracer engine, its bond to OpenGL for reading and writing graphic resources is straightforward.

The geometry stage of deferred shading implements a perspective projection camera, and it is responsible for feeding the G-Buffer with information to be used in the next stages. All subsequent stages operate in image-space and work with an orthographic projection camera using the screen resolution dimensions. The lighting stage receives as input the contents of the G-Buffer as well as light sources information and it accumulates lighting effects into a full resolution P-Buffer (pixel buffer). Table 2 shows the configuration adopted in the implementation of the deferred shading technique.

[1] GL_RGBA32F was chosen because of the need to store the coordinates in view-space in the position buffer and values that lies outside the rage $[0, 1]$ for the Optical Properties buffer.

[2] GL_DC is the abbreviation for GL_DEPTH_COMPONENT.

Table 2. Configuration properties of the G and P-Buffers.

Storage Options for G-Buffer		
Target	*Format*	*Type*
Position	GL_RGBA32F[1]	GL_FLOAT
Normal	GL_RGBA8	GL_UNSIGNED_BYTE
Depth	GL_DC24[2]	GL_DC[2]
Albedo	GL_RGBA8	GL_UNSIGNED_BYTE
Specular	GL_RGBA8	GL_UNSIGNED_BYTE
Optical props	GL_RGBA32F[1]	GL_FLOAT
Antialias	GL_RGBA8	GL_UNSIGNED_BYTE
Destination	GL_RGBA8	GL_UNSIGNED_BYTE

8 Results

In this section, we present the results in terms of performance comparing our hybrid approach with a pure GPU ray tracer as well as a comparisson with Brigade Real-Time Path Tracer. The difference between the pure GPU ray tracer implementation and our hybrid implementation is that, instead of using the G-Buffer information, primary rays are generated by the camera. In the pure GPU ray tracer implementation, OpenGL render is disabled.

We conduct the tests in a desktop system equipped with an AMD Phenom II X4 965 3.4GHz, 16GB of RAM and a NVIDIA GeForce GTX570. As operating system, we use the 64bits version of Windows 7 Professional.

Table 3 summarizes the results in terms of performance with the measurements based on frames per second (FPS) for each scene. All scenes were rendered with resolution 1280x720 pixels.

Performance comparison in milisseconds (*ms*)						
Scene	*Vertices*	*Triangles*	**DR**[3]	**RT**[4]	**Hybrid**	**Gain**
Armadillo (Fig. 5(c))	193737	380655	3,134	25,112	18,518	1,35
Sponza (Fig. 5(a))	145173	262187	2,227	66,667	34,482	1,93
Sponza Simplified	48556	90349	1,666	33,334	25	1,33
Showcase (Fig 5(b))	112603	224440	2,336	40	23,255	1,72

Table 3. Performance comparison between our hybrid render and a pure ray tracer implementation. The times are expressed in milliseconds.

As one can see in Table 3, our hybrid implementation was able to achieve a higher frame rate than a pure ray tracer implementation. It is easy to notice that both the Armadillo Scene and the Showcase Scene got a bigger difference than the

[3] DT stands for *Deferred Rendering Time*. It is the time our implementation spends with the rasterization stage.

[4] RT stands for *Ray tracing Time*. It represents the time necessary to render the scene with a ray tracing-only renderer.

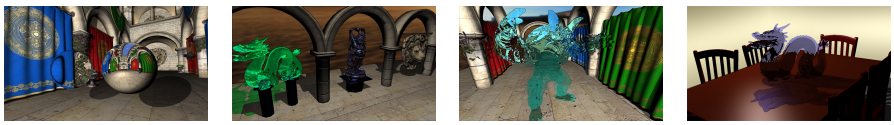
Sponza Scene. This is because Armadillo and Showcase are open environments, which means that more rays are likely to escape avoiding extra computation. Since Sponza Scene is a closed scene, the method tends to spend more time when tracing both primary and secondary rays.

The comparison with Brigade was made with the Dining Room scene. It is available as a demo in Brigade Homepage [19]. Table 4 shows the results.

We expect to reduce the time of our approach using OptiX support for mipmaps, since coalesced memory access in GPU is one of the keys to achieve real-time frame rates.

Performance comparison in milisseconds (*ms*)				
Scene	*Vertices*	*Triangles*	**Brigade**	**Hybrid**
Dining Room (Fig. 5(d))	386541	224954	76,920	28,283

Table 4. Performance comparison between our hybrid render and Brigade. The times are expressed in milliseconds.



(a) (b) (c) (d)

Fig. 5. Images rendered with our hybrid ray tracer implementation. *(a)* Sponza Atrium Scene. This scene highlights the real-time reflection and refractions effects. *(b)* Showcase Scene. In this scene we highlight the colored shadows of transparent objects. *(c)* Armadillo Scene. This is demonstration of a refractive material applied to a model. *(d)* Dining Room Scene. All images have a resolution of 1280x720 pixels.

9 Conclusion

We have described a ray tracer approach with primary ray hits using traditional graphics pipeline, taking advantage of the depth-buffer algorithm implemented on hardware and the data stored in the G-Buffer generated with deferred shading strategy. It was presented a way to avoid merging the scene geometry into a single mesh. We also present the details regarding implementation on how to use the same geometry on OpenGL and OptiX contexts as well as the removal of the common bottleneck of data transfer between GPU and CPU.

