An Adaptive Hybrid Raster Ray Tracing Renderer for Real-Time Applications

Felipe Martinez, Esteban Clua
Thales Sabino, Anselmo Montenegro
Instituto de Computação
Universidade Federal Fluminense

Paulo Pagliosa
Faculdade de Computação
Universidade Federal do Mato Grosso do Sul

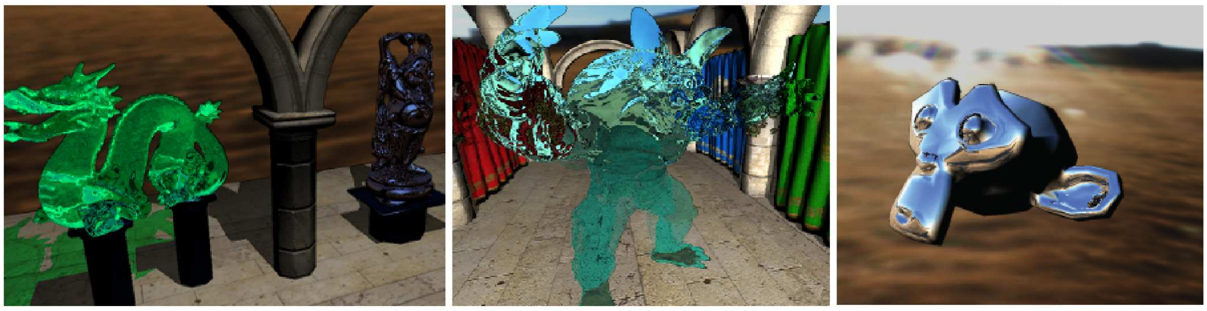
Figure 1: *Three scenes rendered by our hybrid deferred shading/ray tracing method. It is worth noting: the colored shadows in the left image; the refraction effects in the middle one, which reveal colors of the curtains background; and, in the right image, the reflection of the environment map over the metallic monkey surface.*

Abstract

Rendering in real time applications is usually based in rasterization techniques, as they are highly dependent on interactive frame rates. Ray tracing, a far superior method for rendering photorealistic images, has little participation in this media, mainly because it comes with a greater computational cost. The advent of massively parallel processors in the form of GPUs, however, has been changing this scenario since parallelized ray tracing has been showing itself an increasingly viable alternative. Taking advantage of this trend, many works present parallelization methods for the classical ray tracing algorithm, as other ones introduce new ways of combining the two approaches in order to take the better of each one without breaking any of the restraints associated with real time applications.

Although the application of these hybrid rasterizing/ray tracing techniques have reduced the time needed to render the final image and thus, made new scenes viable in real time systems, they are still limited to the complexity of the 3D scene. The need to render a high number of complex objects could drop the performance of the application and reach an impracticable frame rate for real time parameters. While many works investigate new forms of accelerating this process, increasing the number of scenes that able to be rendered in time, we present a more robust proposal, capable of maintaining a reasonable frame rate under any circumstances. It is done by analyzing the scene and rendering it in the best way possible, detailing the most important objects and respecting the time budget for the remaining ones. We also present a heuristic approach that select a subset of the scene to be ray traced, avoiding objects that might not have significant contribution to the real time experience. This selection is a step forward in the field as it is capable of maintaining the real time requirement of some applications, whilst bringing the best possible experience to the viewer.

Keywords:: Real-time rendering, Ray tracing, Rasterization, NVIDIA OptiX

Author's Contact:

{fmartinez, esteban, tsabino, anselmo}@ic.uff.br
pagliosa@facom.ufms.br

1 Introduction

In 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. Thus, even though the improvement that comes with it in the final image quality is welcome in any scenario, when it comes to interactive applications, the rasterization approach is a far more viable option.

Recently, the aforementioned high computational cost of the ray tracing algorithms has been bypassed by the use of parallel techniques. The most modern graphics processing units (GPUs), which have been used as general-purpose massive parallel multiprocessors [NVIDIA 2007; kir], can be highly compatible with real-time ray tracing implementations, since this is a parallel problem by concept [Bigler et al. 2006]. Although this motivates a lot of works in the area, we still find the use of ray tracing techniques in real time scopes restrict to very specific scenarios [Aila and Laine 2009; Bak 2010; Hachisuka 2009; Heirich and Arvo 1998].

Many works have being developed investigating the use of hybrid renderers, aiming to combine the advantages of both approaches in order to obtain a final image with a better quality but still capable of being processed in the time budget required by the interactive character of the application [Beck et al. 2005; Sabino et al. 2012].

Real-time renderers have its success based in the generation of at least fifteen frames per second, so that the application can run smoothly. That is a completely different proposal from offline renderers, which aims for the most photorealistic result leaving the time restraints as a secondary factor. So, naturally, rasterization dominates real-time rendering, a method that convert vector information correspondent to a three-dimensional space into the raster space. It can be done very quickly as the graphics pipeline, the core of modern GPUs design, has been vastly optimized for the process.

