Improving the Performance of the Ray Tracing Algorithm with a GPU

Santiago Cioli*, Gonzalo Ordeix†, Eduardo Fernández‡, Martín Pedemonte§, Pablo Ezzatti¶
Instituto de Computación, Facultad de Ingeniería,
Universidad de la República
Montevideo, Uruguay
*Email: *sancioli@gmail.com, †gonzalo.ordeix@gmail.com*
‡eduardof@fing.edu.uy, §mpedemon@fing.edu.uy, ¶pezzatti@fing.edu.uy

***Abstract*—This article presents the application of parallel computing techniques using a Graphic Processing Unit (GPU) in order to improve the computational efficiency of the ray tracing algorithm. In particular, three different GPU implementations of the ray tracing algorithm are presented. The experimental evaluation of the proposed methods demonstrates that a significant reduction on the computing time can be attained when compared with a full CPU implementation, making a step forward to the calculation of scene brightness in real time on desktop computers.**

***Keywords*-Ray tracing; GPU; Real-time**

I. Introduction

A scene is a collection of objects and light sources that are seen through a camera. Each object in a scene is a geometric primitive, which is usually a simple geometric shape like a polygon, a sphere or a bicubic surface (Hermite, Bézier, Spline). Additionally, the surface of the object has material properties, textures, etc. All global illumination techniques try to solve the same problem consisting in, given a scene, finding a set of images as photorealistic as possible. These algorithms usually differ in how they handle the lighting of the scene.

Several kind of global illumination algorithms can be identified, based on the different light elements considered. Radiosity [11], *ray tracing* [25] and multipass methods (like RADIANCE [24] and photon mapping [13]) produce realistic images in diverse scenarios with several kind of surfaces. Radiosity works well in scenes with Lambertian surfaces, *ray tracing* produces good images in scenes with specular surfaces and the multipass methods are more versatile based on the mixture of methodologies used in them. Developing an algorithm that generates at least twenty images per second -that is, in real time- is a great challenge for the computer graphics research community.

Ray tracing algorithm calculates the brightness of each pixel of an image by throwing rays, and evaluating their bounces in the different surfaces of the image. Each bounce produce new rays that impact in other objects and so on. The color of the pixel is composed with the color of the first object considered plus the color added with each bounce. This algorithm was the first step into photorealistic rendering, and its success is due to its ability to generate good quality images with an algorithm easy to implement.

Graphics Processing Units (GPUs) are devices designed originally for graphics processing only, lightening the workload on the CPU in applications such as video games. Thus, the CPU can be used to perform other computations while most of the graphic processing calculations are performed on the graphic device. GPUs are currently very powerful platforms, provided by tens or hundreds of cores with acceptable clock frequencies (500-600MHz). Additionally, the computing power of GPUs is enhanced due to its intrinsically parallel architecture.

Initially, the progress in the design of GPUs was not accompanied by an advance in the software for programming these devices until 2006, when NVIDIA released CUDA (Compute Unified Device Architecture) [7]. CUDA is an architecture for general purpose parallel computing that allows using the parallel processing core in these devices to solve a wide variety of computational problems more efficiently than a CPU. The *ray tracing* algorithm is highly parallelizable since the calculation of lighting in each pixel is an independent process, and therefore is very suitable for GPUs.

This article studies a *ray tracing* algorithm implemented in GPU for speeding up the computation time. Three parallel versions were developed, in order to exploit different characteristics of the *ray tracing* algorithm and GPU architecture. An analysis of the performance was conducted, measuring the number of frames that could be calculated per second. The preliminary results show that large improvements could be obtained (up to $13\times$) when using a GPU cheaper than a standard multicore computer, such as the ones used in this analysis.

The content of the article is structured as follows. The next section describes the main features of modern GPUs and CUDA architecture. Then, Section III introduces the *ray tracing* algorithm. Section IV describes the three different GPU implementations of *ray tracing* presented in this article. The experimental evaluation of the proposed methods is reported in Section V, where the results are also analyzed. Finally, Section VI presents the conclusions of this research and formulates the main lines for future work.

II. GRAPHIC PROCESSING UNITS

Based on the facilities provided by CUDA [7] for GPU programming, GPUs can be viewed as a set of shared memory multicore processors. GPUs are usually considered *many-cores* processors due to the large number of small cores that contain. GPUs follow the single-program multiple-data (SPMD) parallel programming paradigm in which cores execute the same program on multiple parts of the data, but do not have to be executing the same instruction at the same time [8]. The number of threads that currently graphics card can execute in parallel is in the order of hundreds and is expected to continue increasing rapidly, what makes these devices a powerful and low cost platform for implementing parallel algorithms.

CUDA [15] consists of a stack of software layers including: a hardware driver, a C language application programming interface and the CUDA driver that is dedicated to transfer data between the GPU and CPU. It is available for NVIDIA's GeForce 8 series of graphics cards and superiors. It is compatible with operative systems Linux 32/64 bit and Windows XP and superiors of 32/64 bits.

CUDA architecture is built around a scalable multiprocessor array. Each multiprocessor on current GPUs, consists of eight scalar processors as well as additional units like the multithreading instruction unit and the shared memory chip. When a part of an application could be run many times but independently on different data, it can be isolated in a function, called kernel function, to be executed on the device through many different threads. For this purpose, the kernel function is compiled using the device instruction set and the resulting program is transferred to the device.

When a kernel function is called, a large number of threads are generated on the GPU. The group of all the threads generated by a kernel invocation is called a grid, which is partitioned in many blocks. The blocks group threads that are executed concurrently on a single multiprocessor of the card. There is no fixed order of execution between blocks. If there are enough multiprocessors available on the card, they are executed in parallel. Otherwise, a time-sharing strategy is used.

Threads can access data across multiple memory spaces during their execution. Nowadays, GPUs based on G80 architecture have six different memory spaces: registers, local memory, shared block memory, global memory, constant memory and texture memory. Table I presents the main features of the different GPU memory spaces that are briefly commented next.