We present results in terms of performance for different scenes and show that our hybrid approach can accelerate ray tracing by turning the primary rays

resolution stage into a render task. We highlight the fact that our system was built so that existing shaders are not modified taking advantage of the graphic effects created so far. The heuristic approach guarantees that we process the most important objects with ray tracing and guarantees that the real time constraint is being maintained, something that is fundamental for virtual reality applications.

As future works, we propose the development of new heuristics to find the best elements where to apply ray tracer effects. We also want to compare the use of shadow maps with ray traced shadows in terms of performance and visual quality within our hybrid pipeline. This may drastically reduce the number of rays that need to be traced. At the primary rays resolution stage, we propose a generation and compaction of a list of valid rays. Also, we want to incorporate animations in our implementations in order to evaluate the performance of rebuilding the acceleration data structures on the fly.

Considering the pre-production phase, where the objects are selected to be ray traced, one strategy not explored in our work is to have more than one model to represent the same object, using discreet level of detail approach. The artist could create, for example, tree objects, one highly detailed, one with medium detail and a simplest one. The level of detail does not need to be only in the geometry, but also in the effects characteristics. A translucent object, for instance, could have different diffuse characteristics, proportionately reducing the work necessary to ray trace.

One of the advantages of ray tracing over raster techniques is its natural way of using parametric surfaces. Implicit representation of surfaces requires only the storage of its associated parameters and ray intersection algorithms are faster when dealing with parametric equations. Also, due to the complexity of incorporation in raster systems, in future, we want to incorporate implicit surfaces in our hybrid approach. When computing the first hit stage, the object could be approximated with a low resolution polygonal mesh and then, the ray tracing stage will be responsible for global illumination effects applied on the accurate implicit representation.

As stated, ray tracing is a technique capable of high quality image synthesis. Across with the more efficient implementations, along with the advance of computational hardware, at future it will be possible to simulate effects only reliable in offline rendering. At the future, we may see global illumination effects applied in real-time at high frame rates inside game and visualization environments.

Acknowledgment

The authors would like to thank Leandro Fernandes for his help with shader programs, NVIDIA and people that spent a little of time discussing OptiX and CUDA related subjects in the web forum and CNPq for the financial support of this work. We also thank Crytek for proving a renewed Sponza scene suitable for testing on modern graphics cards.

References

1. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
2. Bigler, J., Stephens, A., Parker, S.: Design for parallel interactive ray tracing systems. In: Interactive Ray Tracing 2006, IEEE Symposium on. (sept. 2006) 187 –196
3. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on gpus. In: Proc. High-Performance Graphics 2009. (2009) 145–149
4. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D.: Fast BVH Construction on GPUs. Computer Graphics Forum **28**(2) 375–384
5. Hachisuka, T.: Ray tracing on graphics hardware. Technical report, University of California at San Diego (2009)
6. Heirich, A., Arvo, J.: A competitive analysis of load balancing strategies for parallel ray tracing. The Journal of Supercomputing **12** (1998) 57–68 10.1023/A:1007977326603.
7. Whitted, T.: An improved illumination model for shaded display. Commun. ACM **23** (June 1980) 343–349
8. Beck, S., c. Bernstein, A., Danch, D., Frohlich, B.: Cpu-gpu hybrid real time ray tracing framework (2005)
9. Bikker, J.: Real-time ray tracing through the eyes of a game developer. In: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, Washington, DC, USA, IEEE Computer Society (2007) 1–10
10. Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., Stich, M.: Optix: A general purpose ray tracing engine. ACM Transactions on Graphics (August 2010)
11. Bak, P.: Real time ray tracing. Master's thesis, IMM, DTU (2010)
12. Chen, C.C., Liu, D.S.M.: Use of hardware z-buffered rasterization to accelerate ray tracing. In: Proceedings of the 2007 ACM symposium on Applied computing. SAC '07, New York, NY, USA, ACM (2007) 1046–1050
13. Deering, M., Winner, S., Schediwy, B., Duffy, C., Hunt, N.: The triangle processor and normal vector shader: a vlsi system for high performance graphics. Volume 22., New York, NY, USA, ACM (June 1988) 21–30
14. Akenine-Möller, T., Haines, E., Hoffman, N.: Real-Time Rendering 3rd Edition. A. K. Peters, Ltd., Natick, MA, USA (2008)
15. Saito, T., Takahashi, T.: Comprehensible rendering of 3-d shapes. SIGGRAPH Comput. Graph. **24**(4) (September 1990) 197–206
16. Shishkovtov, O.: Deffered shading in s.t.a.l.k.e.r. GPU Gems 2 **2** (2005) 143–166
17. Reshetov, A.: Morphological antialiasing. In: Proceedings of the 2009 ACM Symposium on High Performance Graphics. (2009)
18. Williams, L.: Casting curved shadows on curved surfaces. In: In Computer Graphics (SIGGRAPH 1978 Proceedings. (1978) 270–274
19. Bikker, J.: Brigade: Real-time path tracer. Webpage (04 2012)