Offline renderers accurately simulate the way light interact with surfaces. Obviously, the simulation of every beam of light in a 3D space is impractical, so that, other approaches were developed to approximate the way light spread through the scene. An example of that is the Backward Ray Tracing, where the rays are launched from the camera (or the viewer) to the different objects of the scene, instead of the other way around. Depending on the characteristics of the surfaces hit by the rays, secondary rays may be recursively generated. This second flow of rays can be used to model new light effects to the scene such as shadows, reflection, refraction, and many others. Although these recursive rays are responsible for the new photorealistic features of the final image, they come with the high complexity and time consumption disadvantages [Whitted 1980], since each pixel can generate not one, but an infinity of rays. The problem becomes even harder when, for each one of them, comes the need to compute the intersection points with the scene objects.

The concept of a hybrid renderer combining the advantages of a raster pipeline and ray tracing techniques is not new. Beck et al [Beck et al. 2005] proposed a CPU-GPU combined framework based in real-time ray tracing. Besides him, Bikker [Bikker 2007] developed the Brigade, with a similar concept, dividing the render talks between the available processing units, either GPU or CPU cores. Finally, Sabino et al [Sabino and Clua 2012] were able to develop a hybrid pipeline framework in which this work is based on. In this approach, we choose not to split the workflow between the CPU and GPU in order to get a simple and straightforward speedup. Our method uses both raster and ray tracing techniques to obtain the final image, rendering a full rasterized image, computed in GPU, and applying the secondary effects from the ray tracing approach. In this way, the bottleneck effect created by the often communication between the CPU and GPU graphic memories is eliminated.

In this method, the performance of the ray tracing pass is optimized, as we can use the deferred shading [Deering et al. 1988] technique to compute the collision between the 3D scene objects and the first ray batch coming from the camera. The following rays, however, are traced in the conventional way, which result the photorealistic effect we are aiming for. Thus, depending on how many objects there are in the scene and the complexity of calculating the exact color of their surfaces, the renderer may not be able to achieve a frame rate compatible with the application.

The objective of this work is to investigate ways of adapting the advantages of this hybrid renderer in a much more robust system, less sensitive to the complexity of the data input. This system has to render the final scene within the pre-established time constraints, capable to maintain the highest level of quality as much as possible. In this work, our main strategy is to establish the most significant scene aspects to the viewer experience, in other words, stipulate the portion of the scene that causes the most visual impact and render it with an appropriate level of detail, leaving the rest of the scene to be rendered within the time budget to maintain an acceptable frame rate.

This paper is organized as following. In Section 2, we introduce other approaches of hybrid renderers that inspired or contributed with our technique development. After that, in Section 3, we detail the architecture of the framework used in our application. In Section 4, we comment our method to evaluate the final scene in a form of a new heuristic, and the way it can improve the framework into a more robust system. Section5 discusses implementation aspects. In Section 6, we present some results data, and, finally, in Section 7, we present final remarks.

2 Related Work

Beck [Beck et al. 2005] proposes a ray tracing framework for Real-time applications that divides the traditional stages of a conventional ray tracing algorithm in independent tasks scheduled between the available GPU or CPU processing cores. These tasks can be understood as three simple passes from a classical GPU pipeline: the first one consists in generating a shadow map; the second, an algorithm to identify the geometry of a 3D object; and the last one, a blur filter application pass. In the geometry identification pass, the triangle indices are written in a frame buffer coded as a RGB map, and the result of the shadow map from the first pass is blurred in the alpha channel. Then, the last pass includes a shading technique, Phong shading for example, putting all the results so far together.

Chen [che] presented a hybrid ray tracer as well, utilizing both CPU and GPU. In his method, at first, a rasterization pass using Z-buffer is executed in order to identify which triangles are visible in the moment the first light rays are shot from the camera and collide to the scene. After that, the CPU, retrieving the intersection points already computed, is able to shoot the secondary rays to achieve the photorealistic effects expected from the ray tracing. Chen's work also proposes to move the shading operations to the GPU, so that the bottleneck effect from the constant data transfers between the memories can be reduced.

NVIDIA OptiX [Parker et al. 2010] is a general purpose ray tracing engine capable of not only exploiting the NVIDIA graphic processor units, but also other general purpose processors. It architecture offers the possibility of programming a C/C++ and CUDA based language script in each ray, giving a lot of flexibility to the engine as the user can control how the rays are going to be generated and launched. It also provides an easy-to-use well defined structure for storing the input scenarios, which can be accelerated by using numerous data structures. The results acquired from our last experiences with OptiX were very promising, driving the adoption of the engine in our current work.

Making use of OptiX engine, Sabino et al [Sabino et al. 2012] presented a hybrid pipeline model for ray tracing in real-time applications applying the Deferred Rendering technique at first in order to rasterize the whole scene, eliminating the need to ray trace all the elements of the scene without any major lightning effect. Thus, the ray tracer module would only be responsible to add the photorealistic aspect of the scene, i.e., shading, reflection, refraction, among others, and only after the main drawing. Besides that, as well as Chen's contribution, they were able to resolve the first series of collisions between objects and the rays shot from the camera in the raster process, optimizing the second stage of the algorithm. Finally, the proposed model has been chosen as a framework for this work development given the possibilities it showed up for improvement in the interactive rendering area. The objective of this work is to develop these previous techniques, making a more robust system sensitive to a whole new range of possible scenarios, and even able to adapt to them, delivering the best possible final image.