Registers, that are located in the chip, are the fastest memory on the card and are only accessible by each thread. In addition to this, each thread has its own local memory but is one of the slowest memories on the card, because it is located in the device memory and is not cached. Both memory spaces are entirely managed by the compiler. Each block has a shared memory space that is almost as fast as registers and could be accessed by any thread of the block. The shared memory is located on the chip and its lifetime is equal to the lifetime of the block.

Table I
FEATURES OF THE DIFFERENT GPU MEMORY SPACES.

Memory	Scope	Lifetime	Size	Speed
Registers	Thread	Kernel	Very small	Very fast
Local	Thread	Kernel	Small	Slow
Shared	Block	Kernel	Very small	Very fast
Global	Grid	Application	Big	Slow
Constant	Grid	Application	Very small	Fast
Texture	Grid	Application	Very small	Fast

All the threads on the GPU have access to the same global memory on the card. The global memory is one of the slowest memories on the card and is not cached. On the other hand, constant memory is fast although for the device is read-only. It is located in the device memory even though it is cached. In fact, constant memory can be seen more as a cache of the global memory than a different memory space. Finally, the texture memory has the same characteristics that constant memory.

Figure 1 presents the CUDA architecture diagram, including the six different memory spaces. The figure shows local memory close to the threads as local memory is private to each thread. However, local memory is really located in the device memory as the global memory.

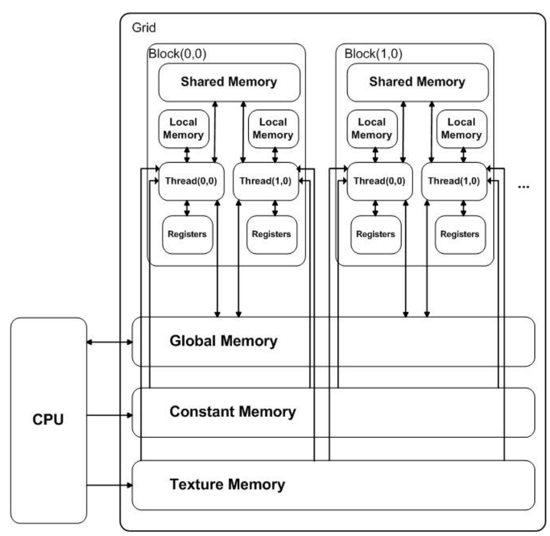


Figure 1. CUDA memory model.

III. RAY TRACING ALGORITHM

The ray casting algorithm [3] was proposed by A. Appel and is based in tracing rays from the observer's viewpoint to a view plane between the observer and the scene. The *ray tracing* algorithm [25] extends the idea of ray casting by making the ray tracing process recursive when the ray intersects an object in its way.

Ray tracing achieves a great realism in the images generated even though its implementation is quite simple. However, the simplifications used in the lighting model do not allow generating caustics caused by light rays reflected or refracted by curved surfaces. Similarly, the calculation of the color component of "ambient light" [14], that is a simplification in the lighting calculation, makes the algorithm unable to produce some effects like "color bleeding" (phenomenon caused by light reflection making the color of a wall spread over the floor area near the wall).

The *ray tracing* algorithm works as follows. For each pixel of the image, a ray is traced from the observer's viewpoint to the pixel (called primary ray). If a ray does not intersect an object in its way, then the pixel is painted with the background color of the scene. On the contrary, if the ray intersects with an object, the effect of the shadows, refraction and reflection are calculated. Figure 2 shows the rays generation of *ray tracing* in a scene with a single light source from a single primary ray.

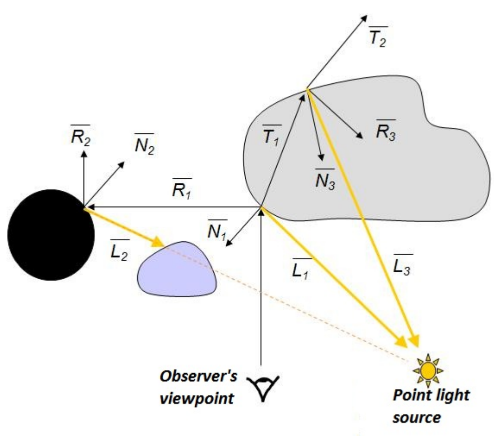


Figure 2. Ray generation from a single primary ray.

To calculate the shadows, a "shadow" ray ($\overline{L_1}$) is traced from the intersection point of the primary ray to each existing light source on the scene. If any of these rays intersects with any object in the scene, the amount of light that passes through the object is calculated, depending on the material of the object. If the object intersected is opaque (as the smallest object in the Figure 2), the intersection point of the primary ray is under the shadow of the object, so the light source is not considered to calculate the illumination at the point. If the object intersected is transparent (as the largest object in the Figure 2), the intensity of the light source is reduced, or may even be totally absorbed by the object. When the light is not completely blocked by the object, it contributes to the illumination of the intersection point of the primary ray.

When the object has specular reflection, a ray ($\overline{R_1}$, called reflection ray) is reflected from the primary ray at the point of intersection with respect to the normal ($\overline{N_1}$). This ray enables to get the intensity of the light that reaches the intersection point of the primary ray due to the phenomenon of reflection.

When the object is transparent and the light is not totally absorbed by the transparency of the object, a ray ($\overline{T_1}$, called refraction ray) is traced through the object. This ray enables to get the intensity of the light that reaches the intersection point of the primary ray due to the phenomenon of refraction.

Each one of the reflection and refraction rays when intersects with an object, can generate new "shadow", reflection and/or refraction rays. Therefore, the steps for the calculating the effect of refraction and reflection should be calculated recursively. For example, the same steps used to calculate the intensity of the light provided by the primary ray should be used to calculate the intensity of the light provided by $\overline{R_1}$. Thus, a ray tree is build for each primary ray, as shown in Figure 3.