3 The Hybrid Ray Tracing Architecture

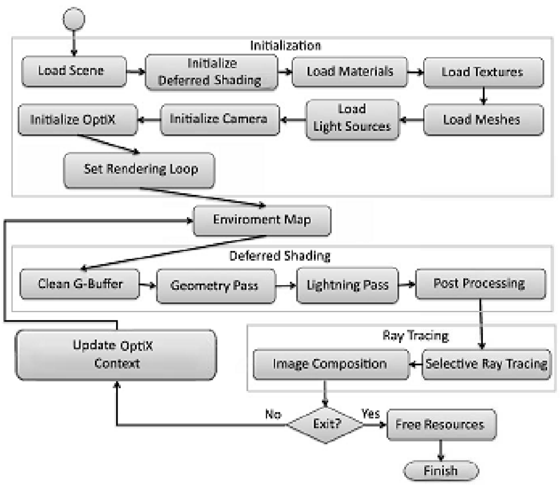

Figure 2: *The architecture of our pipeline.*

As introduced in the previous sections, many works propose the use of hybrid ray tracing techniques, sharing the computational load to generate the image between CPU and GPU cores in order to reach de maximum performance in both sides of the spectrum. Using modern graphic processing units and multiple render target techniques in the form of the deferred shading, we are able to take advantage of the traditional rasterization pipeline as the first stage of the ray tracing process, accelerating the synthesis of the final image by reducing the amount of processing needed to achieve the desired photorealistic result. This work has been developed based on this hybrid pipeline model, which had recently presented promising results in the field.

Although the deferred shading and the ray tracing stages are cov-

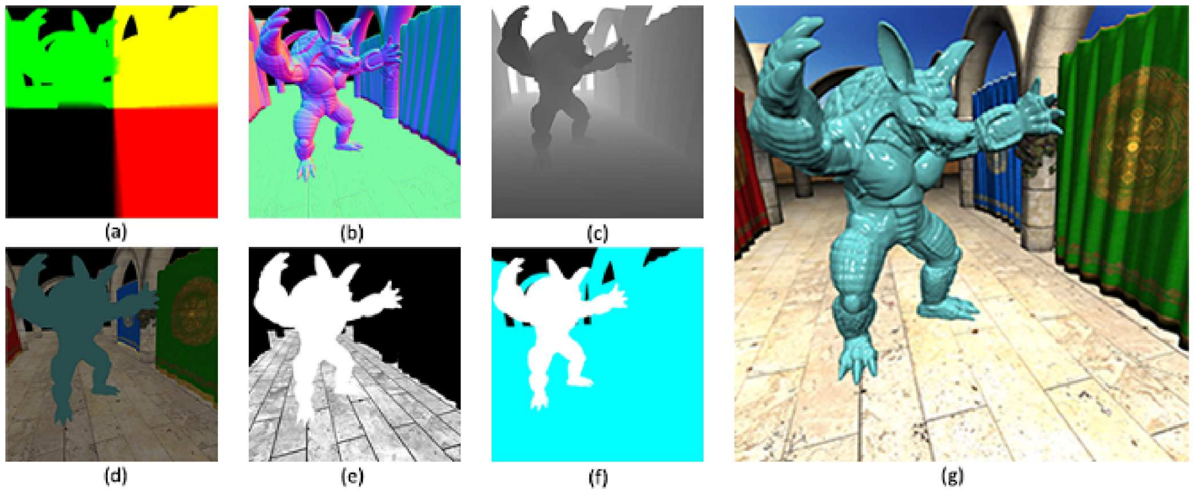

Figure 3: *The six stages generated by the geometry pass: (a) position, (b) normal, (c) z-depth, (d) diffuse or albedo color, (e) specular color, (f) optical properties. (g) is the final rasterized image after the lightning stage.*

ered in the next two subsections, in this work we will not discuss the initialization phase in details. This phase is basically the first step of our pipeline architecture, responsible for loading the scene from its file, as well as the materials, meshes for the models, textures and light sources. It is also responsible for setting up the camera, the data structures related to the Deferred Shader, the OptiX engine and finally the application rendering loop. Addition information is provided in [Sabino and Clua 2012].

3.1 The Deferred Shading

In the traditional GPU rendering pipeline, the Z-Buffering technique is very usual, managing the depth of the scene objects in order to eliminate the overlap between them. Although it have an extensive use, this technique can be inefficient: as Z-buffer normally compute the shading in the moment of a fragment generation, and a single pixel often generates more than one fragment, for each one of them that are not visible, this computing is wasted. It's stated that ordering the scene objects can reduce this issue, but the use of the deferred shading is able to fully eliminate it.

The concept of Deferred Shading was first introduced by Deering [Deering et al. 1988], even if the term have been adopted after that, when the technique became extensively used in real-time rendering applications, mainly video-games.

The technique introduces a key concept: to compute all the shading of a pixel and store the results in an intermediary frame buffer (named G-buffer), instead of writing the immediate result directly in the color buffer. The advantage this method comes by executing all the visibility tests before applying any shading effect on the fragments. The idea is not new, but became viable with the capacity of the GPUs to write in multiple render targets at once.

As mentioned above, deferred shading takes advantage of a structure called G-buffer [Saito and Takahashi 1990] responsible to store the geometric data from the scene, using a so called geometry pass. The values saved into G-buffer in this framework are the following for each render target: position, normal, z-depth, albedo and specular color. This information is used in the lighting stage of the deferred shading in order to create the first layer of rendering (see Figure 2). Although this information is enough to the first render of the scene, as well as enough to determine the origin and direction of shadow rays in the ray tracing pass, it lacks the data needed to compose the reflection and refraction rays. So, one of the contribution of Sabino et al works, also adopted for us was the extension of the G-buffer with an extra render that stores optical properties of a pixel, specifically: reflection and refraction indices; opacity and shininess, in the form of the specular power.

It is important to say that deferred shading does have its drawbacks. As easily perceived, the video memory requirements and fill rate costs for the G-buffers are significant. Besides that, the technique presents some substantial limitations in aliasing and transparency. Even though, these disadvantages cannot be completely ignored, they can be minimized or bypassed applying on the next step of the proposed pipeline, the Post Processing phase, with antialiasing image-based techniques such Shishkovtov edge-detection method [Shishkovtov 2005] and, more recently, Morphological Antialiasing [Reshetov 2009] for better image quality.

3.2 The Ray Tracing Illumination Effects

In the ray tracing stage, the secondary rays — the ones generated from the intersection point between the camera-shot rays and the objects of the scene — are responsible for creating the global illumination effects lacking from the rasterization so far.

After the primary rays are launched of the camera, shadow rays are generated from the collision points with the objects to the light sources. To verify if a certain point is shadowed, the algorithm check if that specific ray coming out of it reaches a light source colliding with any other object in the scene on its path. At this point, we have a fully rasterized scene which shadows are being generated by the ray tracing technique, adding a more polished and realistic effect than real time shadow map raster implementations. Tracing shadow rays adds a significant computational cost to this process, but as this is done in GPU, it is preferable to increase the amount of computation than use additional memory for shadow mapping, for example.

With that done, the algorithm execute as some of the conventional ray tracers. Other secondary rays are shot from the collision points, obtained through the rasterization step, each one in its corresponding direction in order to produce the global illumination effects like reflections and refractions.

Basically, at this time, our pipeline can be roughly summarized in four different image stages: initially, the GPU rasterization of the scene with the primary rays resolution (Fig. 4a) with the edge-detection antialiasing algorithm applied; then, the shadow pass, where shadows rays are generated based on the information extracted from the G-buffer (Fig. 4b); after that, the reflection and refraction synthesis (Fig. 4c); and finally, the final composing of the scene. The result of this process is stored in an image layer that is blended to the rasterized image to add the proper effects expected from the ray tracing (Fig. 4d).

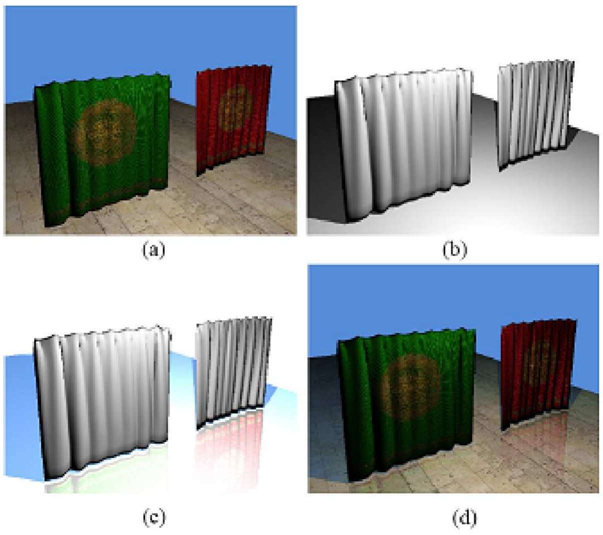

Figure 4: *In (a) the result of the deferred shading phase. In (b), the shadow rays generated by the ray tracing module. Image (c) shows the reflection and refraction computed by other secondary rays, while image (d) shows the final composition of the scene.*

3.3 Composing the Final Scene

In order to compose the final image to be displayed, a blending operation need to be done, adding the ray tracing generated layer to the rasterized image. At this point, the frame buffer stores both processes' result and the simplest way to generate the final image is to superimpose the ray traced image on to the rasterized one. The weights to this blend are relative to the amount of ray tracer effects desired on the final image.

With the filled frame buffer and the desired settings to the blend, the operation can be easily accomplished with a shader program. Applying the shader represents an additional light weight pass in the rendering pipeline, since we only need to render a screen-size (in worst case) quadrilateral primitive.