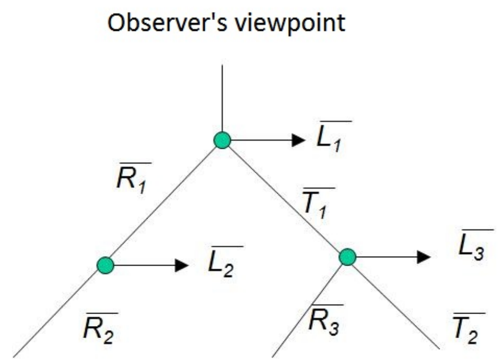


Figure 3. Ray tree resulting from the Figure 2.

A. Spatial acceleration structures: uniform spatial subdivision

Two families of strategies could be used to improve the performance of *ray tracing* algorithm; one which reduces the number of rays and another that optimizes the number of intersection checks performed. Space division of the scene by means of spatial acceleration structures helps to reduce the number of intersection checks, since it guarantees that the entire list of objects of the scene should not be checked for each ray.

In the space division method, the scene is divided into regions. Each region has a list of all the objects that it contains, either in whole or in part. This technique requires a preprocessing stage to create a data structure in which the position and space occupied by each object in the scene are stored. In the preprocessing stage, the total volume of the scene is divided in small volumes or boxes. The criteria used to define the boxes is what distinguishes the different spatial subdivision techniques, having all the techniques exactly the same behavior.

The main advantage of the space division method is that only the objects belonging to boxes traversed by the ray should be checked for possible intersections. As a result, and depending on the object distribution in the scene, it avoids doing many unnecessary calculations.

When the scene is divided in boxes of the same size, the technique is called uniform spatial subdivision. This strategy of space division can effectively accelerate the calculation of the intersections despite of being simple. While there are other alternatives like the kd-tree [23], the uniform spatial subdivision improves the performance of *ray tracing*, it is easy to implement and it does not add extra issues to the algorithm; therefore, it is a spatial acceleration structures interesting to be included in the implementation of *ray tracing* in a GPU.

B. Related work

The *ray tracing* algorithm has a high computational cost especially in the models used in most 3D applications, based on the rasterization of images formed by polygons. For this reason, *ray tracing* technique, until recent times, was not suitable for real-time applications. However, nowadays, some new works have achieved real-time *ray tracing* implementations over CPU architectures, such as a Quake Wars game engine [21] implemented using openRT [18]. A demo of the engine showed in August 2008 runs approximately between 20 - 35 FPS with an image resolution of 1024 by 720 pixels on a Caneland system that has four Dunnington with six cores.

On the other hand, in recent years, applying the computational resources delivered by modern GPUs to *ray tracing* has resulted in a number of implementations that allow rendering scenes in reasonable times. Researchers have introduced many techniques to speed up the construction of acceleration structures and the traversal of rays through an acceleration structure. Works, such as those done by Horn et al. [12], Popov et al. [20], Parket et al. [19], Aila and Laine [1] and the 3D engine developed by researchers of the Alexandra Institute [4].

The Horn et al. work is based on the use of Boundary Volume Hierarchy (BVH), such technique is not cover in our work. Meanwhile, the proposal of Aila and Lane is implemented using a combination of Brook [6] and Direct3D [9], involving a different conceptual abstraction of the GPU model. Parket et al. [19] have proposed a general framework to develop *ray tracing* algorithms, but their work is more focused on developing a flexible and adaptable framework than on the performance of the resulting algorithm. For these reasons, none of the three previous works were considered for the development of our proposal. On the other hand, the algorithm implemented by the *Alexandra Institute* is based on the traditional *ray tracing* algorithm whereas in the work of Popov et al. the kd-tree spatial acceleration structure is used, therefore both works are closely related with our approach.

A good survey of the state of the art in the *ray tracing* techniques can be found in the works of Parker et al. [19], and McGuire and Luebke [16].

IV. Our Proposal

The *ray tracing* algorithm is inherently suitable for parallelization with SPMD techniques, since the calculation of lighting in each pixel is an independent process. This feature makes possible its semi-direct implementation on graphics cards, using a separate thread to calculate each ray. Three different versions of the *ray tracing* algorithm were developed in order to exploit different characteristics of the algorithm and the GPU architecture. The versions were implemented following an incremental approach, incorporating in each version a considerable improvement over the previous one.

This section describes the main characteristics of the different versions implemented. First, a description of some general features of all versions implemented is introduced. Later, the differences between all versions implemented are detailed.

All versions were implemented using the C language and CUDA (version 2.3) to manage the GPU. The general structure of the different versions of the *ray tracing* implemented is presented in the Figure 4. It has five different steps that are discussed below.

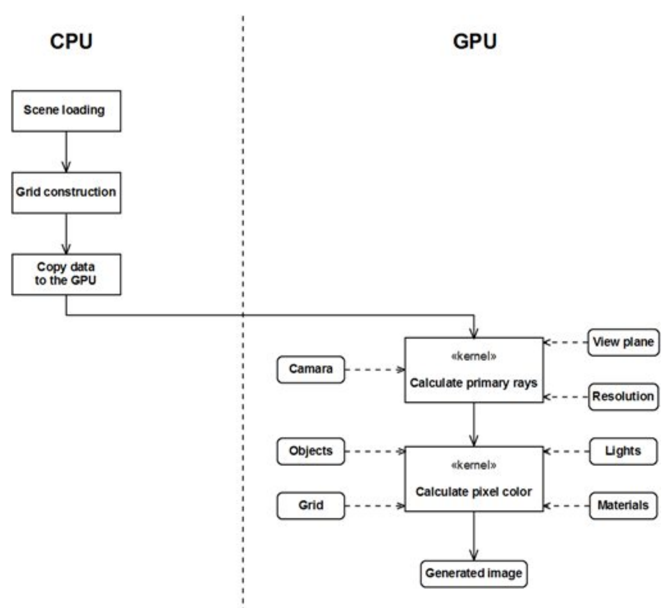


Figure 4. Structure of the *ray tracing* algorithm implemented in CUDA.

In the first step of the algorithm, the data of the scene is loaded from a text file. The file format is based on Wavefront OBJ (version 3.0) [17] that has been adopted by most 3D graphics application vendors. It makes possible to define the elements of the scene (e.g. vertices, points, lines, polygons, curves, etc.) and the materials of the elements. Also at this stage, a configuration file is loaded, containing parameters required to execute such as image resolution, division size for the acceleration grid, maximum number of ray bounces, etc.

In the second step, a spatial acceleration structure corresponding to the uniform space subdivision it is built, because of the simplicity of its construction and its traversal. The grid construction algorithm has been optimized, so that for each object in the scene, it is calculated (grid coordinates) an axis-aligned box that surrounds it. From the box, candidate boxes that may overlap with the object are obtained. For each candidate box, it is tested the box-object overlapping, and if it happens, the object is added to the box.

The third stage involves the transfer of data from the memory space of the CPU to the GPU. The transferred data are: the view plane, the camera, the image resolution, the list of triangles and its normals, the light sources, the boxes of the grid for spatial subdivision and the material of the objects.

After copying the data to the GPU, the kernel that calculates the primary rays is invoked. The data required to calculate the primary rays are the view plane, the camera and the image resolution.

The primary rays are input to the kernel which calculates the color of each pixel. This kernel implements the core of the *ray tracing* algorithm. The data required to calculate the color of each pixel are the list of triangles and its normals, the light sources, the boxes of the grid for spatial subdivision and the material of the objects.

The kernel that calculates the color of each pixel is invoked following a division in patches (group of pixels) of the image to render. The image is divided uniformly, having each patch the same number of pixels. Each patch corresponds directly to a block of threads in CUDA in order to process each division of the image by a different block. Moreover, since each pixel of the image is processed by a different thread, the number of threads per block is equal to the number of pixels contained in each division. For this reason, the division is completely established when fixing the number of threads per block and the image resolution. For example, if the image resolution is 640×480 pixels and the block size is 16×8 threads, the image must necessarily be divided into 40×60 patches.

Each ray threw traverse the spatial acceleration structure, following the reflections and refractions in the objects. The *ray tracing* algorithm implemented only has one type of elements, the triangle, therefore just a method of ray-triangle intersection was required. As it only works with a basic element, the intersection algorithm is simple and requires only a few arithmetic operations to test the intersection with a ray.

An important aspect is that the *ray tracing* algorithm is recursive, but current GPUs do not support recursion. As a consequence, the algorithm has to be implemented iteratively. There are two alternatives to achieve this, implementing a stack to store the recursion tree or simplifying the tree by making it degenerate into a list. The first alternative was ruled out because each thread must have its own stack and the size of the local memory is very limited. To degenerate the recursion tree into a list, it requires as a precondition that the scene has no objects that reflect and transmit light at the same time. This was the alternative implemented, since the limitation imposed on the scene is acceptable.

Finally, after the execution of the kernel that calculates the color of each pixel, the data generated in the GPU is copied to the CPU to be displayed on the screen.

The differences between the versions are discussed in the next subsections.

A. `RT(GPU)` *version*

The first version of *ray tracing* algorithm (`RT(GPU)`) is a GPU-analogue to the CPU implementation. In `RT(GPU)`, all the data is stored in global memory.

B. `RT(GPU-ml)` *version*

Regarding the GPU architecture, and in particular the importance of the correct use of the memory levels, this version (`RT(GPU-ml)`) uses the different memory levels of GPU accessible through CUDA. In particular, texture and constant memories are used, ensuring an improve in the performance.

The data that is most used by the algorithm, such as the list of triangles or the boxes of the grid for spatial subdivision, must be stored in a memory level with fast read access. For this reason, the list of triangles and its normals, the light sources, the boxes of the grid for spatial subdivision and the list of material of the objects are copied to the texture memory, since these data is accessed frequently and do not need updating. Other data such as the view plane, the camera and the image resolution is stored in the constant memory. Reading data from these types of memory is much faster than reading from global memory.

C. `RT(GPU-ii)` *version*

The third version (`RT(GPU-ii)`) improves the procedure for calculating the ray-triangle intersection using the barycentric coordinates method [2]. This method verifies that the ray intersects the plane containing the triangle and then by a change of coordinates verifies that the intersection point is within the limits of the triangle.

V. EXPERIMENTAL ANALYSIS

In this section, we present the test cases and hardware platforms used to evaluate the different versions implemented of the *ray tracing* algorithm. Then, we describe in detail the various experiments conducted to validate the proposal.

In addition to the GPU versions of the *ray tracing* algorithm described in Section IV, a CPU implementation of *ray tracing* `RT(CPU-ii)` was developed to evaluate the comparative performance versus the GPU versions.

A. Test cases

In a first instance, we studied the existing strategies for measuring the quality of the images generated. The survey did not obtain any comprehensive strategy. The choice of the method for measuring the quality of the image depends heavily on the objective of the study. Avcibas et al. [5] and Dirik et al. [10] present a good survey of strategies and discuss their limitations, but none is applicable for the purposes of this study.

In addition to this, there are no standardized test cases or benchmarks that could be used to evaluate the different implementations of the *ray tracing* implemented in this work. For this reason, a set of images were designed trying to cover several aspects of the image generation process, in order to contribute to measure different characteristics of the implemented algorithms. The test set of images designed is divided into three different groups. The first group consists of images that allow evaluating the effect of the distribution of objects in the scene. The test cases of the second group consists of images that allow evaluating the impact of the number of triangles in the scene. Although there are no benchmarks, some images have been sporadically used by the research community (such as the Bunny from Stanford University). Therefore, the third group includes some of those scenes and images that have been used in studies similar to this.