In the first version of the framework, this stage came down to in a simple and direct way in overlaying the ray traced image on to the rasterized one, as both images presented the same dimensions and their pixels were one by one equivalent. The next section introduces a technique which aims to limit the ray tracing module domain, in order to fit the time restrictions in a better fashion. Thus, the blending operation have shown some more complex computation requirement, having the need to compose two different images into one, matching the appropriate illumination effects to their exact place in the final image.

4 A Heuristic Approach to Ray Trace over a Rasterized Image

In order to improve the performance of the hybrid ray tracer for real-time applications, we propose a heuristic to dynamically select a portion of the scene containing the most relevant objects to trace at that time. The main objective of the heuristic is to provide the best possible visual experience to the observer still maintaining the expected frame rate.

4.1 Improving the Visual Impact

Selecting a set of specific objects in a three dimensional scene is not often a trivial task. Usually, the application aims to render in more detail the most distinctive objects that are more relevant to the visual impact of the final image. This strategy is generally justified by the physical inability of the observer to focus attention in the whole scenario at once. Thus, given a scenario where the application is not able to render every element at full capacity, the direct solution is to select only some of the elements to be rendered with full quality, leaving the rest of them with a pre set minimum quality within the time constraints.

The idea of tracing rays in just a specific set of objects is originated analyzing the human being behavior in real world situations. In this case, we are often exposed to an image that constantly varies, for example when we are running or driving, so that the human brain subconsciously ignores a lot of elements merely projecting to us some of its main characteristics. Although this technique presents suitable results to some circumstances, it can be quite disadvantageous in situations where it is capable to select a specific object to be ray traced rather than an adjacent one. Doing so, it draws attention to that section of the rendered image, exposing the often very noticeable differences between the objects, some favored by the heuristics selection and some not, right next to them. In that way, the photorealism achieved though the application of the ray tracing algorithm in some objects of the scene, instead of enhancing the observer immersion factor, has the opposite effect. This can be a crucial issue in some applications, like electronic games, where immersion is a key aspect.

Besides that, in real world, the observer's focus region is not limited by the edge that divides a specific object from the rest of the scene. The attention to the detail of the objects decays smoothly as the distance from the focal point to the rest of the scene increases. In this work, in order to maintain the system running within the time constraints common to interactive applications, we have created a simple heuristic which can be roughly summarized as *"trace rays in the area around the observer's focus of attention until the time budget to render this frame runs out"*. Figure 5 shows an example of an image that could not be rendered in full detail within the time destined to that frame.

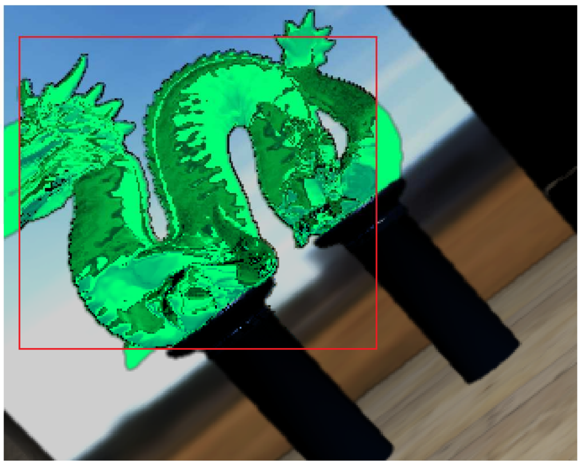

Figure 5: *A image that could not be fully ray traced in the time budget. The focus of attention area is delimited on the figure.*

At this stage, our main objective is no longer to maximize the quality of the generated image. Instead, it comes to be to maximize the final visual experience of the observer, without violating the time constraints associated to real-time aspect of the application. In order to simulate the observer's focus area, we used the cursor position on the screen at that moment to approximate the region where the viewer is probably looking at. Even though in some applications the attention of the observer and the cursor position are dissociated, there are a large amount of situations where this approximation is somewhat valid. From the position of the cursor in screen-space coordinates, a ray-traced image is generated based on that three dimensional section of that environment. After that, in the composition of the final image, the system has to assure that the appropriate illumination effects from that region of the space are blended in its equivalent 2D section of the screen. At first, the size of the attention focus area of the system must be optimized in an empirical way, based on the time budget to render each frame and it may be increased until some glitching in the execution is detected, i.e., the application is not able to run smoothly.

This have been done altering the shader program responsible for the composition of the final image, feeding it not only with the previous input data, but also the cursor screen-space coordinates ate each frame of the application, so that we could perform a bias operation. This operation permits us to do the appropriate fitting of the ray-traced lightning effects on the exact equivalent section of the rasterized one. As both images may no longer be equivalent in terms of size, some arrangements must have been taken care of: at first, we have to map the ray-traced layer on to a screen-aligned quad, and only after that perform the blending; the OptiX context size, that can be understood as the amount of rays to be traced pixel to pixel, is also no longer equivalent to the dimensions of the screen; the program responsible to generate these rays from the camera have to consider the cursor parameters in order to calculate the new position from where they will be shot; the render buffer which will store the output data from the ray tracing pass; and finally, the OpenGL-context equivalent texture that will act as the before screen sized layer to be composed to the rasterized image.