1) Test cases with different object distribution: All scenes have the same number of triangles but a different distribution in the scene. Table II presents the main features of the test cases considered.

2) Test cases with different number of primitives: All images in this group are exactly the same, but were discretized using a different number of primitives. Table III presents the main features of the test cases considered.

3) Test cases from similar studies: All images are taken from similar studies or are images commonly used by the research community. Table IV presents the main features of the test cases considered.

Table II
TEST CASES WITH DIFFERENT DISTRIBUTION IN THE SCENE.

Scene name	# Objects	# Lights	# Triangles	Figure
Dist_I	9	1	10,338	5(a)
Dist_II	9	1	10,338	5(b)
Dist_III	9	1	10,338	5(c)

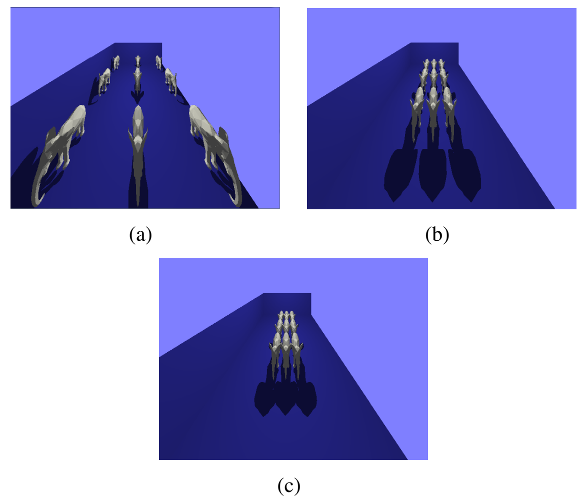
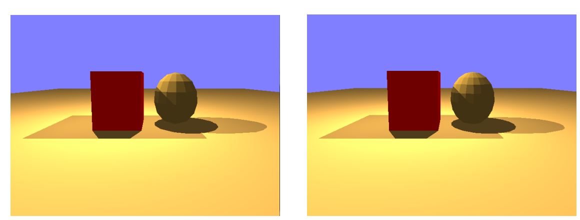


Figure 5. Scenes with different spatial distribution of the objects.

Table III
TEST CASES WITH DIFFERENT NUMBER OF PRIMITIVES.

Scene name	# Objects	# Lights	# Triangles	Figure
Pri_I	2	2	194	6(a)
Pri_II	2	2	274	N/S
Pri_III	2	2	348	N/S
Pri_IV	2	2	482	N/S
Pri_V	2	2	606	6(b)



(a) (b)

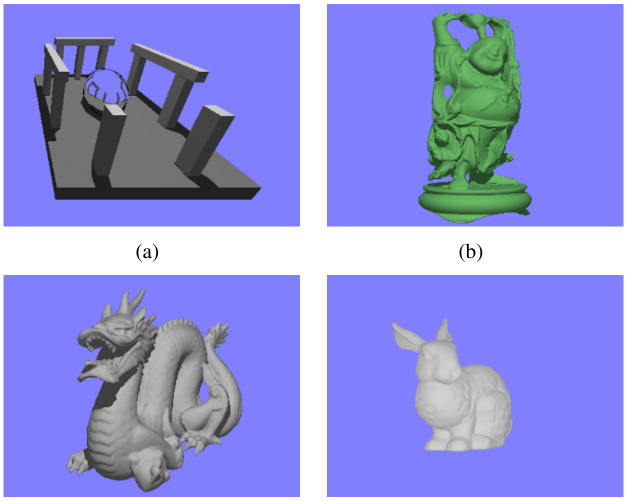
Figure 6. Scenes with a different number of primitives.

Table IV
TEST CASES USED FOR COMPARING WITH OTHER *ray tracing* IMPLEMENTATIONS.

Scene name	# Objects	# Lights	# Triangles	Figure
Alexandra	14	1	236	7(a)
Buddha	1	1	100,000	7(b)
Dragon	1	1	100,000	7(c)
Bunny	1	1	69,698	7(d)

B. Hardware platform

Several hardware platforms were employed to evaluate the implemented algorithms. Each platform consists of a PC



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

Figure 7. Scenes used for comparing with other *ray tracing* implementations.

Core 2 Duo with a GPU of the NVIDIA GeForce series. The main details of the hardware platforms used are presented in the Table V. All the PCs were running the Windows operating system.

Table V
HARDWARE PLATFORMS USED FOR EXPERIMENTAL ANALYSIS.

GPU	GPU memory	CPU	RAM memory
9500M GS	512 MB	T7500 2.20GHz	4GB DDR2 667 MHz
9600M GT	512 MB	P8400 2.26GHz	4GB DDR2 667 MHz
GTX 260	896 MB	E7500 2.93GHz	4GB DDR2 667 MHz

The Table VI provides more details of each one of the GPUs used during the evaluation.

Table VI
GPUS USED FOR EXPERIMENTAL ANALYSIS.

GPU	Multi processors	Cores	Clock (MHz)	Shader clock (MHz)	Memory clock (MHz)
9500M GS	4	32	475	950	400
9600M GT	4	32	500	1250	400
GTX260	27	216	576	1242	999

C. Experimental results

Most of the experiments were conducted with an image resolution of 640 by 480 pixels. However, in the case of the comparison with algorithms implemented by other authors and this work, it was essential to use other resolutions. For comparing with the implementation of the *Alexandra Institute*, an image resolution of 800 by 600 pixels was used. The resolution was determined by the implementation of the *ray tracing* algorithm as it could not be modified. While for comparison with results obtained by Popov et al. [20], an image resolution of 1024 by 1024 pixels was used. The resolution was determined from the experiments described in the article by Popov et al., since the authors worked with that fixed resolution.