4.2 The Adaptive Character of the System

During the development of this work, we aimed not only to propose a new way of facing the problems from real-time rendering with the best possible quality, but also investigate ways of making our system more robust, sensitive to environmental changes. In addition to being a fair response to the issues raised in the previous subsection, the heuristic approach proposed in this paper allow us to apply an adaptive character to the system.

As mentioned before, the size of the region of the scene to be ray traced must fit the time budget to render that frame, so that the sequence of frame by frame rendering can run smoothly. At first, the parameters of the ray tracing rendering area must be settled in an empirical way, so that the system can estimate a reasonable size for most situations within that scenario. So, respecting the proposed heuristic, we have a specific region of the environment being rendered at full potent ion, depending on both cursor position which center the observer focus of attention; and the complexity of the scene itself, since rendering a much complex set of objects would last longer and then result in a smaller ray-traced area. Therefore, the system is capable of render a larger window in full detail where there are fewer complex objects, and a smaller one otherwise. In order to assure the fluidity of the system, we must take into consideration the worst case scenario, which would lead the system into a waste of time budget in many frequent medium-case situations, where the cursor goes over some simpler regions of the scene.

Considering this scenario, the adaptive character of the system comes directly. Within the time constraints related to the desired frame per second rate, the system can compute the minimum size of the ray tracing rendering window, and from that, increase its size on situations in which the engine is not overloaded with complex objects, requiring elaborate illumination effects. Taking advantage of the fact that the current focal point, i.e., the center of the ray-traced area, is relatively close to the previous one in the last frame, it is not difficult to estimate the area that could be rendered within the available time budget. Having the previous frame data, including the attention focus position, time budget, time elapsed in the ray tracing pass, this operation can be done without adding any major load to the system.

Note that the process goes both ways: as it can increase the level of detail of the scene in a specific region, it also adapts to situations where the attention comes to a more complex render area, generating and shooting a lesser amount of rays from the camera and thus, creating a smaller high detail window. The opposed processes alternate through the execution, as the observer focus on simpler or more complex areas of the scene.

In its current status, this adaptive property of the system can be faced as a prototype, as it still presents a considerable room for improvement as commented in a more detailed fashion in the last section of this article. Nevertheless, the results obtained in experimentation of this technique motivate us for further investigation.

5 Implementation Aspects

In this section, some implementation details are discussed. As aforementioned, this work has been developed based on the hybrid raster ray-tracing pipeline framework proposed by [Sabino and Clua 2012] and every modification in its architecture and operational behavior were made concerning efficiency and robustness.

Our system is implemented using C++ as the main language. The Open Asset Import Library [ASSIMP] was used in order to read common scene file formats. The Free Image Project [FREEIMAGE] was used for read texture images. We use GLSL as the shading language to handle the primary rays and for rendering purposes.

Due to its practicability in terms of freedom on building the ray tracing pipeline and hiding GPU hardware details when writing ray tracing shaders, the NVIDIA OptiX engine was adopted in our work to trace and shade secondary rays. Besides the advantages of being a generic ray tracer engine, its bond to OpenGL for reading and writing graphic resources is straightforward.

Mainstream GPUs has a limited amount of memory when compared to current main memory sizes. It is usually a relevant aspect to take into account, as while standard personal computers have access to 12 GB of RAM, a commercial GPU usually have around 1 or 2 GB of memory. Thus, a crucial factor for the development of this work was the intelligent use of these limited resources, mainly for rendering large scenes with a great number of materials and textures. Our first concern was about texture management, so that the system was built assuring that all the objects and materials related data would be available both in GLSL and OptiX contexts. This can be done by loading all the textures only once and keeping track of their references.

We also use vertex buffer objects (VBOs) for data transferring between CPU and GPU cores. This is the best option to render large amounts of data, since they are already loaded in the GPU memory. In order to access them, we need the VBOs available in two contexts. The first one is the OpenGL rendering context, and the second is related to ray tracing related programs inside OptiX engine. The consistency of the data is necessary because when tracing the secondary rays, the engine needs the whole scene geometry information in order to calculate the further illumination effects. Using vertex buffer objects not only brings the advantage of being a fast option for data transfer, but also avoids the same information to be duplicated in the application. As OpenGL and OptiX are in different contexts, creating a VBO inside the OpenGL and registering it to use in OptiX eliminate this problem, making every data modification in one context automatic to the other one.

As mentioned before, the workflow of the system can be understood in three main steps: the initialization, where the scene, the OptiX and the application main loop are set up; the deferred shading, where the first layer of the scene is rendered based on the information of the G-buffer; and finally ray tracing and image composition, where the secondary lightning effects are added to the scene. After that, if the system is not able to fulfill the real-time related constraints, some adaptive model can be executed to assure a lag-free performance.

In the deferred shading, the geometry stage implements a perspective projection camera, which is responsible to feed the G-buffer with all the information used in the next stages. All the following stages operate in image-space with an orthographic projection camera using screen resolution dimensions. The lightning stage the contents of the G-Buffer as input, together with the scene's light sources information and accumulates lightning into a full resolution pixel buffer.