Experiments conducted during the evaluation confirmed that the choice of the grid size can increase the performance of image generation. As a first approximation we considered the value suggested by Thrane and Simonsen [22], which indicates that the resolution is $3\sqrt[3]{N}$ boxes along the shortest axis, where N is the number of triangles in the scene. After several tests it was found that this division is not always the best, and that a good refinement for the grid size is between $\sqrt[3]{N}$ and $3\sqrt[3]{N}$ along the shortest axis. For each of the images, it must be determined empirically the optimal grid size that obtains the better performance within the range of values.

1) Evaluation of the different GPU versions implemented: The performance comparison between different versions of the algorithm implemented on GPU was made using the test cases Pri _I, Pri _II, Pri _III, Pri_IV and Pri _V. This evaluation has two stages. In the first stage, the optimal grid size for the test cases considered is determined, while in the second stage, each one of the GPU versions are executed for each of the test cases using the optimal grid size, founded in the previous stage.

The `RT(GPU-ii)` version was used to determine the optimal grid size for each test case. The results obtained executing in the PC with a GTX260 are summarized in Table VII. The table shows the number of frames per second (FPS) that can be computed for each of the images. From these results, we can conclude that the optimal grid size for all test cases considered, is obtained by halving each axis $(2 \times 2 \times 2)$.

Table VII
FPS OF `RT(GPU-ii)` VERSION FOR DIFFERENT TEST CASES.

Scene	1x1x1	2x2x2	4x4x4	6x6x6	10x10x10	15x15x15
Pri_I	18.7	36.7	32.7	29.7	27.3	24.7
Pri_II	13.9	27.6	25.4	23.3	21.5	20.1
Pri_III	11.3	24.5	22.8	20.9	19.2	18.0
Pri_IV	8.4	18.5	18.0	16.5	15.9	15.1
Pri_V	6.7	16.6	15.5	14.6	13.8	13.5

Once the optimal grid size for each of the test cases was found, all the versions implemented on GPU are executed for the same test cases. Table VIII presents the performance obtained by the different GPU versions in the PC with a GTX260, measured in frames per second. The results show that the performance improves with the version, and that `RT(GPU-ii)` achieved the best performance.

The results obtained shown the importance of exploiting the different memory levels of GPU. In `RT(GPU)` version all the data is stored in the global memory while in the `RT(GPU-ml)` version most of the data is allocated in the texture and the constant memories. For the test cases considered, the use of the different memory levels of the GPU

Table VIII
FPS COMPARISON BETWEEN THE DIFFERENT GPU VERSIONS.

Scene	Optimal grid size	RT(GPU) (FPS)	RT(GPU-ml) (FPS)	RT(GPU-ii) (FPS)
Pri_I	2x2x2	5.8	26.2	36.7
Pri_II	2x2x2	5.1	20.2	27.6
Pri_III	2x2x2	4.9	18.2	24.5
Pri_IV	2x2x2	4.3	14.1	18.5
Pri_V	2x2x2	3.9	12.6	16.6

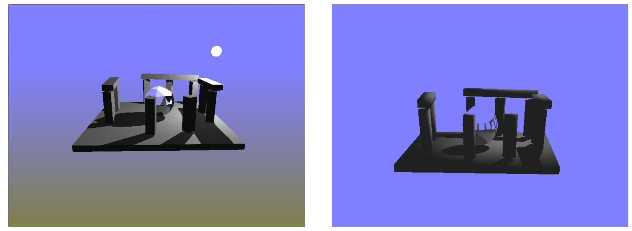
enables to improve the performance due to the reduction in memory access time, making the algorithm on average three and a half times faster than the algorithm that does not use it.

On the other hand, `RT(GPU-ii)` version improves the algorithm of ray-triangle intersection, with an algorithm that requires less arithmetic operations, thereby reducing the time needed to generate images. From the results, it is possible to notice that the improved intersection algorithm helps to make `RT(GPU-ii)` generate images 30% faster than `RT(GPU-ml)` version.

Finally, it can be seen, in Table VII as well as in Table VIII, that the increase in the number of triangles in a scene, increases the time required for generating the image and therefore reduces the FPS.

2) Comparative study with other ray tracing *implementations:* First, it was made a test to compare our proposal with a *ray tracing* implemented in GPU by *Alexandra Institute* [4] considering a scene provided with its implementation. The results obtained in the PC with a 9600M GT graphics card, showed that `RT(GPU-ii)` reach 13.0 FPS, while the implementation of the *Alexandra Institute* obtained 11.7 FPS.

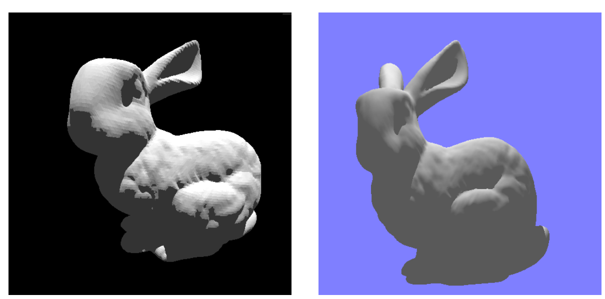
Figure 8(a) shows the rendering generated by the implementation of the *Alexandra Institute* and Figure 8(b) shows the rendering generated by our implementation for the same scene. It should be emphasized that the our implementation renders generic scenes unlike the *Alexandra Institute* implementation that was designed to produce images of a single type of scene, which generally causes that for the same scene our implementation requires more storage space and more memory accesses. Another difference in the implementations is that *Alexandra Institute* implementation considers the light sources as objects of the scene, while this was not considered in our implementation. In the images generated it can be seen subtle differences, such as the background color that could not be properly reproduced for the test case. Also there is a noticeable difference in the specular brightness of the sphere motivated by the incorrect reproduction of material used in the *Alexandra* image. On the other hand, in the image generated by our implementation it is best seen the reflection of objects near the sphere on its surface. Based on an analysis of the images, it could be concluded that there are no significant differences between the two images.



(a) (b)

Figure 8. Render of the image *Alexandra* with *Alexandra Institute* implementation (left) and with our implementation (right).