The ray tracing phase is controlled by the shader programs in the OptiX engine. At this time, the OptiX context has been already instantiated, with the proper parameters, regarding the number of rays to be generated as well as how they will be casted. The shader programs involved in this process were created using an OptiX specific shading language, based on C++ and CUDA. They are several and each responsible for a different purpose. The *ray generation program* is the main program of the selective ray tracing stage, being the responsible for generated an application-given number of rays, and respecting the heuristic approach proposed for us, for casting them around the right section of the image calculating the bias from the origin based on the cursor coordinates. The other shader programs execute depending on the trajectory of the primary rays: the *any hit program* is responsible for the lightning attenuation; the *closest hit*, for tracing the secondary rays from the collision point with the objects; the *shadow ray generation program* which traces shadow rays from the first collision, are some examples. The *bounding box program*, the *exception program* and the other auxiliary shaders used to complete the ray tracing pipeline are discussed in detail in [Sabino and Clua 2012].

The output of the OptiX engine is stored in a render buffer, accessible from the OpenGL context. In the image composition stage, the OpenGL saves the information on a texture, rendered in a screen-aligned quad. After that, a GLSL shader program calculates the proper area of the scene for the blending, using the same parameters given to the OptiX context. Finally, the blending operation is performed, generating the final image.

6 Results

In this section, we present some data acquired by experimentation of the system we have described in this work. The tests are performed in a desktop equipped with an AMD Phenom II X4 965 3.4GHz, 16GB of RAM and a NVIDIA GeForce GTX570.

Table 1 exhibits the performance of our hybrid ray tracing pipeline in five different scenes. In this table, DS is the acronym for deferred shader, representing the frame per second rate when rendering the first pass alone. The RT column, stands for ray tracing, where are given the FPS value for a pure ray tracing approach. The last column refers to the values obtained using our hybrid ray-tracing proposal.

Table 1: *Performance Results*

			Performance in FPS		
Scene	Vertices	Triangles	DS	RT	HRT
Sponza (Fig. 6)	145.173	262.187	449	15	29
Sponza Animated (Fig. 7)	184.178	266.923	388	8	18
Showcase (Fig. 1a)	112.603	224.440	428	25	43
Armadillo (Fig. 1b)	193.737	380.655	319	40	54
Dining Room (Fig. 8)	386.541	224.954	481	13	32

After these tests, the frame per second rate to all the scenes was fixed in 15 in order to analyze the behavior of the system in stressing conditions. In the first scenario, the system was able to fully render about 51.7% of the total screen at once. The second scene reached over 83% of full rendering area, as the third and fourth ones, being more complex scenarios, presented respectively 34.8% and 27.8% on average. Finally, the area that could be ray traced as the attention focus in the last scene was 47% of the total screen.

These results reflects a metric used in this paper to illustrate the behavior of our heuristic approach in the proposed framework. That being said, one must perceive these values are approximations and can change since they relate to the portion of the scene sent to the ray tracing module at each different frame.

7 Conclusion and Future Work

In this work, we introduced the current context of real time ray tracing application and hybrid rendering systems. While most of these works focus on the workload division and the graphic pipeline task scheduling between the CPU and GPU cores, we continued investigating a proposal of dividing that graphic pipeline in a more intelligent fashion: accelerating the first batch of ray tracing through the use of the deferred shading, a hardware-implemented rasterization technique.

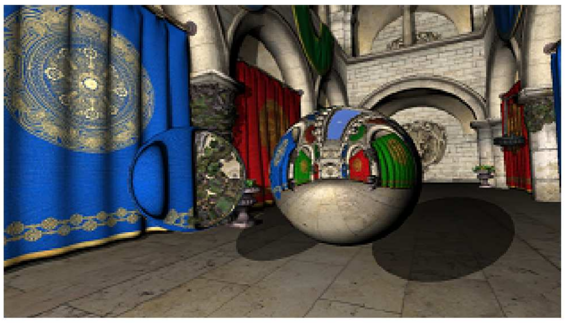
Figure 6: *Sponza scene.*

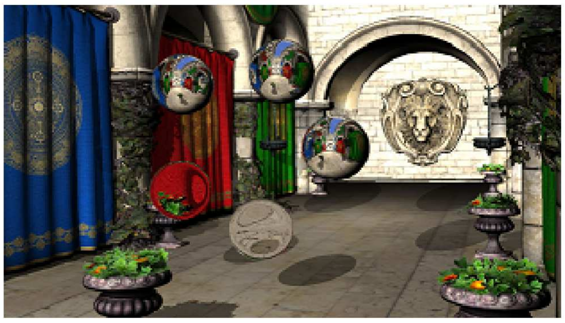
Figure 7: *Sponza animated scene.*

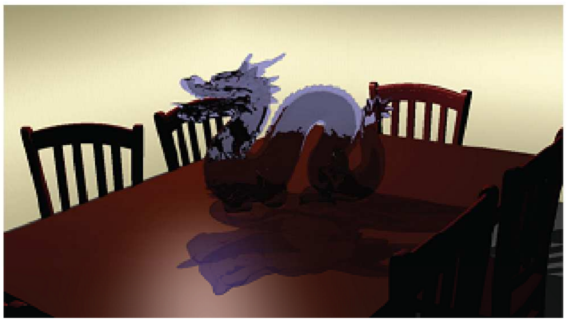
Figure 8: *Dining room scene.*

We were able to identify some of the most common hybrid system limitations, mainly according to highly complex input scenarios, and suggested a new approach to them in order to create a more robust system instead of a simply faster one. In this fashion, we would end up with a functional system to every possible instance, even if it would not be able to maintain its full render quality in extreme situations. We analyzed scenarios that were not been comprehended by the most commonly applied heuristics in hybrid systems, and after that, proposed a new yet natural approach, based on more suitable point of view. We have investigated the final visual impact of our proposal based on the observer experience and found some promising results.