Then, a test was conducted to compare our proposal with a *ray tracing* implemented in GPU developed by Popov et al. [20] that used the kd-tree structure as spatial acceleration method. In their work, the rendering of the scene *Bunny* is presented as well as the time required for the image generation. The scene *Bunny* considered in our experiments was built from the observation of the rendering presented in the work of Popov et al.. The scene could not be exactly the same because of the limitations of the construction method used and the omission in the article of some relevant aspects of the scene. For example, the number of light sources is the same but the position is not identical and the material of the main object could not be accurately reproduced, since the article does not provide this information. The GPU used by Popov et al. is a NVIDIA GeForce 8800 GTX which has 112 cores. The GTX260 is the best suited GPU from the ones available for our experiments but has superior features. Figure 9(a) shows the render presented by Popov et al. and Figure 9(b) shows the render of the scene *Bunny* generated with our implementation.



(a) (b)

Figure 9. Render of the image *Bunny* with Popov et al. [20] implementation (left) and with our implementation (right).

The performance of our implementation of the *ray tracing* under the conditions described above is 6.1 FPS, while the performance of the work of Popov et al. is 5.9 FPS. The performance of both implementations for the test case considered is very similar. The results obtained show that the *ray tracing* implemented is competitive with other ray tracing algorithms implemented in GPU.

3) Comparative study between different platforms: The comparative study of performance between the different platforms used consisted in the execution of the `RT(GPU-ii)` version to the test cases *Dragon*, *Buddha* and *Alexandra*. In each of the platforms, the algorithm is executed for each of the test cases using various grid sizes, determining the optimal grid size for each case and platform.

Tables IX, X and XI show the results obtained for the equipment 9500M GS, 9600M GT and GTX260, respectively.

Table IX
FPS OF `RT(GPU-ii)` VERSION IN A PC WITH A 9500M GS GRAPHICS CARD.

Dragon		Buddha		Alexandra	
Grid size	FPS	Grid size	FPS	Grid size	FPS
20x20x20	1.4	20x20x20	1.3	1x1x1	4.6
46x46x46	2.3	46x46x46	2.4	3x3x3	11.5
92x92x92	2.8	92x92x92	2.5	6x6x6	15.6
138x138x138	2.0	138x138x138	2.1	10x10x10	13.1
180x180x180	1.6	180x180x180	1.7	15x15x15	12.3
230x230x230	1.3	230x230x230	1.3	30x30x30	9.1

Table X
FPS OF `RT(GPU-ii)` VERSION IN A PC WITH A 9600M GT GRAPHICS CARD.

Dragon		Buddha		Alexandra	
Grid size	FPS	Grid size	FPS	Grid size	FPS
20x20x20	1.7	20x20x20	1.4	1x1x1	5.9
46x46x46	3.3	46x46x46	2.8	3x3x3	14.0
92x92x92	3.4	92x92x92	3.1	6x6x6	20.0
138x138x138	2.6	138x138x138	2.6	10x10x10	18.4
180x180x180	2.0	180x180x180	2.1	15x15x15	17.5
230x230x230	1.6	230x230x230	1.7	30x30x30	12.8

Table XI
FPS OF `RT(GPU-ii)` VERSION IN A PC WITH A GTX260 GRAPHICS CARD.

Dragon		Buddha		Alexandra	
Grid size	FPS	Grid size	FPS	Grid size	FPS
20x20x20	5.0	20x20x20	4.9	1x1x1	28.0
46x46x46	12.2	46x46x46	9.8	3x3x3	49.3
92x92x92	16.8	92x92x92	12.4	6x6x6	71.2
138x138x138	14.2	138x138x138	11.4	10x10x10	67.6
180x180x180	11.4	180x180x180	10.3	15x15x15	62.5
230x230x230	9.3	230x230x230	8.4	30x30x30	49.4

The results obtained suggest that for the test cases considered the optimal grid size ($92 \times 92 \times 92$ in *Dragon*, and *Buddha* images, and $6 \times 6 \times 6$ in *Alexandra* image) is the same regardless of the platform used. On the other hand, the experiments confirmed that the generation of images by the *ray tracing* algorithm implemented is faster, when the GPU has a greater number of cores. In addition to this, the results also show that the algorithm implemented can automatically scale with the number of cores of the GPU, this important property arises as a consequence of the CUDA programming model that helps to achieve it very easily.

4) Comparative study between CPU and GPU implementations: The experiments for comparing the performance between the implementations for CPU and GPU used the test cases Dist_I, Dist_II, Dist_III and *Bunny*. The evaluation consisted in the execution of `RT(GPU-ii)` the most efficient version on GPU, and `RT(CPU-ii)` the CPU version of *ray tracing* in the PC with a GTX260 graphics card. Table XII presents the FPS obtained by `RT(GPU-ii)` and `RT(CPU-ii)` versions and the acceleration (defined as $\frac{\texttt{FPS of GPU implementation}}{\texttt{FPS of CPU implementation}}$) achieved by using the GPU.

Table XII
FPS OF THE CPU AND GPU IMPLEMENTATIONS.

Scene	Grid size	CPU (FPS)	GPU (FPS)	Acceleration
Dist_I	50x50x50	1.4	17.2	12.29
Dist_II	50x50x50	1.5	20.1	13.40
Dist_III	50x50x50	1.6	21.1	13.19
Bunny	80x80x80	4.2	23.8	5.67

The results obtained for the test cases Dist_I, Dist_II, Dist_III and *Bunny* show that the GPU implementation produces images in less time than the CPU implementation, and therefore achieves a higher number of frames generated per second. In particular, the GPU implementation is on average more than 11 times faster than the CPU implementation for the four test cases considered in this study.

5) Evaluation of the effect of the object distribution in the scene: The test cases Dist_I, Dist_II and Dist_III were designed for detecting a possible weakness in the spatial acceleration structure chosen, when working with scenes in which objects are not evenly distributed. We assumed that in the case of Dist_III, which has all the items concentrated in the center of the scene, would reach less FPS than in the cases Dist_I and Dist_II, which are more evenly distributed. However, the results obtained (presented in Table XII) show otherwise. One possible explanation for this behavior in such scenes, is that when the objects are more evenly distributed in the scene, they cast more shadows (as it can be seen in Figure 5). Since the calculation of the shadows is computationally expensive, this cost counteracts the benefit gained with uniform distribution of objects in the scene.

VI. CONCLUSIONS AND FUTURE WORK

This work has presented an initial study on applying GPU computing in order to speed up the execution of the *ray tracing* algorithm. Three version of *ray tracing* were implemented in GPU using CUDA and were evaluated on different platforms using several images.

The experimental analysis showed that the GPU implementation increased significantly the number of frames that could be generated per second over the traditional CPU implementation (the `RT(GPU-ii)` version obtained an acceleration of up to $13\times$). This result shows the importance of making a good use of the different levels of memory on current GPUs.

We can also conclude that the performance achieved by our proposal (`RT(GPU-ii)` version) is competitive with the state of the art in real time *ray tracing* implementations on GPU, such as the one developed by the *Alexandra Institute* and the proposal of Popov et al.. In addition to this, our proposal showed a good scalability on the platforms used in this study. This property is very important because the GPUs improve its power at a vertiginous rate, which predicts that our implementation of *ray tracing* will achieve a better performance in new GPUs.

The main line for current and future work consists in evaluating the use of a different spatial acceleration structure, being the kd-trees the structure that best seems to suit. In addition to this, extending this work to a multi-GPU scenery or a hybrid multicore-GPU approach should be addressed in order to attain real time calculation of the frames in more complex scenes.

REFERENCES

[1] Aila, T. and Laine, S., *Understanding the efficiency of ray traversal on GPUs*, Proceedings of the Conference on High Performance Graphics (HPG '09), pp. 145-149, ACM, 2009.

[2] Akenine-Möller, T. and Haines, E., *Real-time rendering*, A. K. Peters Ltd., 2002.

[3] Appel, A., *Some Techniques for Shading Machine Renderings of Solids*, Proceedings of the American Federation of Information Processing Societies (AFIPS '68), pp. 37-45, ACM, 1968.

[4] *Alexandra Institute*, Computer Graphics Group, Denmark. Available at http://cg.alexandra.dk/tag/gpgpu/. Accessed on October 2010.

[5] Avcibas, I., Sankur, B. and Sayood, K., *Statistical Evaluation of Image Quality Measures*, Journal of Electronic Imaging, vol. 11, no. 2, pp. 206-223, 2002.

[6] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P., *Brook for GPUs: Stream computing on graphics hardware*, ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2004, vol. 23, no. 3, pp. 777-786, 2004.

[7] *CUDA website*. Available at http://www.nvidia.com/object /cuda_home_new.html. Accessed on October 2010.

[8] Darema, F., *The SPMD Model: Past, Present and Future*, Proceedings of the 8th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science 2131, p. 1, 2001.

[9] *DIRECTX website*. Available at http://developer.nvidia.com/page/directx.html. Accessed on October 2010.

[10] Dirik A., Bayram, S., Sencar, H. and Memon, N., *New Features to Identify Computer Generated Images*, Proceedings of the International Conference on Image Processing (ICIP 2007), pp. 433-436, IEEE, 2007.

[11] Goral, C., Torrance, K., Greenberg, D. and Battaile, B., *Modeling the Interaction of Light Between Diffuse Surfaces*, SIGGRAPH Computer Graphics, vol. 18, no. 3, pp. 213-222, ACM, 1984.

[12] Horn, D., Sugerman, J., Houston, M. and Hanrahan, P., *Interactive K-D Tree GPU Raytracing*, Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D '07), pp. 167-174, ACM, 2007.

[13] Jensen, H., *Realistic Image Synthesis Using Photon Mapping*, A. K. Peters Ltd., 2001.

[14] Kay, T. and Kajiya, J., *Ray tracing complex scenes*, SIGGRAPH Computer Graphics, vol. 20, no. 4, pp. 269-278, ACM, 1986.

[15] Kirk, D. and Hwu, W., *Programming Massively Parallel Processors: A Hands-on Approach.*, Morgan Kaufmann, 2010.

[16] McGuire, M. and Luebke, D., *Hardware-Accelerated Global Illumination by Image Space Photon Mapping*, Proceedings of the Conference on High Performance Graphics (HPG '09), pp. 77-89, ACM, 2009.

[17] Murray, J. and Van Ryper, W., *Encyclopedia of Graphics File Formats*, O'Reilly Media, 1996.

[18] *OpenRT website*. Available at http://openrt.de/. Accessed on October 2010.

[19] Parker, S., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A. and Stich, M., *OptiX: A General Purpose Ray Tracing Engine*, ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2010, vol. 29, no. 4, pp. 1-13, 2010.

[20] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P., *Stackless KD-Tree Traversal for High Performance GPU Ray Tracing* Computer Graphics Forum - Proceedings of Eurographic, vol. 26, no. 3, pp. 415-424, 2007.

[21] *Quake Wars: Ray Traced website*. Available at http://www.qwrt.de/. Accessed on October 2010.

[22] Thrane, N. and Simonsen, L., *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*, Master's thesis, Department of Computer Science, Faculty of Science, University of Aarhus, Denmark, 2005.

[23] Wald, I., *Realtime Ray Tracing and Interactive Global Illumination*, PhD. Thesis, Computer Graphics Group, Saarland University, 2004.

[24] Ward, G., Rubinstein, F. and Clear, R., *A Ray Tracing Solution for Diffuse Interreflection*, Proceedings of the 15th annual conference on Computer graphics and interactive techniques (SIGGRAPH '88), pp. 85-92, ACM, 1988.

[25] Whitted, T., *An Improved Illumination Model for Shaded Display*, Communications of the ACM, vol. 23, no. 6, pp. 343-349, 1980.