Finally, this work proposes an adaptive approach to extend the static results of the previous framework into a more robust system, sensitive to the input data and able to respond to it, increasing the photorealism at the observer main focus of attention area within the disposable time budget for that frame. Likewise, in an opposite way, where the system struggles to fit into a smooth interactive scenario, it is able to decrease the amount of ray tracing computation in order to overcome these situations.

As future work, we intend to implement a new ray tracing-based layer in the final image composition, in this time with a lower granularity, so that it can work as a mid-way interface between both sides of the spectrum: the fast rasterized and low detail image; and photorealistic higher-cost ray-traced one, often limited to a specific area by the time constraints. Doing so, we believe that the final result can achieve an even more pleasant high-quality visual experience to the observer, as this new layer would soften the transition between the other two. Besides that, we encourage the implementation of a raster based shadow generation technique, because as the current system treat the problem in the ray-tracing pass, in situations where the time constraint limit this pass domain to only a certain section of the final image, the remaining of it stays without shadow support. Even though, we achieved some computational gain in the early stages of the system development, the absence of shadows in some parts of the scene can be a relevant issue when the system cannot complete its full ray tracing pass.

Another interesting research branch from this work is to investigate the different ways that the focus of attention can be estimated. We stated that in general, focusing the attention around the cursor can be a good approximation, but many other approaches may comprehend situations where this first approach lacks accuracy. The integration of an intelligent eye tracking system can completely eliminate this issue, as well as consolidate the whole system.

As we can see, the use of hybrid renderers opens a whole new sort of possibilities, in addition to present a very satisfactory final result, not only in photorealistic terms, but also in real-time computation paradigm. The main objective of this work was to investigate the area, going further than simple parallel optimization and scheduling techniques, facing the problem through a new approach and opening a new set of possibilities of research in the area.

Acknowledgements

Felipe Martinez is thankful to CNPq for the financial support. Esteban Clua is thankful to FAPERJ and to NVIDIA for financial support. Paulo Pagliosa is thankful to FUNDECT for financial support. We also thank all anonymous reviewers that help us to improve the quality of this paper with its valuable comments.

References

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

ASSIMP. Open asset import library. Webpage, 10 2011. [Online]. Available: `http://assimp.sourceforge.net/`.

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

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

BIGLER, J., STEPHENS, A., AND PARKER, S. 2006. Design for parallel interactive ray tracing systems. *Interactive Ray Tracing IEEE Symposium* (september), 187–196.

BIKKER, J. 2007. Real-time ray tracing through the eyes of a game developer. *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing. Washington, DC, USA: IEEE Computer Society [Online]. Available: `http://portal.acm.org/citation.cfm?id=1524874.1524949`*, 1–10.

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 [online]. available: `http://doi.acm.org/10.1145/378456.378468`. 21–30.

FREEIMAGE. The free image project. Webpage, 10 2011. [Online]. Available: `http://freeimage.sourceforge.net/`.

HACHISUKA, T. 2009. *Ray Tracing on Graphics Hardware*. Master's thesis, University of California at San Diego Tech. Rep.

HEIRICH, A., AND ARVO, J. 1998. A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing [Online]. Available: http://dx.doi.org/10.1023/A:1007977326603 12*, 57–68.

NVIDIA, 2007. Nvidia cuda compute unified device architecture. Online: Available at `http://developer.download.nvidia.com/`.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., COLLEGE, W., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics 29*, 66 (July).

RESHETOV, A. 2009. Morphological antialiasing. *Proceedings of the 2009 ACM Symposium on High Performance Graphics*, 109–116.

SABINO, T., AND CLUA, E. 2012. *Uma Arquitetura de Pipeline Hbrida para Rasterização e Traçado de Raios em Tempo Real*. Master's thesis, Universidade Federal Fluminense.

SABINO, T., ANDRADE, P., CLUA, E., MONTENEGRO, A., AND PAGLIOSA, P. 2012. A hybrid gpu rasterized and ray traced rendering pipeline for real time rendering of per pixel effects. *ICEC 2012 (accepted for publication)*.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. [Online]. Available: `http://doi.acm.org/10.1145/97880.97901` 24*, 4 (September), 197–206.

SHISHKOVTOV, O. 2005. Deffered shading in s.t.a.l.k.e.r. *GPU Gems 2 2*, 143–166.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM [Online]. Available: http://doi.acm.org/10.1145/358876.358882 23* (June), 343–349.